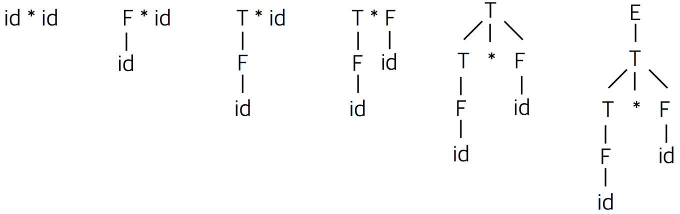
EC3204: Programming Languages and Compilers

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 $\mathbf{id} * \mathbf{id}$:



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

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Bottom-up Parsing

Expression grammar:

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

Unambiguous, left-recursive version:

$$\begin{array}{llll}
(1) & E & \rightarrow & E + T \\
(2) & E & \rightarrow & T \\
(3) & T & \rightarrow & T * F \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & (E) \\
(6) & F & \rightarrow & \mathbf{id}
\end{array}$$

- 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 $T * \mathbf{id}$, we reduce $\mathbf{id}$ to F, because we cannot accept $\mathbf{id} * \mathbf{id}$ 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
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

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 $X \to A\ B\ C$; 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: $\mathbf{id} * \mathbf{id}$

$$
\begin{array}{llll}
(1) & E & \rightarrow & E + T \\
(2) & E & \rightarrow & T \\
(3) & T & \rightarrow & T * F \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & (E) \\
(6) & F & \rightarrow & \mathbf{id}
\end{array}
$$

Stack	Input	Action
$\$$	$\mathbf{id} * \mathbf{id}\$$	shift
$\$\mathbf{id}$	$*\mathbf{id}\$$	reduce by $F \rightarrow \mathbf{id}$
$\$F$	$*\mathbf{id}\$$	reduce by $T \rightarrow F$
$\$T$	$*\mathbf{id}\$$	shift
$\$T*$	$\mathbf{id}\$$	shift
$\$T * \mathbf{id}$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$T * F$	$\$$	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	+	*	(	)	\$	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 Parsing Process

The process of parsing $\mathbf{id} * \mathbf{id}$ using the transition table.

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

LR Parsing Algorithm

Repeat the following:

1. To get an action, look up the top stack state and input symbol.
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 current state (the top of the popped stack), look up X to get "goto n"
     4. 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,

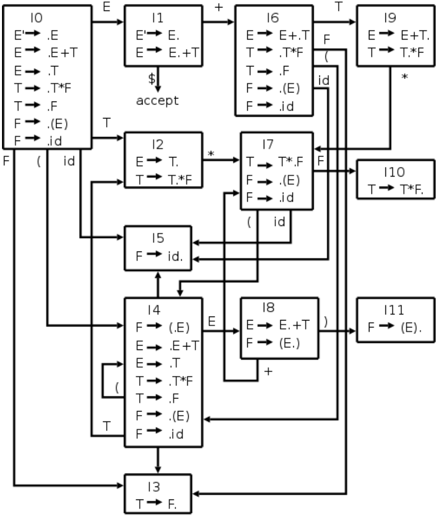
$$\begin{array}{llll}(1) & E & \rightarrow & E+T \\ (2) & E & \rightarrow & T \\ (3) & T & \rightarrow & T*F \\ (4) & T & \rightarrow & F \\ (5) & F & \rightarrow & (E) \\ (6) & F & \rightarrow & \mathrm{id}\end{array}$$

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 $A \rightarrow XYZ$:

$$\begin{array}{lcl} A & \rightarrow & .XYZ \\ A & \rightarrow & X.YZ \\ A & \rightarrow & XY.Z \\ A & \rightarrow & XYZ. \end{array}$$

- $A \rightarrow \epsilon$ 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 processed 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.

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

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

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

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:

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

Construction of LR(0) Automaton

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

As another example, consider (:

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

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

Goto

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:

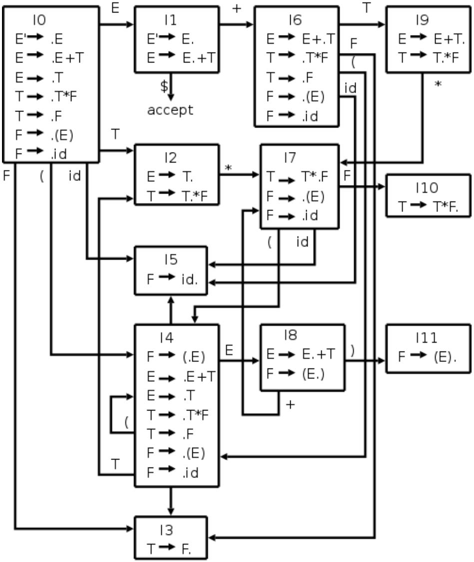
$GOTO(I, X) =$
  set J to the empty set
    for any item $A \rightarrow \alpha.X\beta$ in I
      add $A \rightarrow \alpha X.\beta$ to J
  return $CLOSURE(J)$

Construction of LR(0) Automaton

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

Initialize T to $\{CLOSURE(\{S' \to S\})\}$
Initialize E to empty
repeat
    for each state I in T
        for each item $A \to \alpha.X\beta$ 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 edge $I \xrightarrow{X} J$ where X is a terminal, we put the action <u>shift J</u> at position (I, X) of the table.
- If X is a nonterminal, we put an <u>goto J</u> at position (I, X).
- For each state I containing an item $S' \to S.$, we put an <u>accept</u> action at $(I, \$)$.
- Finally, for a state containing an item $A \to \gamma.$ (production n with the dot at the end), we put a <u>reduce n</u> action at (I, Y) for every token Y.

LR(0) Parsing Table

State	id	+	*	(	)	\$	E	T	F
0	$s5$			$s4$			$g1$	$g2$	$g3$
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

LR(0) Parsing Table

State	id	+	*	(	)	\$	E	T	F
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	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$			

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.

From LR(0) Automaton to SLR Parsing Table

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

SLR 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		$r5$	$r5$		$r5$	$r5$			

More Powerful LR Parsers