BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)		**RIGHTMOST DERIVATION**
a**b**bcde	(A → b)	**S** → aA**B**e
a**Abc**de	(A → Abc)	→ a**A**de
aA**d**e	(B → d)	→ a**A**bcde
aABe	(S → aABe)	→ abbcde
S		

The reductions trace out the right-most derivation in reverse.

<u>Handles:</u>

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

E → $\underline{\mathbf{E+E}}$
→ E+$\underline{\mathbf{E*E}}$
→ E+E*$\underline{\mathbf{id_3}}$
→ E+$\underline{\mathbf{id_2}}$*$id_3$
→ $\underline{\mathbf{id_1}}$+$id_2$*$id_3$

In the above derivation the underlined substrings are called **handles.**

<u>Handle pruning:</u>

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if w is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the n^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
\$	$id_1+id_2*id_3$ \$	shift
\$ id_1	$+id_2*id_3$ \$	reduce by E→id
\$ E	$+id_2*id_3$ \$	shift
\$ E+	id_2*id_3 \$	shift
\$ $E+id_2$	$*id_3$ \$	reduce by E→id
\$ E+E	$*id_3$ \$	shift
\$ E+E*	id3 \$	shift
\$ E+E*id3	\$	reduce by E→id
\$ E+E*E	\$	reduce by E→ E *E
\$ E+E	\$	reduce by E→ E+E
\$ E	\$	accept

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. <u>Reduce-reduce conflict:</u>

Consider the grammar:

M → R+R | R+c | R
R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

Viable prefixes:

- α is a viable prefix of the grammar if there is w such that αw is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent non-terminals.

Example:

Consider the grammar:

E → EAE | (E) | -E | id
A → + | - | * | / | ↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

E → E+E | E-E | E*E | E/E | E↑E | -E | id

Operator precedence relations:

There are three disjoint precedence relations namely

$<\cdot$ - less than
$=$ - equal to
$\cdot>$ - greater than

The relations give the following meaning:

a $<\cdot$ b – a yields precedence to b
a = b – a has the same precedence as b
a $\cdot>$ b – a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2, then make

$\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operators θ_1 and θ_2, are of equal precedence, then make

$\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if operators are left associative
$\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if right associative

3. Make the following for all operators θ:

$\theta <\cdot$ id , id $\cdot> \theta$
$\theta <\cdot$ (, ($<\cdot \theta$
) $\cdot> \theta$, $\theta \cdot>$)
$\theta \cdot>$ \$, \$ $<\cdot \theta$

Also make
(=) , (<· (,) ·>) , (<· id , id ·>) , $ <· id , id ·> $, $ <· (,) ·> $

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

<u>Operator precedence parsing algorithm:</u>

Input : An input string *w* and a table of precedence relations.
Output : If *w* is well formed, a *skeletal* parse tree **,**with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.
Method : Initially the stack contains $ and the input buffer the string *w* $. To parse, we execute the following program :

(1) Set *ip* to point to the first symbol of *w*$;
(2) **repeat forever**
(3) **if** $ is on top of the stack and *ip* points to $ **then**
(4) **return**
else begin
(5) let *a* be the topmost terminal symbol on the stack
and let *b* be the symbol pointed to by *ip;*
(6) **if** *a* <· *b* or *a* = *b* **then begin**
(7) push *b* onto the stack;
(8) advance *ip* to the next input symbol;
end;

```
(9)      else if a ·> b then                 /*reduce*/
(10)        repeat
(11)            pop the stack
(12)        until the top stack terminal is related by <·
                to the terminal most recently popped
(13)     else error( )
       end
```

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w $

where w is the input string to be parsed.

Example:

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is **id+id*id** .The implementation is as follows:

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. When '*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

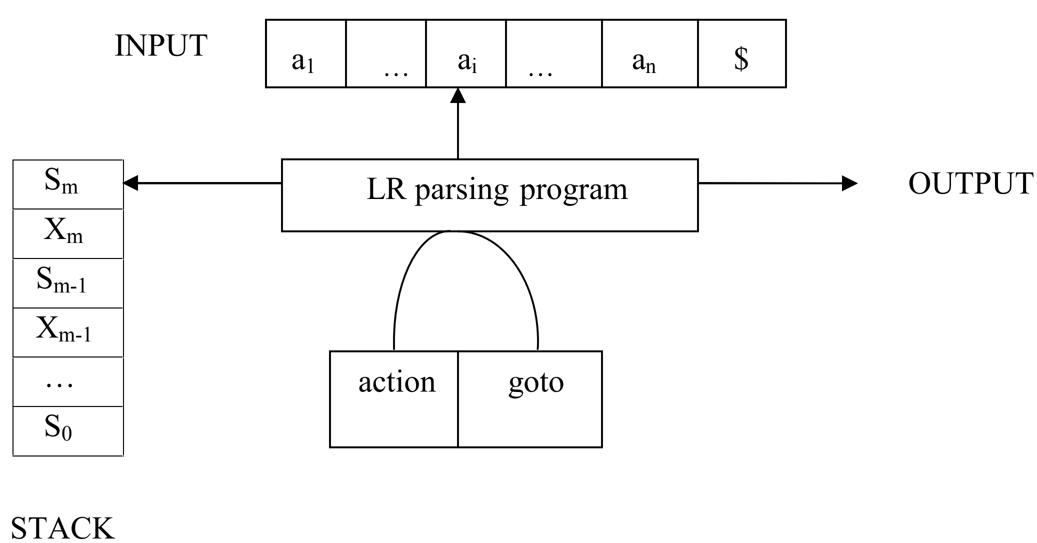
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



It consists of : 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 parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*[s_m,a_i] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever  begin
        let s be the state on top of the stack and
            a  the symbol pointed to by ip;
        if action[s, a] = shift s' then begin
            push a then s' on top of the stack;
            advance ip to the next input symbol
        end
        else if action[s, a] = reduce A→β then begin
              pop 2* | β | symbols off the stack;
              let s' be the state now on top of the stack;
              push A then goto[s', A] on top of the stack;
              output  the production A→ β
        end
        else if action[s, a] = accept then
              return
        else error( )
end
```

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → . XYZ
A → X . YZ
A → XY . Z
A → XYZ .

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'
Method :

1. Construct C = $\{I_0, I_1, \ldots I_n\}$, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [A→α·] is in I_i , then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If $goto(I_i,A) = I_j$, then $goto[i,A] = j$.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

<u>Example for SLR parsing:</u>

Construct SLR parsing for the following grammar :

G : E → E + T | T
    T → T * F | F
    F → (E) | id

The given grammar is :

G : E → E + T ------ (1)
    E →T ------ (2)
    T → T * F ------ (3)
    T → F ------ (4)
    F → (E) ------ (5)
    F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2 : Find LR (0) items.

I_0 : E' → **.** E
    E → **.** E + T
    E → **.** T
    T → **.** T * F
    T → **.** F
    F → **.** (E)
    F → **.** id

<u>GOTO (I_0 , E)</u>

I_1 : E' → E **.**
    E → E **.** + T

<u>GOTO (I_4 , id)</u>

I_5 : F → id **.**

GOTO (I_0 , T)

$I_2 : E \rightarrow T \,.$
$T \rightarrow T \,. * F$

GOTO (I_0 , F)

$I_3 : T \rightarrow F \,.$

GOTO (I_0 , ( )

$I_4 : F \rightarrow (\,. E)$
$E \rightarrow \,. E + T$
$E \rightarrow \,. T$
$T \rightarrow \,. T * F$
$T \rightarrow \,. F$
$F \rightarrow \,. (E)$
$F \rightarrow \,. id$

GOTO (I_0 , id)

$I_5 : F \rightarrow id \,.$

GOTO (I_1 , +)

$I_6 : E \rightarrow E + \,. T$
$T \rightarrow \,. T * F$
$T \rightarrow \,. F$
$F \rightarrow \,. (E)$
$F \rightarrow \,. id$

GOTO (I_2 , *)

$I_7 : T \rightarrow T * \,. F$
$F \rightarrow \,. (E)$
$F \rightarrow \,. id$

GOTO (I_4 , E)

$I_8 : F \rightarrow (E \,.)$
$E \rightarrow E \,. + T$

GOTO (I_4 , T)

$I_2 : E \rightarrow T \,.$
$T \rightarrow T \,. * F$

GOTO (I_4 , F)

$I_3 : T \rightarrow F \,.$

GOTO (I_6 , T)

$I_9 : E \rightarrow E + T \,.$
$T \rightarrow T \,. * F$

GOTO (I_6 , F)

$I_3 : T \rightarrow F \,.$

GOTO (I_6 , ( )

$I_4 : F \rightarrow (\,. E)$

GOTO (I_6 , id)

$I_5 : F \rightarrow id \,.$

GOTO (I_7 , F)

$I_{10} : T \rightarrow T * F \,.$

GOTO (I_7 , ( )

$I_4 : F \rightarrow (\,. E)$
$E \rightarrow \,. E + T$
$E \rightarrow \,. T$
$T \rightarrow \,. T * F$
$T \rightarrow \,. F$
$F \rightarrow \,. (E)$
$F \rightarrow \,. id$

GOTO (I_7 , id)

$I_5 : F \rightarrow id \,.$

GOTO (I_8 ,))

$I_{11} : F \rightarrow (E) \,.$

GOTO (I_8 , +)

$I_6 : E \rightarrow E + \,. T$
$T \rightarrow \,. T * F$
$T \rightarrow \,. F$
$F \rightarrow \,. (E)$
$F \rightarrow \,. id$

GOTO (I_9 , *)

$I_7 : T \rightarrow T * \,. F$
$F \rightarrow \,. (E)$
$F \rightarrow \,. id$

GOTO (I_4 , ( )

I_4 : F $\rightarrow$ (. E)

E $\rightarrow$. E + T

E $\rightarrow$. T

T $\rightarrow$. T * F

T $\rightarrow$. F

F $\rightarrow$. (E)

F $\rightarrow$ id

FOLLOW (E) = { $,) , +)

FOLLOW (T) = { $, + ,) , * }

FOOLOW (F) = { * , + ,) , $ }

SLR parsing table:

	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**