Compiler Design

IT 423

Lecture - 11

Dr. Manish Khare
DAIICT, Gandhinagar
LR Parser

- The most powerful shift-reduce parsing (yet efficient) is:
  \[ \text{LR}(k) \text{ parsing.} \]
  \[
  \begin{align*}
  \text{left to right} & \quad \text{right-most} & \quad \text{k lookahead} \\
  \text{scanning} & \quad \text{derivation} & \quad (k \text{ is omitted } \Rightarrow \text{ it is 1})
  \end{align*}
  \]

- 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.
    \[ \text{LL}(1)-\text{Grammars} \subset \text{LR}(1)-\text{Grammars} \]
  - 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.
LR Parser

➢ **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

➢ **Disadvantage**
  - This technique have too much work is needed to construct an LR parser by hand for a typical programming language.
Techniques for LR Parser

➢ Three techniques for LR Parser

- The first method, called Simple LR (SLR), is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammar on which the other method succeed.

- The second method, called Canonical LR (CLR), is most powerful and work on very large class of grammars. It is very expensive and difficult to implement.

- The third method, called Look Ahead LR (LALR), is intermediate in power between SLR and CLR. It works on most of the class of grammar and with some efforts it can be implemented easily.
Structure of LR Parser

LR Parsing Algorithm

Input: \( a_1 \ldots a_i \ldots a_n \) $ \\
Output: \\
Stack: \( S_m \rightarrow X_m \rightarrow S_{m-1} \rightarrow X_{m-1} \rightarrow \ldots \rightarrow S_1 \rightarrow X_1 \rightarrow S_0 \)

Action Table: 
- terminals and $ 
- four different actions

Goto Table: 
- non-terminal 
- each item is a state number

Note: Each state number corresponds to a non-terminal.
LR Parser consists

- An input
- An output
- A stack
- A driving program
- A parsing table
  - Action
  - Goto
A stack is used by a LR Parser to store a string of the form $S_0X_1S_1X_2\ldots S_mX_m$, where $X_i$ is the grammar symbol, $S_i$ is a state symbol and $S_m$ is on TOP.

Parsing table consists of two parts

- **Action**: is a table of size $n \times m$ where $n$ is the total number of states of the parser and $m$ is the number of terminal symbol in $T(G)$ including $\$$.  
- **Goto**: is a table of size $n \times p$ where $p$ is number of non-terminal symbols in $N(G)$. Function Goto takes a state and grammar symbol as arguments and produces a state.
The program driving the LR parser behaves as follows:

- It determines $S_m$, the state currently on the top of the stack, and $a_i$ the current input symbol. It then consults $\text{ACTION } [S_m, a_i]$ the parsing action table entry for state $S_m$ and input $a_i$.

- The entry $\text{ACTION } [S_m, a_i]$ can have one of the four possible entries:
  - $S_i$, which means shift to state $i$
  - $R_j$, which means reduce by using the $j$th rule.
  - $\text{acc}$, which means accept
  - $e$, which means error
A configuration of a LR parsing is:

\[( S_0 \ X_1 \ S_1 \ldots \ X_m \ S_m, \ a_i \ a_{i+1} \ldots \ a_n \, \$ ) \]

- \( S_m \) and \( a_i \) decides the parser action by consulting the parsing action table. (Initial Stack contains just \( S_0 \) )

- A configuration of a LR parsing represents the right sentential form:

\[ X_1 \ldots X_m \ a_i \ a_{i+1} \ldots \ a_n \, \$ \]
1. IF ACTION \([S_m, a_i] = \text{Shift } s\)
the parser executes a shift move, entering the configuration

\[
(S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n $) \Rightarrow (S_o X_1 S_1 ... X_m S_m s, a_{i+1} ... a_n $)
\]

Here the parser has shift both the current input symbol \(a_i\) and the next state \(S\), which is given in ACTION \([S_m, a_i]\) onto stack, and \(a_{i+1}\) becomes the current input symbol.

2. IF ACTION \([S_m, a_i] = \text{reduce } A \rightarrow \beta\)
then the parser executes a reduce move, entering the configuration

\[
(S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n $) \Rightarrow (S_o X_1 S_1 ... X_{m-r} S_{m-r} A s, a_i ... a_n $)
\]

Where \(s=\text{goto}[s_{m-r}, A]\) and \(r\) is the length of \(\beta\), the right side of the production.
3. IF ACTION \([S_m, a_i]\) = Accept
Parsing successfully completed

4. IF ACTION \([S_m, a_i]\) = Error
The parser has discovered an error and calls error recovery routine.
Construction of SLR or LR(0) Parser