

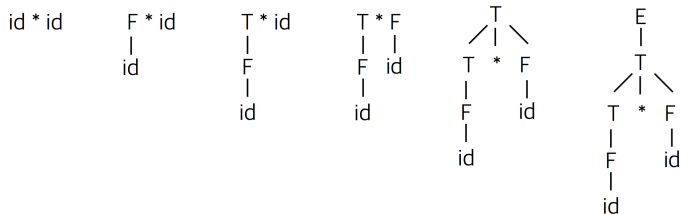
EC3204: Programming Languages and Compilers

Lecture 7 — Syntax Analysis (3): *Bottom-up Parsing*

Sunbeom So
Fall 2024

Bottom-Up Parsing

- Bottom-up parsing constructs a parse tree beginning at the leaves (the bottom) and working up towards the root (the top).
- Bottom-up parsing is a process of **reducing** a string w to the start symbol.
- **Reduction**: a step that applies a production in reverse (i.e., replace a matched body of the production with the head of that production).
- For example, given an input string **id * id**:



- Bottom-up parsing constructs the rightmost-derivation in reverse:

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

Bottom-up Parsing

Expression grammar:

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

Unambiguous, left-recursive version:

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

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

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

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

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

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

- Top-down parsing is not useful for parsing the left-recursive grammar.
- By contrast, bottom-up parsing can handle the left-recursive grammar because bottom-up parsing does not recursively (and infinitely) expand parse trees.

Handle

- In bottom-up parsing, a key challenge is in making decisions about when to reduce and what production to apply.
 - ▶ For example, given $T * id$, we reduce id to F , because we cannot accept $id * id$ by reducing T to E .
- A bottom-up parsing is a process that aims to find a **handle** and reducing it (can and must reduce at handles!).
- **Handle**: a substring that matches the body of a production and whose reduction leads to a right-sentential form.

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.
- LR(k)
 - ▶ **L**eft-to-right scanning of the input
 - ▶ **R**ightmost-derivation in reverse
 - ▶ **k**-tokens lookahead
- General, widely used:
 - ▶ $LL(k) \subseteq LR(k)$: the class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed by LL methods.
 - ▶ Most parsers are based on LR parsing.

LR Parsing Overview

An LR parser takes two primary actions, based on the current stack state and the next input symbols.

- **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

- **Shift**

- ▶ performed when the top of the stack is not a handle
- ▶ shift (move) the first input token to the stack

Example: $\text{id} * \text{id}$

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

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

cf) Conventionally, the top of the stack appears on the right in bottom-up parsing.

Recognizing Handles

Q. How to recognize handles?

A. By DFA. The corresponding 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			

LR Parsing Process

The process of parsing **id * id** using the transition table.

Stack	Symbols	Input	Action
0		id * id\$	shift to 5
0 5	id	*id\$	reduce by 6 ($F \rightarrow \text{id}$)
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	$T * \text{id}$	\$	reduce by 6 ($F \rightarrow \text{id}$)
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:

- ➊ To get an action, look up the top stack state and input symbol.
- ➋ If the action is
 - ▶ Shift(n): Advance input one token; push n on stack
 - ▶ Reduce(k):
 - ➊ Pop stack as many times as the number of symbols on the right hand side of rule k
 - ➋ Let X be the left-hand-side symbol of rule k
 - ➌ In the current state (the top of the popped stack), look up X to get "goto n "
 - ➍ Push n on the top of the stack
 - ▶ Accept: Stop parsing, report success.
 - ▶ Error: Stop parsing, report failure.

LR(0) and SLR Parser Generation

To parse the grammar,

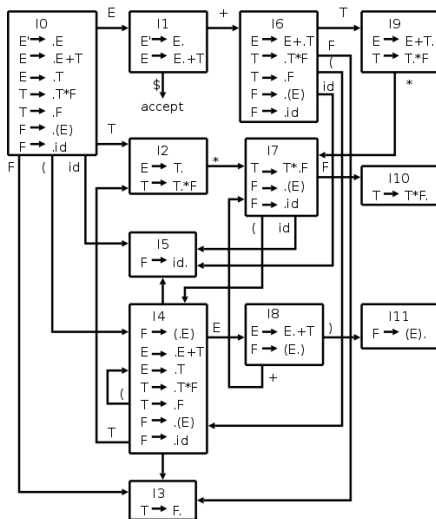
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \text{id}$

we should 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.
 - ▶ $A \rightarrow X.YZ$: we have parsed the input string derivable from X and we hope to see a string derivable from YZ .
 - ▶ $A \rightarrow XYZ.$: we have seen the body XYZ and it may be time to reduce XYZ to A .

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. In other words, we collect items that are “virtually equivalent” in terms of parsing progress.

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

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

Construction of LR(0) Automaton

From the initial state

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

we can compute its next states by shifting each grammar symbol.

For example, consider E :

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

$$I_1 = \begin{array}{lcl} E' & \rightarrow & E. \\ E & \rightarrow & E. + T \end{array}$$

Construction of LR(0) Automaton

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

As another example, consider (:

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

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

Let I be a set of items and X be 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

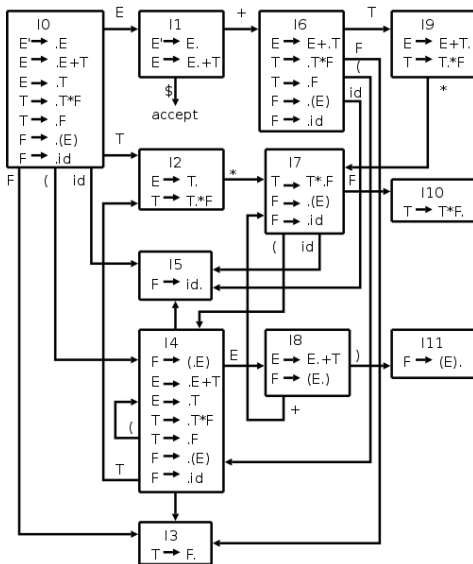
 let J be $GOTO(I, X)$

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

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

until E and T do not change

LR(0) Automaton



From LR(0) Automaton to LR(0) Parsing Table

- For each state I containing an item $S' \rightarrow S.$, we put an **accept** action at $(I, \$)$.
- For each edge $I \xrightarrow{X} J$:
 - 1 If X is a terminal,

From LR(0) Automaton to LR(0) Parsing Table

- For each state I containing an item $S' \rightarrow S.$, we put an **accept** action at $(I, \$)$.
- For each edge $I \xrightarrow{X} J$:
 - 1 If X is a terminal, we put the action **shift** J at position (I, X) of the table.
 - 2 If X is a nonterminal,

From LR(0) Automaton to LR(0) Parsing Table

- For each state I containing an item $S' \rightarrow S.$, we put an **accept** action at $(I, \$)$.
- For each edge $I \xrightarrow{X} J$:
 - ① If X is a terminal, we put the action **shift** J at position (I, X) of the table.
 - ② If X is a nonterminal, we put an **goto** J at position (I, X) .
- For a state containing an item $A \rightarrow \gamma.$ (production n with the dot at the end), for every token Y , we put a **reduce** n action at (I, Y)

LR(0) Parsing Table

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

LR(0) Parsing Table

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

Q. Do you see any problems?

LR(0) Parsing Table

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

Q. Do you see any problems?

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.

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.

Q. How can we resolve the shift/reduce conflict in the previous table?

From LR(0) Automaton to SLR¹ Parsing Table

- For each state I containing an item $S' \rightarrow S.$, we put an **accept** action at $(I, \$)$.
- For each edge $I \xrightarrow{X} J$:
 - ① If X is a terminal, we put the action **shift** J at position (I, X) of the table.
 - ② If X is a nonterminal, we put an **goto** J at position (I, X) .
- For a state containing an item $A \rightarrow \gamma.$ (production n with the dot at the end), for every token $Y \in \underline{FOLLOW(A)}$, we put a **reduce** n action at (I, Y)

¹We usually omit the “(1)” after the “SLR”.

SLR Parsing Table

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			

cf) More Powerful LR Parsers

