

COSE312: Compilers

Lecture 5 — Syntax Analysis (3): Bottom-Up Parsing

Hakjoo Oh
2026 Spring

Expression Grammar

Expression grammar:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

Unambiguous version:

$$(1) \quad E \rightarrow E + T$$

$$(2) \quad E \rightarrow T$$

$$(3) \quad T \rightarrow T * F$$

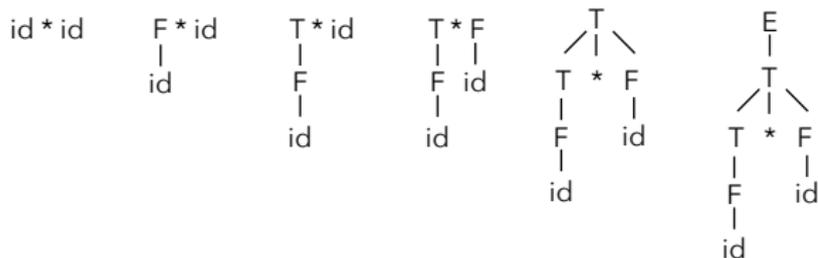
$$(4) \quad T \rightarrow F$$

$$(5) \quad F \rightarrow (E)$$

$$(6) \quad F \rightarrow \text{id}$$

Bottom-Up Parsing

- Construct a parse tree beginning at the leaves and working up towards the root.
- Ex) for input **id * id**:



- A process of “reducing” a string w to the start symbol.
- Construct the rightmost-derivation in reverse:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

Handle

- In bottom-up parsing, we have to make decisions about when to reduce and what production to apply.
- For instance, for $T * id$, we reduce id to F because reducing T does not lead to a right-sentential form.
- Handle: a substring that matches the body of a production and whose reduction leads to a right-sentential form.
- A bottom-up parsing is a process of finding a handle and reducing it.

Right Sentential Form	Handle	Reducing Production
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

LR Parsing

- The most prevalent type of bottom-up parsing.
- Handles are recognized by a deterministic finite automaton.
- LR(k)
 - ▶ “L”: Left-to-right scanning of the input
 - ▶ “R”: Rightmost-derivation in reverse
 - ▶ “k”: k-tokens lookahead
- We consider LR(0), SLR, LR(1), LALR(1) parsing algorithms.

Why LR parsing?

- Widely used:
 - ▶ Most automatic parser generators are based on LR parsing
- General and powerful:
 - ▶ $LL(k) \subseteq LR(k)$
 - ▶ Many programming languages can be described by LR grammars

LR Parsing Overview

An LR parser has a *stack* and an *input*. Based on the lookahead and stack contents, perform two kinds of actions:

- Shift
 - ▶ performed when the top of the stack is not a handle
 - ▶ move the first input token to the stack
- Reduce
 - ▶ performed when the top of the stack is a handle
 - ▶ choose a rule $X \rightarrow A B C$; pop C, B, A ; push X

Example: $id * id$

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

Stack	Input	Action
\$	id * id\$	shift
\$id	*id\$	reduce by $F \rightarrow id$
\$F	*id\$	reduce by $T \rightarrow F$
\$T	*id\$	shift
\$T*	id\$	shift
\$T * id	\$	reduce by $F \rightarrow id$
\$T * F	\$	reduce by $T \rightarrow T * F$
\$T	\$	reduce by $E \rightarrow T$
\$E	\$	shift (accept)

Recognizing Handles

By using a deterministic finite automaton. The transition table (parsing table) for the expression grammar:

State	id	+	*	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			g1	g2	g3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			g8	g2	g3
5		r6	r6		r6	r6			
6	s5			s4				g9	g3
7	s5			s4					g10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Recognizing Handles

- Given a parse state

Stack	Input
T^*	$id\$$

- Run the DFA on stack, treating shift/goto actions as edges of the DFA: $0 \rightarrow 2 \rightarrow 7$.
 - Look up the entry $(7, id)$ of the transition table: shift 5. (not a handle)
 - Push id onto the stack.
- Given a parse state

Stack	Input
$T * id$	$\$$

- Run the DFA on stack: $0 \rightarrow 2 \rightarrow 7 \rightarrow 5$.
- Look up the entry $(5, \$)$ of the transition table: reduce 6. (handle)
- Reduce by rule 6: $F \rightarrow id$

LR Parsing Process

To avoid rescanning the stack for each token, the stack maintains DFA states:

Stack	Symbols	Input	Action
0		id * id\$	shift to 5
0 5	id	*id\$	reduce by 6 ($F \rightarrow id$)
0 3	F	*id\$	reduce by 4 ($T \rightarrow F$)
0 2	T	*id\$	shift to 7
0 2 7	T^*	id\$	shift to 5
0 2 7 5	$T * id$	\$	reduce by 6 ($F \rightarrow id$)
0 2 7 10	$T * F$	\$	reduce by 3 ($T \rightarrow T * F$)
0 2	T	\$	reduce by 2 ($E \rightarrow T$)
0 1	E	\$	accept

LR Parsing Algorithm

Repeat the following:

- ① Look up top stack state, and input symbol, to get an action.
- ② If the action is
 - ▶ Shift(n): Advance input one token; push n on stack
 - ▶ Reduce(k):
 - ① Pop stack as many times as the number of symbols on the right hand side of rule k
 - ② Let X be the left-hand-side symbol of rule k
 - ③ In the state now on top of stack, look up X to get “goto n ”
 - ④ Push n on top of stack
 - ▶ Accept: Stop parsing, report success.
 - ▶ Error: Stop parsing, report failure.

LR(0) and SLR Parser Generation

For the augmented grammar

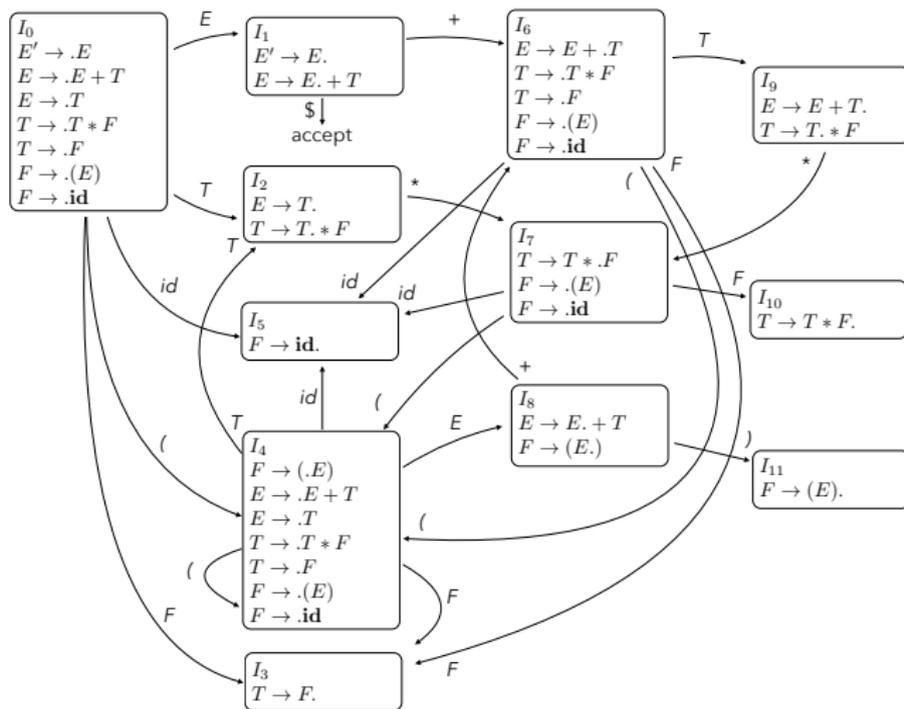
- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \text{id}$

construct the parsing table:

State	id	+	*	()	\$	E	T	F
0	s5			s4			g1	g2	g3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			g8	g2	g3
5		r6	r6		r6	r6			
6	s5			s4				g9	g3
7	s5			s4					g10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR(0) Automaton

The parsing table is constructed from the LR(0) automaton:



LR(0) Items

A state is a set of *items*.

- An item is a production with a dot somewhere on the body.
- The items for $A \rightarrow XYZ$:

$$A \rightarrow .XYZ$$

$$A \rightarrow X.YZ$$

$$A \rightarrow XY.Z$$

$$A \rightarrow XYZ.$$

- $A \rightarrow \epsilon$ has only one item $A \rightarrow \cdot$.
- An item indicates how much of a production we have seen in parsing.

The Initial Parse State

- Initially, the parser will have an empty stack, and the input will be a complete E -sentence, indicated by item

$$E' \rightarrow .E$$

where the dot indicates the current position of the parser.

- Collect all of the items reachable from the initial item without consuming any input tokens:

$$I_0 = \begin{array}{l} E' \rightarrow .E \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array}$$

Closure of Item Sets

If I is a set of items for a grammar G , then $CLOSURE(I)$ is the set of items constructed from I by the two rules:

- 1 Initially, add every item in I to $CLOSURE(I)$.
- 2 If $A \rightarrow \alpha.B\beta$ is in $CLOSURE(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to $CLOSURE(I)$, if it is not already there. Apply this rule until no more new items can be added to $CLOSURE(I)$.

In algorithm:

```
 $CLOSURE(I) =$   
  repeat  
    for any item  $A \rightarrow \alpha.B\beta$  in  $I$   
      for any production  $B \rightarrow \gamma$   
         $I = I \cup \{B \rightarrow .\gamma\}$   
  until  $I$  does not change  
  return  $I$ 
```

Construction of LR(0) Automaton

For the initial state

$$I_0 = \begin{array}{l} E' \rightarrow \cdot E \\ E \rightarrow \cdot E + T \\ E \rightarrow \cdot T \\ T \rightarrow \cdot T * F \\ T \rightarrow \cdot F \\ F \rightarrow \cdot (E) \\ F \rightarrow \cdot id \end{array}$$

construct the next states for each grammar symbol.

Consider E :

- 1 Find all items of form $A \rightarrow \alpha \cdot E \beta$: $\{E' \rightarrow \cdot E, E \rightarrow \cdot E + T\}$
- 2 Move the dot over E : $\{E' \rightarrow E \cdot, E \rightarrow E \cdot + T\}$
- 3 Closure it:

$$I_1 = \begin{array}{l} E' \rightarrow E \cdot \\ E \rightarrow E \cdot + T \end{array}$$

Construction of LR(0) Automaton

$$I_0 = \begin{array}{l} E' \rightarrow .E \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array}$$

Consider (:

- 1 Find all items of form $A \rightarrow \alpha.(\beta: \{F \rightarrow .(E)\})$
- 2 Move the dot over E : $\{F \rightarrow (.E)\}$
- 3 Closure it:

$$I_4 = \begin{array}{l} F \rightarrow (.E) \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array}$$

Exercises

$$I_0 = \begin{array}{l} E' \rightarrow .E \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array}$$

- $GOTO(I_0, T) =$
- $GOTO(I_0, F) =$

Goto

When I is a set of items and X is a grammar symbol (terminals and nonterminals, $GOTO(I, X)$ is defined to be the closure of the set of all items $A \rightarrow \alpha X \beta$ such that $A \rightarrow \alpha X \beta$ is in I .

In algorithm:

```
 $GOTO(I, X) =$   
  set  $J$  to the empty set  
  for any item  $A \rightarrow \alpha X \beta$  in  $I$   
    add  $A \rightarrow \alpha X \beta$  to  $J$   
  return  $CLOSURE(J)$ 
```

Construction of LR(0) Automaton

- T : the set of states
- E : the set of edges

Initialize T to $\{CLOSURE(\{S' \rightarrow S\})\}$

Initialize E to empty

repeat

 for each state I in T

 for each item $A \rightarrow \alpha.X\beta$ in I

 let J be $GOTO(I, X)$

$T = T \cup \{J\}$

$E = E \cup \{I \xrightarrow{X} J\}$

until E and T do not change

Construction of LR(0) Parsing Table

- For each edge $I \xrightarrow{X} J$ where X is a terminal, we put the action *shift* J at position (I, X) of the table.
- If X is a nonterminal, we put an *goto* J at position (I, X) .
- For each state I containing an item $S' \rightarrow S.$, we put an *accept* action at $(I, \$)$.
- Finally, for a state containing an item $A \rightarrow \gamma.$ (production n with the dot at the end), we put a *reduce* n action at (I, Y) for every token Y .

LR(0) Parsing Table

State	id	+	*	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	<i>s5</i>			<i>s4</i>			<i>g1</i>	<i>g2</i>	<i>g3</i>
1		<i>s6</i>				acc			
2	<i>r2</i>	<i>r2</i>	<i>r2, s7</i>	<i>r2</i>	<i>r2</i>	<i>r2</i>			
3	<i>r4</i>	<i>r4</i>	<i>r4</i>	<i>r4</i>	<i>r4</i>	<i>r4</i>			
4	<i>s5</i>			<i>s4</i>			<i>g8</i>	<i>g2</i>	<i>g3</i>
5	<i>r6</i>	<i>r6</i>	<i>r6</i>	<i>r6</i>	<i>r6</i>	<i>r6</i>			
6	<i>s5</i>			<i>s4</i>				<i>g9</i>	<i>g3</i>
7	<i>s5</i>			<i>s4</i>					<i>g10</i>
8		<i>s6</i>			<i>s11</i>				
9	<i>r1</i>	<i>r1</i>	<i>r1, s7</i>	<i>r1</i>	<i>r1</i>	<i>r1</i>			
10	<i>r3</i>	<i>r3</i>	<i>r3</i>	<i>r3</i>	<i>r3</i>	<i>r3</i>			
11	<i>r5</i>	<i>r5</i>	<i>r5</i>	<i>r5</i>	<i>r5</i>	<i>r5</i>			

Conflicts

The parsing table may contain conflicts (duplicated entries). Two kinds of conflicts:

- Shift/reduce conflicts: the parser cannot tell whether to shift or reduce.
- Reduce/reduce conflicts: the parser knows to reduce, but cannot tell which reduction to perform.

If the LR(0) parsing table for a grammar contains no conflicts, the grammar is in LR(0) grammar.

Construction of SLR Parsing Table

- For each edge $I \xrightarrow{X} J$ where X is a terminal, we put the action *shift* J at position (I, X) of the table.
- If X is a nonterminal, we put an *goto* J at position (I, X) .
- For each state I containing an item $S' \rightarrow S.$, we put an *accept* action at $(I, \$)$.
- Finally, for a state containing an item $A \rightarrow \gamma.$ (production n with the dot at the end), we put a *reduce* n action at (I, Y) for every token $Y \in FOLLOW(A)$.

SLR Parsing Table

- In state 2, we have an item $E \rightarrow T.$, so we put action $r2$ at the $(2, Y)$ entries where $Y \in FOLLOW(E) = \{\$, +,)\}$
- In state 3, we have an item $T \rightarrow F.$, so we put action $r4$ at the $(3, Y)$ entries where $Y \in FOLLOW(T) = \{\$, +,), *\}$
- In state 5, we have an item $F \rightarrow id.$, so we put action $r6$ at the $(5, Y)$ entries where $Y \in FOLLOW(F) = \{\$, +,), *\}$

State	id	+	*	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			g1	g2	g3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			g8	g2	g3
5		r6	r6		r6	r6			
6	s5			s4				g9	g3
7	s5			s4					g10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR(0) vs. SLR(1)

- LR(0) parsing makes shift/reduce decisions based solely on the DFA states, which often leads to conflicts (shift-reduce, reduce-reduce).
- SLR addresses this by using a 1-token lookahead. It checks the next token in the current state and uses that information to reduce conflicts. SLR is also called SLR(1).
 - ▶ For example, if $A \rightarrow \alpha$ is a candidate for reduction in the current state, the reduction occurs only if the next token is in *FOLLOW*(A).

Limitations of SLR

Consider the unambiguous grammar:

$$\begin{aligned} S &\rightarrow L = R \mid R \\ L &\rightarrow *R \mid \text{id} \\ R &\rightarrow L \end{aligned}$$

The LR(0) items include

$$\begin{array}{ll} I_2 : & S \rightarrow L. = R \\ & R \rightarrow L. \\ & \\ & I_6 : S \rightarrow L = .R \\ & R \rightarrow .L \\ & L \rightarrow .* R \\ & L \rightarrow .id \end{array}$$

where we put the “shift 6” action in the (2, =) entry of the parsing table. Since = is in *FOLLOW*(*R*) (since $S \Rightarrow L = R \Rightarrow *R = R$), we also put the “reduce $R \rightarrow L$ ” action in the entry. E.g.,

Stack	Input	Action
\$	id = id\$	shift
\$id	= id\$	reduce $L \rightarrow \text{id}$
\$L	= id\$	shift / reduce $R \rightarrow L$

where the correct action is shift in the context of $S \rightarrow L = R$. Reduce $R \rightarrow L$ is only possible when the next token is \$. SLR does not consider such a context.

More Powerful LR Parsers

We can extend LR(0) parsing to use one symbol of lookahead on the input:

- LR(1) parsing:
 - ▶ The parsing table is based on LR(1) items.
 - ★ E.g., ($R \rightarrow L., \$$): “reduce with $R \rightarrow L$ when the next token is $\$$ ”
(do not reduce when the next token is $=$)
 - ▶ Make full use of the lookahead symbol.
 - ▶ Generate a large set of states.
- LALR(1) parsing.
 - ▶ Based on the LR(0) items.
 - ▶ Introducing lookaheads into the LR(0) items.
 - ▶ Parsing tables have many fewer states than LR(1), no bigger than that of SLR.

Summary

