Bottom-up parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). It is convenient to describe parsing as the process of building parse trees.
  For ex, consider grammar below,
  
  \[
  E \rightarrow E + T \mid T \\
  T \rightarrow T * F \mid F \\
  F \rightarrow ( E ) \mid \text{id}
  \]
  
  The sequence of tree construction is showed below to parse id * id.

- The general way bottom-up parser implemented is shift-reduce parsing. The largest class of grammars for which shift-reduce parsers can be built are LR grammars. In general, building LR grammars might take too much work by hand, thus tools called automatic parser generators make it easy to construct efficient LR parsers.

- Reductions

  - The bottom-up parsing can be thought as the process of reducing a string ‘w’ to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.
    
    For ex, the expression id * id, F * id, T * id, T * F, T, E.

  - A ‘handle’ is a substring that matches the body of a production and whose reduction represents one step along the reverse of rightmost derivation.
    
    For ex, during a parse of id \(_1\) * id \(_2\),

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
### Shift-reduce parsing

- It is a form of bottom-up parsing in which a stack holds grammar symbols, an input buffer holds the rest of the string to be parsed and the handle always appears at the top of the stack.

- The $ symbol is used to mark the bottom of the stack and also the right-end of the input. Initially, the stack is empty and string 'w' is on the input, as follows,

```
Stack      Input
$          w$
```

- During a left to right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. The parser repeats the cycle until it has detected an error or until the stack contains the start symbol and the input is empty, as showed below.

```
Stack      Input
$ S        $
```

- Consider below grammar, with the input string id₁ * id₂ and the steps might take in parsing the input using shift-reduce parser.

\[
\begin{align*}
E & \rightarrow E + T | T \\
T & \rightarrow T * F | F \\
F & \rightarrow ( E ) | id
\end{align*}
\]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id₁ * id₂ $</td>
<td>Shift</td>
</tr>
<tr>
<td>$ id₁</td>
<td>* id₂ $</td>
<td>Reduce F → id</td>
</tr>
</tbody>
</table>

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
Following are the operations used during shift-reduce parser:

a) Shift - Shifts the next input symbol onto the Top of the stack

b) Reduce - Locate the left end of the string within the stack and decide with the non-terminal to replace the string.

c) Accept - Announce successful completion of parsing

d) Error - Discover a syntax error and call a recovery routine.

### Conflicts during Shift-reduce parsing

- There are some CFGs for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and next input symbol, cannot decide whether to shift or to reduce, which is a shift/reduce conflict and cannot decide which of several reductions to make, which is a reduce/reduce conflict. These grammars are referred to as non-LR grammars.

- An ambiguous grammar can never be LR. For ex, consider below grammar,

  \[ \text{stmt} \rightarrow \text{if expr then stmt} \mid \text{if expr then stmt else stmt} \mid \text{other} \]

  Consider a situation showed below,

  Stack | Input
  -------|-------
  ... if expr then stmt | else ...

  Here there is a shift/reduce conflict, since, it might correct to shift \text{if expr then stmt} to else or to reduce it to stmt.
LR parsing

- The LR (k) parsing, 'L' is for left-to-right scanning of the input, the 'R' for constructing a right-most derivation in reverse and the 'k' for the number of input symbols of look-ahead that are used in making parsing decisions. The easiest method for constructing shift-reduce parsers are Simple LR or SLR.

- Items

LR parser makes shift-reduce decisions by maintaining states to keep track of parsing. States represent sets of items. An LR (0) item of a grammar G is a production of a grammar G is a production of G with a dot at some position of the body. For ex, the production $A \rightarrow X Y Z$, yields four items as showed below.

$$A \rightarrow \cdot X Y Z$$
$$A \rightarrow X \cdot Y Z$$
$$A \rightarrow X Y \cdot Z$$
$$A \rightarrow X Y Z \cdot$$

The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$. This indicates that how much of a production that has seen at a given point in the parsing process. For ex, $A \rightarrow \cdot X Y Z$ indicates that it is hoped to see a string derivable from $XYZ$ next on the input. Another production, $A \rightarrow X \cdot Y Z$ indicates that a string derived from $X$ has been seen and it is hoped to see a string derivable from $Y Z$. One collection of sets of LR (0) items, called the canonical LR (0) collection provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called LR (0) automaton.

- 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 two rules:

a) Initially, add every item in I to CLOSURE (I).

b) If $A \rightarrow \alpha \cdot B \beta$ is in CLOSURE (I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to CLOSURE (I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE (I).

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
Ex: Consider below grammar with the augmented expression grammar and the closure of the item sets are showed below.

\[
E' \rightarrow E \\
E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
E \rightarrow (E) | id
\]

- **Goto function**

GOTO \((I, X)\) is defined to be the closure of the set of all items \([A \rightarrow \alpha X \cdot \beta]\) such that \([A \rightarrow \alpha \cdot X \beta]\) is in \(I\), for \(I\) is the set of items and \(X\) is a grammar symbol.

Ex: If \(I\) is the set of two items \(\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}\), then GOTO \((I, +)\) contains the following items,

\[
E \rightarrow E + \cdot T \\
T \rightarrow \cdot T * F \\
T \rightarrow \cdot F \\
F \rightarrow \cdot (E) \\
F \rightarrow \cdot id
\]

- Below table illustrates the actions of a shift-reduce parser on input id * id, using the LR(0) automaton showed above.

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
LR-parsing algorithm

- The symbolic representation of an LR parser is showed below. It contains an input, an output, a stack, a driver program and a parsing table that has two parts, ACTION and GOTO. The driver program is the same for all LR parsers, only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time, where a shift-reduce parser would shift a symbol, an LR parser shifts a state.

```
<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>id * id $</td>
<td>Shift to 5</td>
</tr>
<tr>
<td>0 5</td>
<td>$ id</td>
<td>* id $</td>
<td>Reduce by F → id</td>
</tr>
<tr>
<td>0 3</td>
<td>$ F</td>
<td>* id $</td>
<td>Reduce by T → F</td>
</tr>
<tr>
<td>0 2</td>
<td>$ T</td>
<td>* id $</td>
<td>Shift to 7</td>
</tr>
<tr>
<td>0 2 7</td>
<td>$ T *</td>
<td>id $</td>
<td>Shift to 5</td>
</tr>
<tr>
<td>0 2 7 5</td>
<td>$ T * id</td>
<td>$</td>
<td>Reduce by F → id</td>
</tr>
<tr>
<td>0 2 7 10</td>
<td>$ T * F</td>
<td>$</td>
<td>Reduce by T → T * F</td>
</tr>
<tr>
<td>0 2</td>
<td>$ T</td>
<td>$</td>
<td>Reduce by E → T</td>
</tr>
<tr>
<td>0 1</td>
<td>$ E</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
```

Each state summarizes the information contained in the stack below it. The stack holds a sequence of states $s_0$, $s_1$, ..., $s_m$, $s_m$ on the top.

- Structure of the LR parsing table

The parsing table contains two parts: a) parsing-action function ACTION and b) goto function GOTO.

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
a) The ACTION function takes as arguments a state i and a terminal ‘a’ (or $, the input end-marker). The value of ACTION [i, a] can have one of four forms:

- Shift ‘j’, the action taken by the parser effectively shifts input ‘a’ to the stack, but uses state j to represent ‘a’.
- Reduce A → β. The action of the parser effectively reduces β on the top of the stack to head A.
- Accept. The parser accepts the input and finishes parsing.
- Error. The parser discovers an error in its input and takes some corrective action.

b) The GOTO function can extend, defined on sets of items, to states, if GOTO [I_i, A] = I_j, then GOTO also maps a state ‘i’ a non-terminal A to state j.

Ex: Consider the grammar below and the table shows the ACTION and GOTO functions of an LR-parsing table for the grammar.

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

The codes for the actions are: S_i means shift and stack state ‘i’, R_j, means reduce by the production ‘j’, acc means accept and blank means error.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>S_5</td>
<td>S_4</td>
</tr>
<tr>
<td>1</td>
<td>S_6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R_2</td>
<td>S_7</td>
</tr>
<tr>
<td>3</td>
<td>R_4</td>
<td>R_4</td>
</tr>
<tr>
<td>4</td>
<td>S_5</td>
<td>S_4</td>
</tr>
<tr>
<td>5</td>
<td>R_6</td>
<td>R_6</td>
</tr>
<tr>
<td>6</td>
<td>S_5</td>
<td>S_4</td>
</tr>
<tr>
<td>7</td>
<td>S_5</td>
<td>S_4</td>
</tr>
<tr>
<td>8</td>
<td>S_6</td>
<td></td>
</tr>
</tbody>
</table>

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
Consider an input id * id + id, the sequence of stack and input contents are as showed below

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id</td>
<td>* id + id $</td>
<td>Shift</td>
</tr>
<tr>
<td>0 5</td>
<td>id</td>
<td>* id + id $</td>
<td>Reduce F → id</td>
</tr>
<tr>
<td>0 3</td>
<td>F</td>
<td>* id + id $</td>
<td>Reduce T → F</td>
</tr>
<tr>
<td>0 2</td>
<td>T</td>
<td>* id + id $</td>
<td>Shift</td>
</tr>
<tr>
<td>0 2 7</td>
<td>T *</td>
<td>id + id $</td>
<td>Shift</td>
</tr>
<tr>
<td>0 2 7 5</td>
<td>T * id</td>
<td>+ id $</td>
<td>Reduce F → id</td>
</tr>
<tr>
<td>0 2 7 10</td>
<td>T * F</td>
<td>+ id $</td>
<td>Reduce T → T * F</td>
</tr>
<tr>
<td>0 2</td>
<td>T</td>
<td>+ id $</td>
<td>Reduce E → T</td>
</tr>
<tr>
<td>0 1</td>
<td>E</td>
<td>+ id $</td>
<td>Shift</td>
</tr>
<tr>
<td>0 1 6</td>
<td>E +</td>
<td>id $</td>
<td>Shift</td>
</tr>
<tr>
<td>0 1 6 5</td>
<td>E + id</td>
<td>$</td>
<td>Reduce F → Id</td>
</tr>
<tr>
<td>0 1 6 3</td>
<td>E + F</td>
<td>$</td>
<td>Reduce T → F</td>
</tr>
<tr>
<td>0 1 6 9</td>
<td>E + T</td>
<td>$</td>
<td>Reduce E → E + T</td>
</tr>
<tr>
<td>0 1</td>
<td>E</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Constructing SLR parsing table

- The SLR method begins with LR (0) items and LR (0) automata. Given a grammar ‘G’, augmented grammar G’ is constructed using a new start symbol S’. From G’, the canonical collection of sets of items for G’ together with GOTO function is constructed. Following algorithm is used to construct ACTION and GOTO entries in the parsing table, which uses FOLLOW (A) for each non-terminal A of a grammar.

a) Construct C = {I₀, I₁, ..., Iₙ}, the collection of sets of LR (0) items for G’.

b) State ‘i’ is constructed from Iᵢ. The parsing actions for state ‘i’ are determined as follows:

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
• If \([A \rightarrow \alpha \cdot a \beta] \) is in \( I_i \) and GOTO \((I_i, a) = I_j \), then set \( \text{ACTION}[i, a] \) to shift \( 'j' \), 'a' is a terminal.

• If \([A \rightarrow \alpha \cdot] \) is in \( I_i \), then set \( \text{ACTION}[i, a] \) to reduce \( A \rightarrow \alpha \) for all \( a \) in \( \text{FOLLOW}(A) \), here \( A \) may not be \( S' \).

• If \([S' \rightarrow S \cdot] \) is in \( I_i \), then set \( \text{ACTION}[i, \$] \) to accept.

If any conflicting actions result from the above rules, the algorithm fails and the grammar is not SLR (1).

c) The goto transitions for state \( 'i' \) are constructed for all non-terminals \( 'A' \) using the following rule:

\[
\text{If GOTO}(I_i, A) = I_j, \text{ then GOTO}[i, A] = j.
\]

d) All entries that are undefined by rules (b) & (c) are made error.

e) The initial state of the parser is the one constructed from the set of items containing \([S' \rightarrow S]\)

- The parsing table consisting of the ACTION and GOTO functions determined by above algorithm is called SLR (1) table for \( 'G' \). An LR parser using the SLR (1) table for \( G \) is called the SLR (1) parser for \( 'G' \) and grammar having an SLR (1) parsing table is said to be SLR (1).

- Ex: Consider below grammar, to construct SLR table for the augmented expression grammar.

The set of items \( I_0 \) are as showed below.

\[
\begin{align*}
E' & \rightarrow \cdot E \\
E & \rightarrow \cdot E + T \\
E & \rightarrow \cdot T \\
T & \rightarrow \cdot T * F \\
T & \rightarrow \cdot F \\
F & \rightarrow \cdot (E) \\
F & \rightarrow \cdot id
\end{align*}
\]

For terminals, the item \( F \rightarrow \cdot (E) \) giving the entry \( \text{ACTION}[0, \!] = \text{Shift 4} \) and \( F \rightarrow \cdot id \) to the entry \( \text{ACTION}[0, \text{id}] = \text{Shift 5} \). Other items in \( I_0 \) yield no actions.

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
Consider item $I_2$,

\[
E \rightarrow T \cdot \\
T \rightarrow T \cdot * F
\]

Since $\text{FOLLOW}(E) = \{\$, +, \})$, the first item makes,

\[
\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2, \}) = \text{Reduce } E \rightarrow T
\]

The second item makes, $\text{ACTION}[2, *] = \text{Shift } 7$. Continuing in the same manner, the table showed above is obtained.

- **Viable prefixes**
  
  The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. For ex, an item, $A \rightarrow \beta_1 \beta_2$ is a valid for a viable prefix $\alpha \beta$, if there is a derivation $S' \Rightarrow_m \alpha A w \Rightarrow_m \alpha \beta_1 \beta_2 w$.

**Canonical LR (1) items**

- The extra information is incorporated into the state by redefining items to include a terminal symbols as a second component. The general form of such an item is $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha \beta$ is a production and '$a'$ is a terminal or the right end-marker $.$
  
  The look-ahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, for $\beta$ is not $\epsilon$, but for an item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if next input is '$a'$.

- Following are the modified steps to construct the CLOSURE and GOTO procedures.
  
  - Let $G'$ be the augmented grammar of given grammar $G$, initialize the first item to CLOSURE ($[S' \rightarrow \cdot S, \$]$)
  
  - For each item $[A \rightarrow \alpha \cdot B \beta, a]$ in $I'$, for each production $B \rightarrow \gamma$ in $G'$ and for each terminal $b$ in FIRST ($\beta a$), add $[B \rightarrow \cdot \gamma, b]$ to set $I$, until no more items are added to $I$.
  
  - For GOTO ($I$, $X$), for each item $[A \rightarrow \alpha \cdot X \beta, a]$ in $I$, add item $[A \rightarrow \alpha X \cdot \beta, a]$ to set $I'$.

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
Ex: Consider the grammar, for which the LR (1) sets of items are constructed below.

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow C \ C \\
C & \rightarrow c \ C \mid d
\end{align*}
\]

Initially, consider the closure of \([S' \rightarrow \cdot S, \$]\), by comparing with the item, \([A \rightarrow \alpha \cdot B \beta, \ a]\) in the above algorithm, \(A = S', \ \alpha = \epsilon, \ B = S, \ \beta = \epsilon\) and \(a = \$\). Now, for each production, \(B \rightarrow \gamma\), add \([B \rightarrow \cdot \gamma, \ b]\), where \(b\) in FIRST (\(\beta a\)) = \{$$\}. So, add the items \([S' \rightarrow \cdot S, \$]\) and \([S \rightarrow \cdot C \ C, \$]\) in the set \(I_0\). For the item, \([S \rightarrow \cdot C \ C, \$]\), \(A = S, \ \alpha = \epsilon, \ B = C\) and \(\beta = C\). Now, the FIRST (\(\beta a\)) = FIRST (\(C\$\)) = \{c, d\}. So, add the items \([C \rightarrow \cdot c \ C, c], \ [C \rightarrow cC, d]\), \([C \rightarrow \cdot d, c]\) and \([C \rightarrow \cdot d, d]\) in set \(I_0\). Thus, the initial set of items is

\[
I_0 : S' \rightarrow \cdot S, \$
\]

\[
\begin{align*}
S & \rightarrow \cdot C \ C, \$ \\
C & \rightarrow \cdot c \ C, c \mid d \\
C & \rightarrow \cdot d, c \mid d
\end{align*}
\]

Similar, by continuing for all the sets of items the above transition diagram is obtained.
Constructing Canonical LR (1) parsing tables

Let \( G' \) be the augmented grammar, the construction of canonical-LR parsing tables is as follows.

a) Construct \( C' = \{I_0, I_1, \ldots, I_n\} \) the collection of sets of LR (1) items for \( G' \)
b) State 'i' of the parser is constructed from \( I_i \). The parsing action for state 'i' is determined as follows.
   - If \([A \to \alpha \cdot a \beta, b] \) is in \( I_i \) and GOTO \((I_i, a) = I_j \), then set ACTION \([i, a]\) to Shift \( j \).
     Here 'a' must be a terminal.
   - If \([A \to \alpha \cdot, a] \) is in \( I_i \), then set ACTION \([i, a]\) to reduce \( A \to \alpha \).
   - If \([S' \to S \cdot, \$] \) is in \( I_i \), then set ACTION \([i, \$]\) to "accept".

If any conflicting actions result from above rules, then the grammar in not in LR (1).

c) The goto transitions for state i are constructed for all non-terminals A using the rule: If GOTO \((I_i, A) = I_j \), then goto \( \text{GOTO}[i, A] = j \)
d) All entries not defined by rules (2) & (3) are made "error".

e) The initial state of the parser is the one constructed from the set of items containing \([S' \to \cdot S, \$] \)

Ex: Consider the above grammar,

1. \( S \to C C \)
2. \( C \to c C \)
3. \( C \to d \)

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S₃</td>
<td>S₄</td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S₆</td>
<td>S₇</td>
</tr>
<tr>
<td>3</td>
<td>S₃</td>
<td>S₄</td>
</tr>
<tr>
<td>4</td>
<td>R₃</td>
<td>R₃</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>R₁</td>
</tr>
<tr>
<td>6</td>
<td>S₆</td>
<td>S₇</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>R₃</td>
</tr>
<tr>
<td>8</td>
<td>R₂</td>
<td>R₂</td>
</tr>
</tbody>
</table>

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition
Constructing LALR parsing tables

- This method is often used in practice, because the tables obtained by it are smaller than the canonical LR tables.

- For ex, consider from above example for the states I₄ and I₇, by replacing I₄ and I₇, consisting of the set of three items represented by [C → d •, c | d | $]. The goto's on 'd' to I₄ or I₇ from I₀, I₂, I₃ and I₆ now enter I₄₇. LALR parsing table for the grammar of above example is as showed below.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
<td>$</td>
</tr>
<tr>
<td>0</td>
<td>S₃₆</td>
<td>S₄₇</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>S₃₆</td>
<td>S₄₇</td>
</tr>
<tr>
<td>3₆</td>
<td>S₃₆</td>
<td>S₄₇</td>
</tr>
<tr>
<td>4₇</td>
<td>R₃</td>
<td>R₃</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>R₁</td>
</tr>
<tr>
<td>8₉</td>
<td>R₂</td>
<td>R₂</td>
</tr>
</tbody>
</table>

Ref: Compilers - principles, techniques and tools, Aho, Lam, Sethi and Ullman, second edition