Compiler Design

IT 423

Lecture - 10

Dr. Manish Khare
DAIICT, Gandhinagar
There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.

2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - Canonical LR – most general LR parser
     - LALR – intermediate LR parser (lookahead LR parser)
   - SLR, Canonical LR and LALR work same, only their parsing tables are different.
Operator-Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.

- In an *operator grammar*, no production rule can have:
  - $\varepsilon$ at the right side
  - two adjacent non-terminals at the right side.

- Ex:

  - $E \rightarrow AB$
  - $A \rightarrow a$
  - $B \rightarrow b$
  - not operator grammar

  - $E \rightarrow EOE$
  - $E \rightarrow id$
  - $O \rightarrow +|*|/$
  - not operator grammar

  - $E \rightarrow E+E |$
  - $E*E |$
  - $E/E | id$
  - operator grammar
Let G be an $\epsilon$-free operator grammar (No $\epsilon$-Production). For each terminal symbols $a$ and $b$, the following conditions are satisfies.

1. $a \equiv b$, if $\exists$ a production in RHS of the form $\alpha a \beta b \gamma$, where $\beta$ is either $\epsilon$ or a single non Terminal. $Ex\ S \rightarrow iCtSeS$ implies $i \equiv t$ and $t \equiv e$.

2. $a < b$ if for some non-terminal $A \exists$ a production in RHS of the form $A \rightarrow \alpha a A \beta$, and $A \Rightarrow^+ \gamma b \delta$ where $\gamma$ is either $\epsilon$ or a single non-terminal. $Ex\ S \rightarrow iCtS$ and $C \Rightarrow^+ b$ implies $i < b$.

3. $a > b$ if for some non-terminal $A \exists$ a production in RHS of the form $A \rightarrow \alpha A b \beta$, and $A \Rightarrow^+ \gamma a \delta$ where $\delta$ is either $\epsilon$ or a single non-terminal. $Ex\ S \rightarrow iCtS$ and $C \Rightarrow^+ b$ implies $b > t$. 
Precedence Relations

➢ In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

\[
\begin{align*}
    a \prec b & \quad \text{b has higher precedence than a} \\
    a = \cdot b & \quad \text{b has same precedence as a} \\
    a \succ b & \quad \text{b has lower precedence than a}
\end{align*}
\]

➢ These relations may appear similar to the ‘less than’, ‘equal to’ and ‘greater than’ operator.

➢ The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators.
How to Create Operator-Precedence Relations

➢ We use associativity and precedence relations among operators.

1. If operator $O_1$ has higher precedence than operator $O_2$, 
   $\Rightarrow O_1 \cdot O_2$ and $O_2 \cdot O_1$

2. If operator $O_1$ and operator $O_2$ have equal precedence, 
   they are left-associative $\Rightarrow O_1 \cdot O_2$ and $O_2 \cdot O_1$ 
   they are right-associative $\Rightarrow O_1 \cdot O_2$ and $O_2 \cdot O_1$

3. For all operators $O$, 
   $O \cdot id, \ id \cdot O, \ O \cdot (, (\cdot O, O \cdot ), ) \cdot O, \ O \cdot $, and $\cdot $ < $O$

4. Also, let 
   (\cdot ) $\cdot ( id \cdot )$ ) \cdot > $)
   ( \cdot ( $\cdot id id \cdot $) ) \cdot >$)
   ( \cdot id
<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>^</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>-</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>/</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>^</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>id</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>=</td>
</tr>
<tr>
<td>)</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;·</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
<td>&lt;·</td>
</tr>
</tbody>
</table>
Handling Unary Operators

➢ Operator-Precedence parsing cannot handle the unary minus when we also the binary minus in our grammar.

➢ The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - The lexical analyzer will need a lookahead to distinguish the binary minus from the unary minus.

➢ Then, we make

\[ O < \text{unary-minus} \]

for any operator

\[ \text{unary-minus} \rightarrow O \]

if unary-minus has higher precedence than O

\[ \text{unary-minus} \leftarrow O \]

if unary-minus has lower (or equal) precedence than O
Compilers using operator precedence parsers do not need to store the table of precedence relations.

The table can be encoded by two precedence functions $f$ and $g$ that map terminal symbols to integers.

For symbols $a$ and $b$.

- $f(a) < g(b)$ whenever $a < b$
- $f(a) = g(b)$ whenever $a = b$
- $f(a) > g(b)$ whenever $a > b$
Using Operator-Precedence Relations

➢ The intention of the precedence relations is to find the handle of a right-sentential form,

\[<· \text{with marking the left end,} \]

\[=· \text{appearing in the interior of the handle, and} \]

\[·\rangle \text{marking the right hand.} \]

➢ To be more precise, suppose we have a right-sentential form of an operator grammar.

➢ The fact no adjacent non-terminals appear on the right sides of productions implies that no right-sentential form will have two adjacent non-terminals either.

➢ Thus, we may write the right-sentential form as \( \beta_0 a_1 \beta_1 a_2 \ldots a_n \beta_1 \), where \( \beta_i \) each is either \( \varepsilon \) (the empty string) or a single non-terminal, and each \( a_i \) is a single terminal.
Using Operator-Precedence Relations

➢ Suppose that between $a_i$ and $a_{i+1}$ exactly one of the relations $\prec$, $\equiv$, and $\succ$ holds. Further let us use $\$\$ to mark each end of the string, and define $\$ \prec \$ B and $B \succ \$.

➢ Now suppose we remove the non-terminals from the string and place the correct relation $\prec$, $\equiv$, and $\succ$, between each pair of terminals and between endmost terminals and the $\$’s marking the ends of the strings.

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E^E \mid (E) \mid -E \mid id$

The partial operator-precedence table for this grammar:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>+</td>
<td>$&lt;$</td>
<td>$&gt;$</td>
<td>$&lt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>*</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>$$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
</tr>
</tbody>
</table>

Then the input string ‘id+id*id’ with the precedence relations inserted will be:

$\$ \prec id \succ + \prec id \succ \succ id \succ \succ \succ \succ \succ id \succ \succ \succ \succ \succ \succ \succ \succ \succ $
To Find The Handles

1. Scan the string from left end until the first $\to$ is encountered.

2. Then scan backwards (to the left) over any $\to$ until a $\leftarrow$ is encountered.

3. The handle contains everything to left of the first $\to$ and to the right of the $\leftarrow$ is encountered.

$\leftarrow \text{id} \to + \leftarrow \text{id} \to * \leftarrow \text{id} \to \$$
$\leftarrow + \leftarrow \text{id} \to * \leftarrow \text{id} \to \$
$\leftarrow + \leftarrow \cdot \leftarrow \text{id} \to \$
$\leftarrow + \leftarrow \cdot \leftarrow \cdot \to \$
$\leftarrow + \leftarrow \leftarrow \text{id} \to \$
$\$$

$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$

$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$

$E \to E*E$
$E \to E+E$
$E \to E+E$
$E \to E+E$
$E \to E+E$
$E \to E+E$

$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$
$E \to \text{id}$

$\text{id} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$

$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$
$\text{E} + \text{id} * \text{id}$$

$\text{E}$
$\text{E}$
$\text{E}$
$\text{E}$
$\text{E}$
$\text{E}$

$\text{E}$
$\text{E}$
$\text{E}$
$\text{E}$
$\text{E}$
$\text{E}$
Operator-Precedence Parsing Algorithm

The input string is $w$, the initial stack is $\$$ and a table holds precedence relations between certain terminals

**Algorithm:**

set $p$ to point to the first symbol of $w$;

repeat forever

if ($\$$ is on top of the stack and $p$ points to $\$$) then return

else {

let $a$ be the topmost terminal symbol on the stack and let $b$ be the symbol pointed to by $p$;

if ($a < b$ or $a \equiv b$) then {

/* SHIFT */

push $b$ onto the stack;

advance $p$ to the next input symbol;

}

else if ($a > b$) then

/* REDUCE */

repeat pop stack

until (the top of stack terminal is related by $<$ to the terminal most recently popped);

else error();

}
Operator-Precedence Parsing Algorithm - Example

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id

Stack | Input | Action
--- | --- | ---
$ | id+id*id$ | $< id shift$
$id | +id*id$ | id $>$ + reduce E $\rightarrow$ id
$ | +id*id$ | $< + shift$
$+$ | id*id$ | $< id shift$
$+$ | *id$ | id $>$ * reduce E $\rightarrow$ id
$+$ | *id$ | + $<$ * shift
$+$ | id$ | * $<$ id shift
$+$ | id | id $>$ $ reduce E $\rightarrow$ id
$+$ | *id | * $>$ $ reduce E $\rightarrow$ E*E
$+$ | id | id $>$ $ reduce E $\rightarrow$ E+E
$ | id | accept
Disadvantages of Operator Precedence Parsing

➢ Disadvantages:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

➢ Advantages:
  - simple
  - powerful enough for expressions in programming languages
LR Parser

• The most powerful shift-reduce parsing (yet efficient) is:
  
  LR(k) parsing.

  - left to right scanning
  - right-most derivation
  - k lookahead (k is omitted → it is 1)

• LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
    LL(1)-Grammars ⊂ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.