

Bottom-UP Parsing

The Process

- The **parse tree** is built starting from the leaf nodes labeled by the **terminals** (tokens).
- It tries to build the **subtree** of the **rightmost non-terminal** of the **right-sentential** form (with the the remaining input).
- In other words it tries to discover the **rightmost derivations** in **reverse order** and use the corresponding **reductions**.

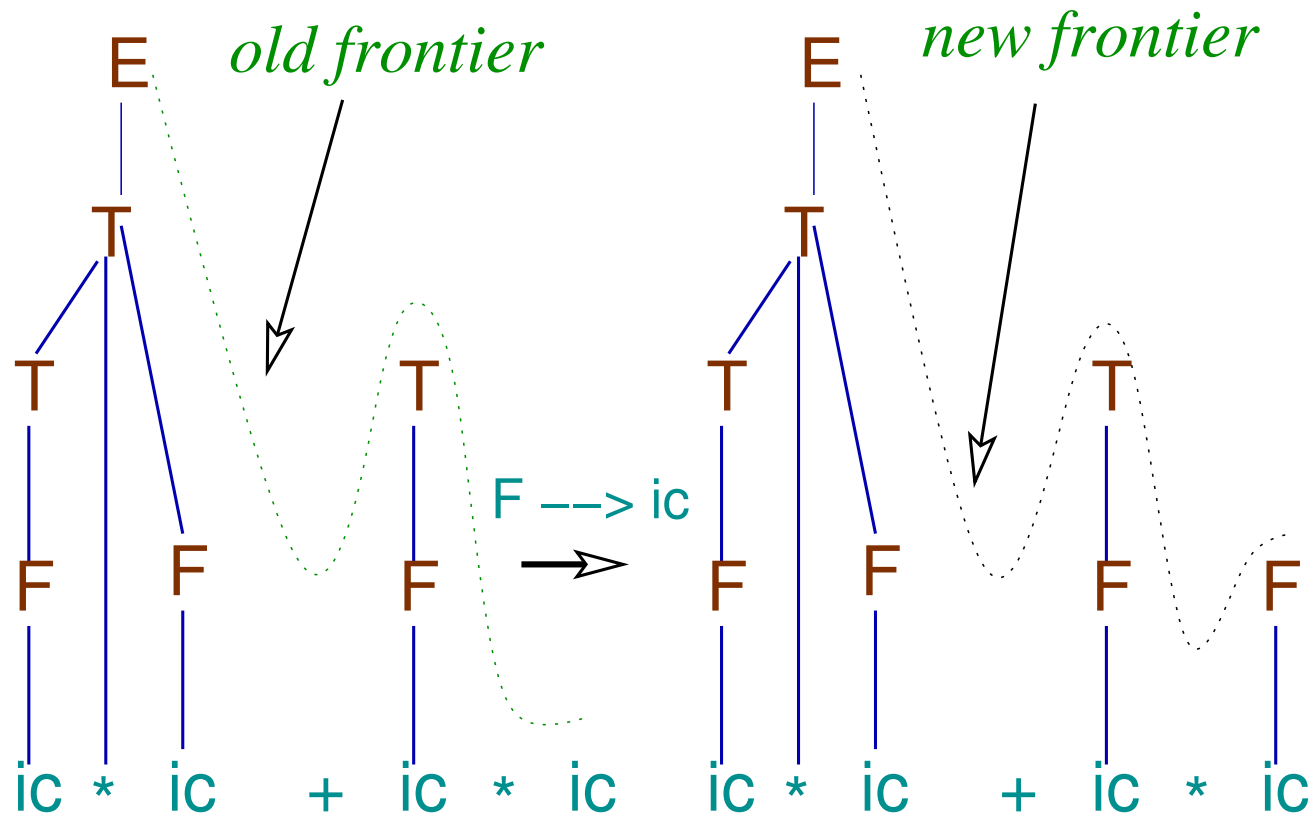
The Process

- The process ends at the **root** of the tree labeled by the **start symbol**, or with an error condition.
- At any intermediate point there is a sequence of **sub-trees** and their **roots**. This sequence may be called the **frontier** of the parse tree.

The Process

- At every step the parser tries to find an appropriate β in the **frontier**, which can be reduced by a rule $A \rightarrow \beta$ to get the **previous right-sentential** form.
- 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 reports an error.

Growing Frontier



The Process

- As a parser reads input from left-to-right, the first reduction is the last step of derivation at the left-most end.
- Input further away from the left-end were produced by earlier steps of derivation.
- The reduction takes place following the sequence of rightmost derivations in reverse order.

Rightmost Derivation of $id_1 + id_2 * id_3$

$$\begin{aligned} E &\rightarrow E + \underline{T} \\ &\rightarrow E + T * \underline{F} \\ &\rightarrow E + \underline{T} * id_3 \\ &\rightarrow E + \underline{F} * id_3 \\ &\rightarrow \underline{E} + id_2 * id_3 \\ &\rightarrow \underline{T} + id_2 * id_3 \\ &\rightarrow \underline{F} + id_2 * id_3 \\ &\rightarrow id_1 + id_2 * id_3 \end{aligned}$$

Reduction of $id_1 + id_2 * id_3$

$$\begin{aligned} & \underline{id_1} + id_2 * id_3 \\ \rightarrow & \underline{F} + id_2 * id_3 \\ \rightarrow & \underline{T} + id_2 * id_3 \\ \rightarrow & E + \underline{id_2} * id_3 \\ \rightarrow & E + \underline{F} * id_3 \\ \rightarrow & E + T * \underline{id_3} \\ \rightarrow & E + \underline{T * F} \\ \rightarrow & \underline{E + T} \\ \rightarrow & E \end{aligned}$$

Note

- A frontier is a prefix of a right sentential form.
- The parse tree is built following post-order traversal.

Frontiers of $id_1 + id_2 * id_3$

id_1

one new token

→ F

→ T

→ $E + \underline{id_2}$

two new tokens

→ $E + \underline{F}$

→ $E + T * \underline{id_3}$ two new tokens

→ $E + \underline{T * F}$

→ $E + T$

→ E

Handle

- Let $\alpha\beta x$ and αAx be the $(i+1)^{th}$ and i^{th} right sentential forms, and $A \rightarrow \beta$ be a production rule ($x \in \Sigma^*$).
- If k is the position of β in $\alpha\beta x$, the doublet $(A \rightarrow \beta, k)$ is called a handle of the frontier $\alpha\beta$ or the right sentential form $\alpha\beta x$.

Example

- In the first example $(ic * ic + ic \dots)$, after the reduction of $E + F$ to $E + T$, the parser does not find any other handle in the **frontier** and invokes the scanner. It supplies the token for $*$.
- The parser forms the corresponding **leaf node** and includes it in the **frontier** $(E + T*)$.

Example

- Still there is no handle and the scanner is invoked again to get the next token 'ic'.
- The parser detects the handle $(F \rightarrow \text{ic}, E + T * \underline{\text{ic}})$ and reduces it to F .

Handle

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

Example

Let the input be $ic_1 + ic_2 * ic_3$. The ambiguous expression grammar is $E \rightarrow E + E \mid E * E \mid ic$.

Handle	I	II	Reduction
1 st	<u>ic_1</u>	<u>ic_1</u>	$E \rightarrow ic$
2 nd	$E + \underline{ic_2}$	$E + \underline{ic_2}$	$E \rightarrow ic$
3 rd	$E + E * \underline{ic_3}$	<u>$E + E$</u>	$E \rightarrow ic,$ $E \rightarrow E + E$

Example

Let the input be $ic_1 + ic_2 * ic_3$. The ambiguous expression grammar is $E \rightarrow E + E \mid E * E \mid ic$.

Handle	I	II	Reduction
4^{th}	$E + \underline{E * E}$	$E * \underline{ic_3}$	$E \rightarrow E * E,$ $E \rightarrow ic$
5^{th}	$\underline{E + E}$	$\underline{E * E}$	$E \rightarrow E + E,$ $E \rightarrow E * E$
accept	E	E	

Shift-Reduce Parsing

A **bottom-up parser** essentially takes two types of actions,

- if it detects a **handle** in the frontier, that is **reduced** 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 frontier.

Shift-Reduce Parsing

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

^aIn the previous example of the ambiguous grammar, the right sentential form $E + E * ic$ has two handles $E + E$ or ic .

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 $P \rightarrow \text{main DL SL end}$

2 $DL \rightarrow D DL \mid D$

4 $D \rightarrow T VL ;$

5 $VL \rightarrow \text{id VL} \mid \text{id}$

7 $T \rightarrow \text{int} \mid \text{float}$

9 $SL \rightarrow S SL \mid \varepsilon$

Production Rules

11 $S \rightarrow ES \mid IS \mid WS \mid IOS$

15 $ES \rightarrow id := E ;$

16 $IS \rightarrow \text{if be then SL end} \mid$
 $\text{if be then SL else SL end}$

18 $WS \rightarrow \text{while be do SL end}$

19 $IOS \rightarrow \text{scan id ;} \mid \text{print e ;}$

^a

^aWe 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 ($)` and the `bottom-of-stack` is also marked by `$`.

Parsing

Value Stack	Next Input	Handle	Action
\$	main	nil	shift
\$ main	int	nil	shift
\$ main <u>int</u>	id	$(T \rightarrow \text{int})$	reduce
\$ main T	id	nil	shift
\$ main T <u>id</u>	;	$(VL \rightarrow \text{id})$	reduce
\$ main T VL	;	nil	shift
\$ main <u>T VL ;</u>	id	$(D \rightarrow T \text{ VL } ;)$	reduce
\$ main <u>D</u>	id	$(DL \rightarrow D)$	reduce

Note

- The position of the **handle** is always on the **top-of-stack**. The problem is its **detection**.
- When does the parser asks for a **new token** from the scanner and **push** it in the stack?
- And how does it detect the **handle** and **reduce** it.

Automaton of Viable Prefixes

- It is known that the viable prefixes of any CFG is a regular language over $\Sigma \cup N$.
- For some class of context-free grammar it is possible to design a DFA that can be used (along with some heuristic information) to take the shift-reduce decision of a parser on the basis of the DFA state and a fixed number of look-ahead of tokens.

$LR(k)$ 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.

^aOperator 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.

^aIt may look-ahead for early detection of error.

Note

- An LR parser decides about **shift** or **reduce** actions depending on the state of the automaton accepting the **viable prefixes** and examining a fixed number of current input tokens (**look-ahead**).
- The states of the deterministic automaton are subsets of **items** defined as follows.

$LR(0)$ Items

- Given a context-free grammar G , an $LR(0)$ item corresponding to a production rule $A \rightarrow \alpha$ is $A \rightarrow \beta \bullet \gamma$ where $\alpha = \beta\gamma$.
- $LR(0)$ items corresponding to the rule $E \rightarrow E + T$ are $E \rightarrow \bullet E + T, \dots, E \rightarrow E + T \bullet$.
- The $LR(0)$ item of $A \rightarrow \varepsilon$ is $A \rightarrow \bullet$.

Viabie Prefix and Valid Item

An $LR(0)$ item $A \rightarrow \alpha_1 \bullet \alpha_2$ is said to be **valid** for a **viabie prefix** $\alpha\alpha_1$ if there is a **right-most sentential form** $\alpha\alpha_1\alpha_2x$, where $x \in \Sigma^*$. It essentially means that during parsing the **viabie prefix** $\alpha\alpha_1$ may be extended to a **handle** $\alpha_1\alpha_2$,

$$S \Rightarrow_{rm}^* \alpha Ax \Rightarrow_{rm} \alpha\alpha_1\alpha_2x.$$

Note

- Given a **viable prefix** there may be more than one **valid items**.
- As an example, in the expression grammar, the **valid items** corresponding to the **viable prefix** $E + T$ are $E \rightarrow E + T \bullet$ and $T \rightarrow T \bullet * F$.

Note

- Using the first one the prefix can be extended to right sentential form as $E + T\varepsilon = E + T, E + T + ic, \dots$.
- Using the second one the prefix can be extended as $E + T * ic, \dots$.

Main Theorem

The main theorem of LR parsing claims that, the **set of valid items** of a **viable prefix** α forms the state of a deterministic finite automaton that can be **reached** from the start state by a path labeled by α .

Note

- An item $A \rightarrow \alpha \bullet \beta$ in the state of the automaton indicates that the parser has already seen the string of terminals x derived from α ($\alpha \rightarrow x$) and it expects to see a string of terminals derivable from β .
- If $\beta = B\mu$ i.e. $A \rightarrow \alpha \bullet B\mu$, where B is a non-terminal; then the parser also expects to see any string generated by ' B '.

Note

- So all the items of the form $B \rightarrow \bullet \gamma$ are included in the state of $A \rightarrow \alpha \bullet B \beta$.
- In terms of **finite automaton**, it is equivalent to **ε -transition** from the state of $A \rightarrow \alpha \bullet B \mu$. So $B \rightarrow \bullet \gamma$ is included in the DFA state of $A \rightarrow \alpha \bullet B \mu$ (ε -closure).

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.

^aIt is a set of sets.

Example

Consider the following grammar:

$$1 : P \rightarrow m L s e$$

$$2 : L \rightarrow D L$$

$$3 : L \rightarrow D$$

$$4 : D \rightarrow T V ;$$

$$5 : V \rightarrow d V$$

$$6 : V \rightarrow d$$

$$7 : T \rightarrow i$$

$$8 : T \rightarrow f$$

Closure()

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

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

If I is a set of $LR(0)$ items, the $\text{Closure}(I) = \bigcup_{i \in I} \text{Closure}(i)$.

Example

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

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

$\text{Goto}(I, X)$

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

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

$\text{Goto}()$ is the state transition function δ of the DFA.

Example

From our previous example

$\text{Goto}(\text{Closure}(P \rightarrow m \bullet L s e), D)$ 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 input string is well-formed and the parser **accepts** it.

$LR(0)$ Automaton

- The alphabet of the automaton is $\Sigma \cup N$.
- The start state is $s_0 = \text{Closure}(S' \rightarrow \bullet S\$)$, the automaton expects to see the string generated by S followed by $\$$.
- All **constructed states** are **final states**^a of the automaton as it accepts a **prefix language**.

^aThe constructed automaton is incompletely specified and all unspecified transitions lead to the only **non-final** or **dead state**.

$LR(0)$ Automaton

- For every $X \in \Sigma \cup N$ and for all states s already constructed, we compute $\text{Goto}(s, X)^a$ to build the automaton.

^aThis nothing but $\delta(s, X)$.

Example: States

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

States

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

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)										
	<i>m</i>	<i>s</i>	<i>e</i>	<i>;</i>	<i>d</i>	<i>i</i>	<i>f</i>	<i>P</i>	<i>L</i>	<i>D</i>	<i>V</i> <i>T</i>
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:

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

- Non-kernel item: $\{A \rightarrow \bullet \alpha\} \setminus \{S' \rightarrow \bullet S\$ \}.$

Every non-kernel 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.

Complete Item

- An item of the form $A \rightarrow \alpha \bullet$ is known as a **complete item**.
- If a state has a complete item $A \rightarrow \alpha \bullet$, it indicates that the parser has possibly seen a **handle** and it may be **reduced**.
- But there may be other **complications** that we shall discuss afterward.

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 of viable prefixes and a parsing table.
- The content of the table is different for different types of LR parsers^a.

^aDepends on the type of DFA and other information.

Structure of LR Parsing Table

- The parsing table has two parts, action and goto.
- The $\text{action}(i, a)$ is a function of two parameters, i is the current state of the DFA^a and ' a ' is the current token.
- The table is indexed by ' i ' and ' a '. The action stored in the table, are of four different types.

^aThe current state is available at the top of the stack.

Action-1

- $\text{action}(i, a) = s_j$, **push** the state j in the stack^a. In the automaton $\delta(i, a) = j$.
- The parser has not yet found the **handle** and augments the **frontier** by including a **new token** (forms a leaf node).

^aIn fact the input **token** and the related **attributes** are also pushed in the same or a different **stack** (**value stack**) for semantic actions. But that is not required for acceptance of input.

Action-2

- $\text{action}(i, a) = r_j$, **reduce** the handle by the rule number $j : A \rightarrow \alpha$.
- If $\alpha = \alpha_1 \alpha_2 \cdots \alpha_k$, then 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 $\delta(q, A) = \text{goto}(q, A) = p$ is pushed.
- Old stack: $\$ \cdots qq_{i_1} q_{i_2} \cdots q_{i_k} \ (q_{i_k} = i)$.
New stack: $\$ \cdots qp$

^a $\text{Action}(q, \alpha_1) = q_{i_1}, \dots, \text{Action}(q_{i_{k-1}}, \alpha_k) = q_{i_k}$.

Goto in the Table

- After a reduction (action 2) by the rule $A \rightarrow \alpha$, the top-of-stack has the state q .
- The parser driver needs to find $\delta(q, A) = \text{goto}(q, A) = p$ and **push** it 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

- An LR-parser accepts the input at the accept state when the eof (\$) is reached.
- A parser rejects the input at a state where the table entry is undefined on the current token.

Configuration

- A **configuration** of an **LR-parser** is specified by the content of the **stack** and the remaining input.
- An **LR-parser** starts with the initial state at the top of the stack and the **input**. This is the **initial configuration**:
 $(\$q_0, a_1 \cdots a_j a_{j+1} \cdots a_n \$)$.

Configuration

- At any point of computation, the top-of-stack contains the current state of the DFA. A 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 $\alpha_1\alpha_2 \cdots \alpha_k a_j \cdots a_n\$$.

Final Configurations

- A **final configuration** is $(\$q_0q_f, \$)$, where $\text{Goto}(q_0, S) = q_f$, and the token stream is **empty**.
- An **error configuration**.:
 $(\$q_0 \cdots q, a_j a_{j+1} \cdots a_n \$)$, where $\text{Action}(q, a_j)$ is not defined.