Bottom-UP Parsing

Note

The *parser* builds the *parse tree* starting from the leaf nodes labeled by the *terminals (tokens)*. It tries to discover appropriate reductions and introduces the internal nodes that are labeled by *non-terminals* corresponding to the reductions. The process finally ends at the root node labeled by the *start symbol*, or ends with an error condition.

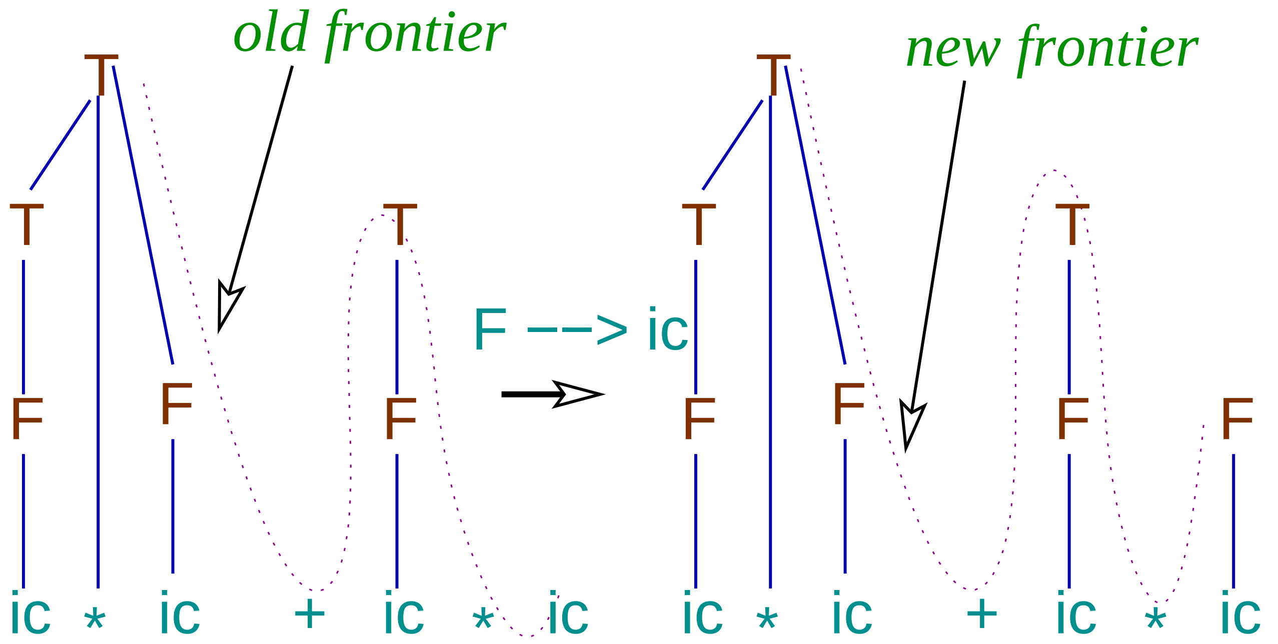
Note

At any intermediate point there is a sequence of roots of partially constructed trees (from left to right). This sequence is called an *upper frontier* of the parse tree.

Note

At every step the parser tries to find an appropriate β in the *upper frontier*, which can be reduced by a rule $A \rightarrow \beta$, so that it leads to acceptance of input.
If no such β is available, the parser either calls the *scanner* to get a new token, creates a leaf node and extend the frontier, or it reports an error.

Upper Frontier



Note

A parser reads input from left-to-right. In a bottom-up parser, the first reduction discovered is the last step of derivation at the left-most end.

Input further away from the left-end are produced by earlier steps of derivation. This indicates a natural sequence of rightmost derivations.

Note

A bottom-up parser follows the derivation sequence in reverse order as it performs reduction. So an *upper frontier* is a *prefix* of a *right sentential form*.

Handle

Let $\alpha\beta x$ be a *right sentential form*, $A \rightarrow \beta$ be a production rule, and $\alpha A x$ be the previous *right sentential form* $(x \in \Sigma^*)$. If k indicates the position of β in $\alpha\beta x$, the doublet $(A \rightarrow \beta, k)$ is called a handle of the *frontier* $\alpha\beta$ or *right sentential form* $\alpha\beta x$.

Handle

In an unambiguous grammar the rightmost derivation is unique, so a handle of a right sentential form is unique. But that need not be true for an ambiguous grammar.

Example

In our example, after the reduction of $T+F$ to $T+T$, the parser does not find any other handle in the *frontier* and invokes the scanner, that supplies the token for '$*$'. The parser forms the corresponding *leaf node* and includes it in the *frontier* $(T+T*)$. Still there is no *handle* and the scanner is invoked again to get the next token '`ic`'. The parser detects the *handle* $(F \rightarrow \texttt{ic}, T+T*\underline{\texttt{ic}})$ and reduces it to F.

Shift-Reduce Parsing

The parser essentially takes two types of actions,

- it detects a *handle* in the frontier and reduces it to get a new frontier, or
- if the handle is not present, it calls the scanner, gets a new token and extends (shifts) the frontier.

Note

The parser may fail to detect a *handle* and may report an error. But if discovered, the *handle* is always present at the right end of the *upper frontier*.

Shift-Reduce Parsing

This is called a *shift-reduce parser*. It uses a stack to hold the *upper frontier* (left end at the bottom of the stack). The *upper frontier* is a prefix of a right-sentential form at most up to the current handle. A prefix of the frontier is also called a *viable prefix* of the *right sentential form*.

Accept

If the parser can successfully reduce the whole input to the start symbol of the grammar. It reports acceptance of the input i.e. the input string is *syntactically (grammatically) correct*.

Example

Consider our old grammar:

1	`P`	$\rightarrow$	`main DL SL end`
2	`DL`	$\rightarrow$	`D DL` | `D`
4	`D`	$\rightarrow$	`T VL ;`
5	`VL`	$\rightarrow$	`id VL` | `id`
7	`T`	$\rightarrow$	`int` | `float`
9	`SL`	$\rightarrow$	`S SL` | ε

Production Rules

```
11 S    →  ES | IS | WS | IOS
15 ES   →  id := E ;
16 IS   →  if be then SL end |
             if be then SL else SL end
18 WS   →  while be do SL end
19 IOS  →  scan id ; | print e ;
```

[a]

[a] We are considering BE and E as terminals.

Input

Let the input be

```
main
    int id ;
    id := E ;
    print E ;
end$
```

The end of input is marked by *eof* ($) and the bottom-of-stack is marked also be $.

Parsing

Stack	Next Input	Handle	Action
$	main	*nil*	*shift*
$ main	int	*nil*	*shift*
$ main int	id	$(T \rightarrow$ int$)$	*reduce*
$ main T	id	*nil*	*shift*
$ main T id	;	(VL $\rightarrow$ id)	*reduce*
$ main T VL	;	*nil*	*shift*
$ main T VL ;	id	$(D \rightarrow$ T VL ;$)$	*reduce*
$ main D	id	(DL $\rightarrow$ D)	*reduce*

Note

The position of the handle is always on the top-of-stack. But the main problem is the detection of handle - when to push a token in the stack and when to reduce, and by which rule.

Automaton of Viable Prefixes

It is interesting to note that the *viable prefixes* of any CFG is a regular language. For some class of CFG it is possible to design a DFA that can be used[a] to make the *shift-reduce* decision of a parser depending on the state and fixed number of look-ahead.

[a] Along with some heuristic information.

$LR(k)$ Parsing

$LR(k)$ is an important class of CFG where a bottom-up parsing technique can be used efficiently[a].
The 'L' is for left-to-right scanning of input, and 'R' is for discovering the rightmost derivation in reverse order (reduction) by looking ahead at most k input tokens.

[a] *Operator precedence* parsing is another bottom-up technique that we shall not discuss. The time complexity of $LR(k)$ is $O(n)$ where n is the length of the input.

Note

We shall consider the cases where $k = 0$ and $k = 1$. We shall also consider two other special cases, *simple* $LR(1)$ or *SLR* and *look-ahead LR* or *LALR*. An $LR(0)$ parser does not look-ahead to decide its *shift* or *reduce* actions[a].

[a] It may look-ahead for early detection of error.

State of Viable Prefix Automaton

For every production rule $A \rightarrow \alpha$, the ordered pair $(A \rightarrow \alpha, \beta)$, where β is a *prefix* of α may be viewed as a state of the *viable prefix* automaton.
The automaton is in state $(A \rightarrow \alpha, \beta)$ after consuming β and expects to see γ so that $\alpha = \beta\gamma$.

State of Viable Prefix Automaton

The trouble is that there may be more than one production rules of the form $A \rightarrow \beta\gamma_1$ and $B \rightarrow \beta\gamma_2$. So both the pairs $(A \rightarrow \beta\gamma_1, \beta)$ and $(B \rightarrow \beta\gamma_2, \beta)$ are valid and will be in the state of the automaton[a].
A pair $(A \rightarrow \beta\gamma, \beta)$ is represented as an $LR(0)$ *item*.

---

[a] $(S \rightarrow A, \varepsilon)$, $(A \rightarrow B, \varepsilon)$, $(A \rightarrow \beta\gamma_1, \varepsilon)$, $(B \rightarrow \beta\gamma_2, \varepsilon)$.

$LR(0)$ Items

Given a CFG G, an $LR(0)$ item is a production rule $A \rightarrow \alpha$ with a dot ('$\bullet$') anywhere in α. As an example, if $\alpha = pq$, the corresponding $LR(0)$ items are $A \rightarrow \bullet pq$, $A \rightarrow p \bullet q$ and $A \rightarrow pq\bullet$. If the length of α is k, it can generate $k + 1$ $LR(0)$ items. If $A \rightarrow \varepsilon$, then the only $LR(0)$ item is $A \rightarrow \bullet$.

State

An $LR(0)$ item corresponds to a state of the *viable prefix automaton*. The item $A \to \alpha \bullet \beta$ indicates that the 'α' portion of the right-hand side '$\alpha\beta$' has already been seen by the automaton.

It is possible that there are more than one *viable prefixes* of the form $\gamma\alpha\beta$ and $\gamma\alpha\beta'$, with the *handles* $A \to \alpha\beta$ and $B \to \alpha\beta'$. So both '$\alpha \bullet \beta$' and '$\alpha \bullet \beta'$' may indicate the same state.

In general set of items corresponds to a state of the automaton.

Note

An item $A \rightarrow \alpha \bullet \beta$ indicates that the parser has already seen the string of terminals derived from α ($\alpha \rightarrow x$) and it expects to see a string of terminals derivable from β.
If $\beta = B\mu$ i.e. $A \rightarrow \alpha \bullet B\mu$; then the parser also expects to see any string generated by 'B' and all the items of the $B \rightarrow \bullet\gamma$ are to be included in the state of $A \rightarrow \alpha \bullet B\beta$[a].

[a] This is actually ε-closure of $A \rightarrow \alpha \bullet B\mu$.

Canonical $LR(0)$ Collection

The set of states of the the DFA of the viable prefix automaton is a collection of the set of $LR(0)$ items and is known as the *canonical* $LR(0)$ collection. A state of the DFA is an element of this canonical collection.

Example

Consider the following grammar:

$$
\begin{aligned}
1: &\quad P \rightarrow m\, L\, s\, e \\
2: &\quad L \rightarrow D\, L \\
3: &\quad L \rightarrow D \\
4: &\quad D \rightarrow T\, V\, ; \\
5: &\quad V \rightarrow d\, V \\
6: &\quad V \rightarrow d \\
7: &\quad T \rightarrow i \\
8: &\quad T \rightarrow f
\end{aligned}
$$

Closure()

If i is an $LR(0)$ item, then Closure(i) is defined as follows:

- $i \in$ Closure(i) - basis,
- If $A \rightarrow \alpha \bullet B\beta \in$ Closure(i) and $B \rightarrow \gamma$ is a production rule, then $B \rightarrow \bullet\gamma \in$ Closure(i).

The closure of I, a set of $LR(0)$ items, is defined as Closure(I) $= \bigcup_{i \in I}$ Closure(i).

Example

Let $i = P \rightarrow m \bullet L\ s\ e$,

$$\text{Closure}(i) = \{$$
$$P \rightarrow m \bullet L\ s\ e$$
$$L \rightarrow \bullet D\ L$$
$$L \rightarrow \bullet D$$
$$D \rightarrow \bullet T\ V\ ;$$
$$T \rightarrow \bullet i$$
$$T \rightarrow \bullet f$$
$$\}$$

Goto(I, X)

Let I be a set of $LR(0)$ items and $X \in \Sigma \cup N$. The set of $LR(0)$ items, $\text{Goto}(I, X)$ is

$$\text{Closure}\,(\{A \rightarrow \alpha\; X \bullet \beta :\; A \rightarrow \alpha \bullet X\; \beta \in I\})\,.$$

Goto() is the state transition function δ of the DFA.

Example

From our previous example
Goto(Closure($P \rightarrow m \bullet L\ s\ e$), D) is

$$\{L \rightarrow D \bullet L$$
$$L \rightarrow D\bullet$$
$$L \rightarrow \bullet DL$$
$$L \rightarrow \bullet D$$
$$D \rightarrow \bullet TV;$$
$$T \rightarrow \bullet i$$
$$T \rightarrow \bullet f\}$$

Augmented Grammar

We augment the original grammar with a new start symbol, say S', that has only one production rule $S' \rightarrow S\$$, where S is the start symbol of the original grammar. When we come to a state corresponding to $(S' \rightarrow S\$, S)$ or with the $LR(0)$ item $S' \rightarrow S \bullet \$$, we know that the parsing is OK.

$LR(0)$ **Automaton**

The alphabet of the automaton is $\Sigma \cup N$.
The start state of the automaton is $s_0 = \text{Closure}(S' \rightarrow \bullet S\$)$, the automaton expects to see the string generated by S.
All constructed states are final states[a] of the automaton as it accepts a *prefix* language.
For every $X \in \Sigma \cup N$ and for all states s already constructed, we compute $\text{Goto}(s, X)$[b] to build the automaton.

[a] The constructed automaton is incompletely specified and all unspecified transitions lead to the only non-final state.

[b] This nothing but $\delta(s, X)$.

Example: States

s_0 :	$S' \rightarrow \bullet P\$$	$P \rightarrow \bullet m\ L\ s\ e$	
s_1 :	$S' \rightarrow P \bullet \$$		
s_2 :	$P \rightarrow m \bullet L\ s\ e$	$L \rightarrow \bullet D\ L$	$L \rightarrow \bullet D$
	$D \rightarrow \bullet T\ V\ ;$	$T \rightarrow \bullet i$	$T \rightarrow \bullet f$
s_3 :	$P \rightarrow m\ L \bullet s\ e$		
s_4 :	$L \rightarrow D \bullet L$	$L \rightarrow D\bullet$	$L \rightarrow \bullet D\ L$
	$L \rightarrow \bullet D$	$D \rightarrow \bullet T\ V\ ;$	$T \rightarrow \bullet i$
	$T \rightarrow \bullet f$		

States

s_5 :	$D \to T \bullet V\ ;$ $\quad V \to \bullet d\ V$ $\quad V \to \bullet d$
s_6 :	$T \to i\bullet$
s_7 :	$T \to f\bullet$
s_8 :	$P \to m\ L\ s \bullet e$
s_9 :	$L \to D\ L\bullet$
s_{10} :	$D \to T\ V\bullet;$
s_{11} :	$V \to d \bullet V$ $\quad V \to d\bullet$ $\quad V \to \bullet d\ V$ $\quad V \to \bullet d$

States

s_{12} :	$P \rightarrow m\ L\ s\ e\bullet$
s_{13} :	$D \rightarrow T\ V\ ;\bullet$
s_{14} :	$V \rightarrow d\ V\bullet$

State Transitions

CS	*NS (Input)*											
	m	s	e	$;$	d	i	f	P	L	D	V	T
0	2							1				
2						6	7		3	4		5
3		8										
4						6	7		9	4		5
5					11						10	
8			12									
10				13								
11					11						14	

Items

- *kernel item*:

  $\{S' \rightarrow \bullet S\$\} \cup \{A \rightarrow \alpha \bullet \beta : \ \alpha \neq \varepsilon\}.$

- *nonkernel item*: $\{A \rightarrow \bullet\alpha\} \setminus \{S' \rightarrow \bullet S\$\}.$

Every *nonkernel item* in a state comes from the *closure* operation and can be generated from the *kernel items*. So it is not necessary to store them explicitly.

Note

If a state has an item of the form $A \rightarrow \alpha\bullet$, it indicates that the parser has possibly seen a *handle* and it may reduce the current right sentential form to the previous right sentential form. But there may be other complications that we shall take up.

Structure of LR Parser

Every LR-parser has a similar structure with a core parsing program, a stack to store the states of the DFA and a parsing table. The content of the table is different for different types of LR parsers.

Structure of LR Parsing Table

The parsing table has two parts, *action* and *goto*.
The $action(i, a)$ is a function of two parameters, the current state i of the DFA[a] and the current input symbol, a. The table is indexed by 'i' and 'a'. The outcome or the value, stored in the table, are of four different types.

[a] This state is stored on the top of the stack.

Action-1

Push Goto$(i, a) = j$ in the stack. This is encoded as s_j[a] - shift j.
The parser has not yet found the *handle* and augments the *upper frontier* by including the next token (in the leaf node).

[a] In fact the input token and the related attribute are also pushed in the same or a different stack (value stack) for the semantic actions. But that is not required for parsing.

Action-2

Reduce by the rule number $j : A \rightarrow \alpha$. Let the length of $\alpha = \alpha_1\alpha_2\cdots\alpha_k$ be k. The top k states on the stack $\$\cdots qq_{i_1}q_{i_2}\cdots q_{i_k}$, corresponding to this α[a], are popped out and $\mathrm{Goto}(q, A) = p$ is pushed. This is encoded as r_j - reduce by rule j.

Old stack: $\$\cdots qq_{i_1}q_{i_2}\cdots q_{i_k}$

New stack: $\$\cdots qp$

[a] $\mathrm{Goto}(q, \alpha_1) = q_{i_1}, \cdots$, $\mathrm{Goto}(q_{i_{k-1}}, \alpha_k) = q_{i_k}$.

goto **Portion**

After a reduction (*action 2*) by the rule $A \to \alpha$, the *top-of-stack* is q and we have to push $\mathrm{Goto}(q, A) = p$ on the stack. This information is stored in the *goto* portion of the table. This is the state-transition function restricted to the non-terminals.

Action-3 & 4

The parser accepts the input on some state when the only input left is the *eof ($)*. The parser rejects the input on some state where the table entry is undefined.

Configuration

The configuration of the parser is specified by the content of the stack and the remaining input. If the parser starts with the initial state of the DFA in the stack, the top-of-stack always contains the current state of the DFA. So the configuration is $(\$q_0q_{i_1}\cdots q_{i_k}, a_ja_{j+1}\cdots a_n\$)$. In terms of the sentential form it is $\alpha_1\alpha_2\cdots\alpha_k a_{j+1}\cdots a_n\$$.

Initial and Final Configurations

Initial Config.: $(\$q_0, a_1 \cdots a_j a_{j+1} \cdots a_n\$)$.
Final Config.: $(\$q_0 q_f, \$)$,
where $\text{Goto}(q_0, S) = q_f$ and the token stream is empty.

Error Configuration

Error Config.: $(\$q_0 \cdots q, a_j a_{j+1} \cdots a_n\$)$,
where Action(q, a_j) is not defined.