**Theory of Compilation**

**Lecture 04**

**Syntax Analysis**

**Bottom-up Parsing**

---

**You are here**

- Compiler
- Source text
- Bottom-up Parsing
- Syntax Analysis
- Lexical Analysis
- Code Generation
- Executable code

---

**Last Time**

- Parsing
  - Top-down or bottom-up
- Top-down parsing
  - Recursive descent
  - LL(k) grammars
  - LL(k) parsing with pushdown automata
- LL(k) parsers
  - Cannot deal with common prefixes and left recursion
  - Left-recursion removal might result in complicated grammar

---

**Parser Classes – Reminder**

- Top-down (predictive)
- Bottom-up (shift-reduce)

---

**LR(k) Grammars**

- A grammar is in the class LR(k) when it can be derived via:
  - Bottom-up analysis
  - Scanning the input from left to right (L)
  - Producing the rightmost derivation (R)
  - With lookahead of k tokens (k)
- A language is said to be LR(k) if it has an LR(k) grammar
- The simplest case is LR(0), which we discuss next
LR is More Powerful, But...

- Any $LL(k)$ language is also in $LR(k)$ (and not vice versa), i.e., $LL(k) \subset LR(k)$
- But the lookahead is counted differently in the two cases:
  - With $LL(k)$, the algorithm sees $k$ tokens of the right-hand side of the rule and then must select the derivation rule
  - With $LR(k)$, the algorithm sees all right-hand side of the derivation rule plus $k$ more tokens
- $LR(0)$ sees the entire right-side

The LR family of parsers is more popularly used today.

Example: a Simple LR(0) Grammar

$$E \rightarrow E \* B \mid E + B \mid B$$

- Let us number the rules:
  1. $E \rightarrow E \* B$
  2. $E \rightarrow E + B$
  3. $E \rightarrow B$
  4. $B \rightarrow 0$
  5. $B \rightarrow 1$

Goal: Reduce the Input to the Start Symbol

Example: $0 + 0 \* 1$

Stack | Input | Action
-----|-------|-------
$S$  | $E$   | shift
$E$  | $0 + 0 \* 1$ | reduce (2)
$E$  | $0 + 1$ | shift
$B$  | $0 \* 1$ | shift
$B$  | $+ 1$ | reduce (3)
$E$  | $E + B$ | reduce (2)
$E$  | $B$ | shift
$E$  | $* 1$ | reduce (4)
$E$  | $B$ | shift
$E$  | $B$ | shift
$E$  | $B$ | shift
$E$  | $E$ | accept

Shift / Reduce Parser — Intuition

- Gather input token by token
  - until we find a right-hand side of a rule
  - then, replace it with the non-terminal on the left hand side
- Going over a token and recording it in the stack is a shift
  - Each shift moves to a state that records what we’ve seen so far
- A reduce replaces a string on the stack with a nonterminal that derives it
LR(0) Item

For a production rule \( N \rightarrow \alpha \beta \) in the grammar:

- Already matched
- To be matched

\( N \rightarrow \alpha \beta \)

So far we've matched \( \alpha \), expecting to see \( \beta \)

LR(0) Item

\[
\begin{align*}
E \rightarrow & E \ast B | E + B | B \\
B \rightarrow & 0 | 1
\end{align*}
\]

- \( E \rightarrow E \ast B \) **Shift Item**
- \( E \rightarrow E \ast B \beta \) **Reduce Item**

Example: Parsing with LR(0) Items

\[
\begin{align*}
Z \rightarrow & \text{expr} \\
\text{expr} \rightarrow & \text{term} \mid \text{expr} + \text{term} \\
\text{term} \rightarrow & \text{ID} \mid (\text{expr})
\end{align*}
\]

\[
\begin{align*}
Z \rightarrow & E \$ \\
E \rightarrow & T \mid E + T \\
T \rightarrow & i \mid (E)
\end{align*}
\]

(Just a shorthand of the grammar on top)

Example: Parsing with LR(0) Items

Input

\[
\begin{align*}
Z \rightarrow & E \$ \\
E \rightarrow & T \mid E + T \\
T \rightarrow & i \mid (E)
\end{align*}
\]

Closure

\[
\begin{align*}
Z \rightarrow & E \$ \\
E \rightarrow & T \mid E + T \\
T \rightarrow & i \mid (E)
\end{align*}
\]

Reduce Item!
input $i + i \ 5$

input $i + i \ 5$

input $i + i \ 5$

input $i + i \ 5$

input $i(\ )\ 5$

input $i(\ )\ 5$

input $i(\ )\ 5$
Reducing the initial rule means accept

How does the parser know what to do?
- Pushdown Automaton!
  - A state will keep the info gathered so far
  - A table will tell it "what to do" based on current state and next token
  - Some info will be kept in a stack

Why do we need a stack?
- Suppose so far we have discovered $E \rightarrow B \rightarrow 0$ and $+$; we have constructed sentential form "$+e$".
- In the given grammar this can only mean $E \rightarrow E + B$
- Suppose current state $q_6$ represents this situation.
  - Now, the next token is 0, and we need to ignore $q_6$ for a minute, and work on $B \rightarrow 0$ to obtain $E + B$.
  - Therefore, we push $q_6$ to the stack, and after identifying $B$, we pop it to continue.
The Stack

- The stack contains states
- For readability we also include variables and tokens (the recognizer does not need them)
- The initial stack contains $q_0$ only
- Apart from $q_0$ at the bottom of the stack, the rest of the stack contains pairs of (state, token) or (state, nonterminal)

The ACTION Table

- At each step we need to decide whether to shift the next token to the stack (and move to the appropriate state) or reduce a production rule from the grammar
- The ACTION table tells us what to do based on current state and next token:
  - shift $n$: shift and move to $q_n$
  - reduce $m$: reduce according to production rule $(m)$
  (also: accept and error conditions)

The GOTO Table

- Defines what to do on reduce actions
- After reducing a right-hand side to the deriving non-terminal, we need to decide what the next state is
- This is determined by the previous state (which is on the stack) and the variable we got
- Suppose we reduce according to $N \rightarrow \beta$:
  - We remove $\beta$ from the stack, and look at the state $q$ that is now at the top. $\text{GOTO}(q, N)$ specifies the next state.

For example...

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$E$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>$s1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s5$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$r1$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$r3$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$r5$</td>
</tr>
<tr>
<td>$q_6$</td>
<td>$s7$</td>
</tr>
<tr>
<td>$q_7$</td>
<td>$r7$</td>
</tr>
</tbody>
</table>

The Algorithm, Formally

- Initialize the stack to $q_0$
- Repeat until halting:
  - Consider $\text{ACTION}(q, t)$ for $q$ at the top of stack and $t$ the next token
    - "shift $n$": Remove $t$ from the input and push $t$ and then $q_n$ to the stack.
    - "reduce $m$", where rule $(m)$ is $N \rightarrow \beta$:
      - Remove $|\beta|$ pairs from the stack; let $q$ be the state at the top of the stack.
      - Push $N$ and the state $\text{GOTO}(q, N)$ to the stack.
    - "accept$^*$": halt successfully.
  - empty cell: halt with an error.
Using LR Items to Build the Tables

• Typically a state consists of several LR items
• For example, if we identified a string that is reduced to E, then we may be in one of the following LR items:

  E \rightarrow E + B \quad \text{or} \quad E \rightarrow E^* B

• Therefore one state would be:

  q = \{ E \rightarrow E + B , E \rightarrow E^* B \}

• But if the current state includes E \rightarrow E + B, then we must allow B to be derived too — Closure!

Construct the Closure

• Proposition: a closure set of LR(0) items has the following property — if the set contains an item of the form

  A \rightarrow \alpha \cdot B \delta

then it must also contain an item

  B \rightarrow \delta

for each rule of the form B \rightarrow \delta in the grammar.

• Building the closure set for a given item set is recursive, as \delta may also begin with a variable.

Closure: an example

• The closure of the set C is

  \text{clos}(C) = \{ E \rightarrow E + B ,
  B \rightarrow \cdot 0 ,
  B \rightarrow \cdot 1 \}

• This will become another parser state

Extended Grammar

• Goal: simple termination condition
  • Assume that the initial variable only appears in a single rule.
    This guarantees that the last reduction can be (easily) detected.
  • Any grammar can be (easily) extended to have such structure.

Example: the grammar

Can be extended into

The Initial State

• To build the ACTION/GOTO table, we go through all possible states during derivation
• Each state represents a (closure) set of LR(0) items
• The initial state is \(q_0\), the closure of the initial rule
• In our example the initial rule is \(S \rightarrow \cdot E\), and therefore the initial state is

  \(q_0 = \text{clos}(S \rightarrow \cdot E) = \{ S \rightarrow \cdot E ,
  E \rightarrow E + E^* B ,
  E \rightarrow E + B ,
  E \rightarrow \cdot B ,
  B \rightarrow \cdot 0 ,
  B \rightarrow \cdot 1 \}\)

• We build all possible next states by following a single symbol (token or variable)
The Next States

• For each possible terminal or variable X, and each possible state (closure set) q,
  1. Find all items in the set of q in which the dot is before an X.
     We denote this set by q|X
  2. Move the dot ahead of the X in all items in q|X
  3. Find the closure of the obtained set:
     this is the state into which we move from q upon seeing X

• Formally, the next set of a set C and next symbol X
  \[ \text{nextSet}(C, X) = \text{clos}(\text{step}(C, X)) \]

Recall that in our example

\[ q_0 = \text{clos}\{ S \rightarrow \_E \} = \{ S \rightarrow \_E, E \rightarrow \_E * B, E \rightarrow \_E + B, E \rightarrow \_B, B \rightarrow \_0, B \rightarrow \_1 \} \]

Let us check which states are reachable from it.

States reachable from \( q_0 \) in the example

From these new states there are more reachable states

• From \( q_3, q_5, q_6 \), there are no steps because the dot is at the end of every item in their sets.

• From state \( q_3 \) we can reach the following two states —

Finally

• From \( q_7 \) we can proceed with x=0, or x=1, or x=B.
  • For x=0 we reach q_1 again and for x=1 we reach q_2.
  • For x=B we get q_7:

Similarly, from q_7 with x=B we get q_8:

These two states have no further steps. (Why?)
Building the Tables

- A row for each state.
- If \( q_j \) was obtained at \( q_i \) upon seeing \( x \), then in row \( q_i \) and column \( x \) we write \( j \).

Building the Tables: accept

- Add accept in column 5 for each state that has \( S \rightarrow E \) as an item.

Building the Tables: Shift

- Any number \( n \) in the action table becomes \( \text{shift } n \).

Building the Tables: Reduce

- For any state whose set includes the item \( A \rightarrow \alpha \), such that \( A \rightarrow \alpha \) is production rule \( \eta \), fill all columns of that state in the ACTION table with \( \text{reduce } m \).
Note on LR(0)

When a reduce is possible, we execute it without checking the next token.

GOTO/ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>q0</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>q0</td>
<td>x2</td>
<td>x2</td>
</tr>
<tr>
<td>q0</td>
<td>s1</td>
<td>s1</td>
</tr>
<tr>
<td>q0</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>q0</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>q0</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>q0</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>q0</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>q0</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>q0</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>q0</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>q0</td>
<td>r9</td>
<td>r9</td>
</tr>
<tr>
<td>q0</td>
<td>r10</td>
<td>r10</td>
</tr>
<tr>
<td>q0</td>
<td>r11</td>
<td>r11</td>
</tr>
<tr>
<td>q0</td>
<td>r12</td>
<td>r12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>q0</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>q0</td>
<td>x2</td>
<td>x2</td>
</tr>
<tr>
<td>q0</td>
<td>s1</td>
<td>s1</td>
</tr>
<tr>
<td>q0</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>q0</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>q0</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>q0</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>q0</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>q0</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>q0</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>q0</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>q0</td>
<td>r9</td>
<td>r9</td>
</tr>
<tr>
<td>q0</td>
<td>r10</td>
<td>r10</td>
</tr>
<tr>
<td>q0</td>
<td>r11</td>
<td>r11</td>
</tr>
<tr>
<td>q0</td>
<td>r12</td>
<td>r12</td>
</tr>
</tbody>
</table>

GOTO/ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>q0</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>q0</td>
<td>x2</td>
<td>x2</td>
</tr>
<tr>
<td>q0</td>
<td>s1</td>
<td>s1</td>
</tr>
<tr>
<td>q0</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>q0</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>q0</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>q0</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>q0</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>q0</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>q0</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>q0</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>q0</td>
<td>r9</td>
<td>r9</td>
</tr>
<tr>
<td>q0</td>
<td>r10</td>
<td>r10</td>
</tr>
<tr>
<td>q0</td>
<td>r11</td>
<td>r11</td>
</tr>
<tr>
<td>q0</td>
<td>r12</td>
<td>r12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>q0</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>q0</td>
<td>x2</td>
<td>x2</td>
</tr>
<tr>
<td>q0</td>
<td>s1</td>
<td>s1</td>
</tr>
<tr>
<td>q0</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>q0</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>q0</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>q0</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>q0</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>q0</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>q0</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>q0</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>q0</td>
<td>r9</td>
<td>r9</td>
</tr>
<tr>
<td>q0</td>
<td>r10</td>
<td>r10</td>
</tr>
<tr>
<td>q0</td>
<td>r11</td>
<td>r11</td>
</tr>
<tr>
<td>q0</td>
<td>r12</td>
<td>r12</td>
</tr>
</tbody>
</table>

Are we done?

- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- ...but the states are not always clear on what to do

⇒ Cannot make a deterministic ACTION table for every grammar

LR(0) Conflicts

shift/reduce conflict
• shift/reduce conflict...

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_0$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_1$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_2$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_3$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_4$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_5$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_6$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_7$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

LR(0) Conflicts

Can there be a shift/shift conflict?

View in Action/Goto Table

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_0$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_1$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_2$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_3$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_4$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_5$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_6$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_7$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

LR(0) vs. $\varepsilon$-Rules

- Whenever a nonterminal has an $\varepsilon$ production, it will be reduced as soon as it is reached in the grammar (without looking at the next token).
- If the variable has another production with a terminal prefix, there is an inherent shift/reduce conflict.

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_0$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_1$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_2$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_3$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_4$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_5$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_6$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\delta_7$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Coming Up
Reminder – Parser Classes

• Top-down (predictive)

• Bottom-up (shift-reduce)

Reminder – LR(0) Parsing

Input

Stack

Output

ACTION Table

GOTO Table

LR(0) Parsing Algorithm

LR(0) Automaton
Reminder – LR(0) Conflicts

\[ Z \rightarrow E \$ \]
\[ E \rightarrow T \]
\[ E \rightarrow E + T \]
\[ T \rightarrow i \]
\[ T \rightarrow ( E ) \]
\[ T \rightarrow \{ E \} \]

Shift/reduce conflict

Reminder – LR(0) Conflicts

\[ Z \rightarrow • E \$
\[ E \rightarrow • T \]
\[ E \rightarrow • E + T \]
\[ T \rightarrow • i \]
\[ T \rightarrow • ( E ) \]
\[ T \rightarrow • \{ E \} \]

Reduce/reduce conflict

Back to Action/Goto Table

- Remember? Reductions ignore the input...

<table>
<thead>
<tr>
<th>Rule</th>
<th>GoTo</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Z → E $</td>
<td>q0</td>
<td>s5</td>
</tr>
<tr>
<td>2. E → T</td>
<td>q1</td>
<td>s3</td>
</tr>
<tr>
<td>3. E → E + T</td>
<td>q2</td>
<td>r1</td>
</tr>
<tr>
<td>4. T → i</td>
<td>q3</td>
<td>s5</td>
</tr>
<tr>
<td>5. T → ( E )</td>
<td>q4</td>
<td>r4</td>
</tr>
<tr>
<td>6. T → { E }</td>
<td>q5</td>
<td>s7</td>
</tr>
</tbody>
</table>

SLR Grammars

- A string should only be reduced to a nonterminal N if the look-ahead is a token that can follow N.
- A reduce item N → α* is applicable only when the look-ahead is in FOLLOW(N).
- Differs from LR(0) only on the original "reduce" rows.
- Allows us to sometimes not reduce, instead shift (or do nothing = error).

GOTO/ACTION Table

<table>
<thead>
<tr>
<th>Rule</th>
<th>GoTo</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Z → E $</td>
<td>q0</td>
<td>s5</td>
</tr>
<tr>
<td>2. E → T</td>
<td>q1</td>
<td>s3</td>
</tr>
<tr>
<td>3. E → E + T</td>
<td>q2</td>
<td>r1</td>
</tr>
<tr>
<td>4. T → i</td>
<td>q3</td>
<td>s5</td>
</tr>
<tr>
<td>5. T → ( E )</td>
<td>q4</td>
<td>r4</td>
</tr>
<tr>
<td>6. T → { E }</td>
<td>q5</td>
<td>s7</td>
</tr>
</tbody>
</table>

Rule 1 can only be used with the end of input "$" sign.
The tokens that can follow E are '+' ']' and '$'.

(1)  

Now let's add “T → i [ E ]”

Now let's add “T → i [ E ]”
SLR: check next token when reducing

- Simple LR(1), or SLR(1), or SLR.
- Example demonstrates elimination of a shift/reduce conflict.
- Can eliminate reduce/reduce conflicts when conflicting rules’ left-hand sides satisfy:
  \[ \text{FOLLOW}(T) \cap \text{FOLLOW}(V) = \emptyset. \]
- But cannot resolve all conflicts.

Consider this non-LR(0) grammar

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow L = R \\
S & \rightarrow R \\
L & \rightarrow * R \\
L & \rightarrow \text{id} \\
R & \rightarrow L \\
S' & \rightarrow S \\
S & \rightarrow L = R \\
S & \rightarrow R \\
L & \rightarrow * R \\
L & \rightarrow \text{id} \\
R & \rightarrow L \\
S' & \rightarrow S \\
S & \rightarrow L = R \\
S & \rightarrow R \\
L & \rightarrow * R \\
L & \rightarrow \text{id} \\
R & \rightarrow L \\
S' & \rightarrow S \\
S & \rightarrow L = R \\
S & \rightarrow R \\
L & \rightarrow * R \\
L & \rightarrow \text{id} \\
R & \rightarrow L \\
\end{align*}
\]

Shift/reduce conflict

- \( S \rightarrow L = R \) vs. \( R \rightarrow L \).
- FOLLOW(R) contains “\( = \)”.
- SLR cannot resolve the conflict either.

Resolving the Conflict

- In SLR, a reduce item \( N \rightarrow \alpha \) is applicable when the lookahead is in \( \text{FOLLOW}(N) \).
- But there is a whole sentential form that we have discovered so far.
- We can ask what the next token may be given all previous reductions.
- For example, even looking at the FOLLOW of the entire sentential form is more restrictive than looking at the FOLLOW of the last variable.
- In a way, FOLLOW(N) merges look-ahead for all possible occurrences of N:
  \[ \text{FOLLOW}(\sigma N) \subseteq \text{FOLLOW}(N) \]
- LR(1) keeps look-ahead with each LR item.
So far we've matched $\alpha$, expecting to see $\beta$, followed by the lookahead $\sigma$.

**Example:** The production $L \rightarrow \text{id}$ yields the following LR(1) items:

- $[L \rightarrow \text{id}, \ast]$
- $[L \rightarrow \text{id}, \sigma]$
- $[L \rightarrow \text{id}, \text{id}]$
- $[L \rightarrow \text{id}, \text{else}]$
- $[L \rightarrow \ast \text{id}, \ast]$
- $[L \rightarrow \ast \text{id}, \text{else}]$
- $[L \rightarrow \ast \text{id}, \text{id}]$
- $[L \rightarrow \ast \text{id}, \text{else}]$

**Creating the states for LR(1)**

- We start with the initial state: $q_0$ will be the closure of: $(S' \rightarrow \cdot S, \$)$
- Closure for LR(1):
  - For every $(A \rightarrow \alpha \cdot B \beta, c)$ in the state:
    - For every production $B \rightarrow \delta$ and every token $b \in \text{FIRST}(\beta c)$
      - $[B \rightarrow \cdot \delta, b]$ should also be in the state

**Closure of $(S' \rightarrow \cdot S, \$)$**

- We would like to add rules that start with $S$, but keep track of possible lookahead.
- $[S' \rightarrow S, \$]$
- $[S \rightarrow L = R, \$]$  - Rules for $S$
- $[S \rightarrow R, \$]$  - Rules for $S$
- $[L \rightarrow \ast R, =]$  - Rules for $L$
- $[L \rightarrow \text{id}, =]$  - Rules for $L$
- $[L \rightarrow \ast R, \$]$  - More rules for $L$
- $[L \rightarrow \text{id}, \$]$  - More rules for $L$

**The State Machine**

- States and transitions for the LR(1) parser.
The State Machine

Building the Tables

Back to the conflict

Building the Table
**Bottom-up Parsing**
- LR(k)
- SLR
- LALR (variant of LR(1))
  - All follow the same pushdown-based algorithm
  - Differ on type of "LR Items"

<table>
<thead>
<tr>
<th>LR(0)</th>
<th>LR(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N → αβ</td>
<td>N → αβ, 0</td>
</tr>
</tbody>
</table>

**Chomsky Hierarchy**

**Grammar Hierarchy**

**Building the Parse Tree**
- Done at the time of **reduce**.

```java
{ result = new Node("E");
  result.addChild($1);
  result.addChild($2);
  result.addChild($3);
  pop(6);
  next = GOTO[stack[top-1], "E"]; push(result); push(next);
}
```

**Building the Parse Tree**
- Done at the time of **reduce**.

```java
{ result = new Node("E");
  result.addChild($1);
  result.addChild($2);
  result.addChild($3);
  pop(6);
  next = GOTO[stack[top-1], "E"]; push(result); push(next);
}
```
Building the **Abstract** Syntax Tree

- Generally — just "skip over" the creation of some internal nodes and you get an AST

```java
E -> E + B {
    $0 = new Node("+");
    $0.addChild($1);
    $0.addChild($3);
}
E -> B {
    $0 = $1;
}
E -> E * B {
    $0 = new Node("*");
    $0.value = 1;
}
```

**Summary**

- Bottom up derivation
- LR(k) can decide on a reduce after seeing the entire right side of the rule plus k look-ahead tokens.
- Particularly LR(0) — must reduce without lookahead.
- Using a table and a stack to derive.
- Definition of LR Items and the automaton.
- Creating the table from the automaton.
- LR(0), SLR, LR(1) — different kinds of LR items, same basic algorithm
- LALR: in the tutorial