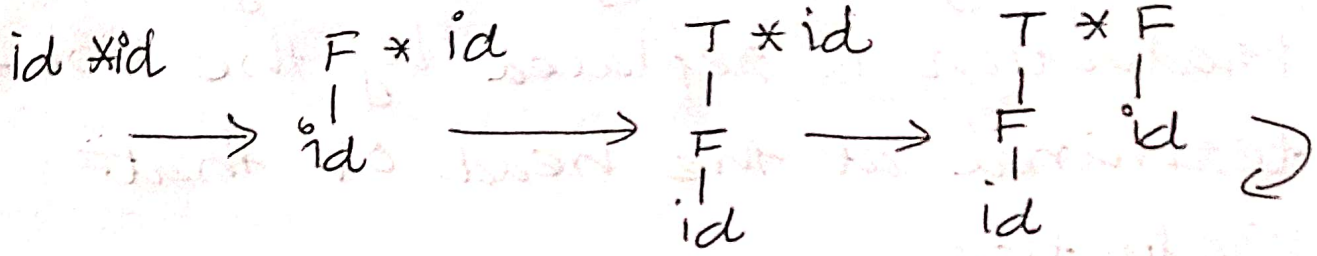
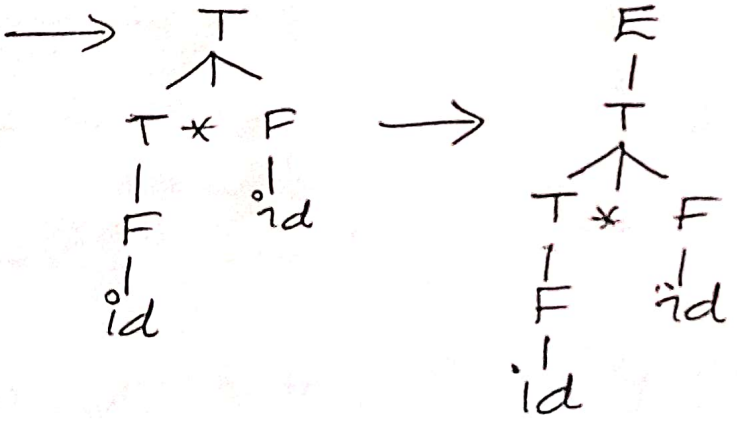
Bottom-up parsing Module 3

It is the construction of a parse tree for an input string beggining at the leaves (the bottom) and working up towards the root (the top).





For the grammar,

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

A bottom-up parse for id*id.

→ The above grammar cannot be used for top-down parsing, b'coz its left recursive.

→ Also known as Shift-Reduce parsing.

→ It allows Right Most Derivation in reverse Order.

Reductions

Bottom-up parsing 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.

Handle Pruning

Bottom-up parsing during a left to right scan of the input constructs a right most derivation in reverse.

A handle is a substring that matches the body of a production whose reduction represents one step along the reverse of a rightmost derivation.

Example

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

input string 'abbcde'

Handles during parsing of abbcde

Right sentential form	Handle	Reducing Production
a**b**bcde	b	$A \rightarrow b$
a**Abc**de	Abc	$A \rightarrow Abc$
a**A**d**e**	d	$B \rightarrow d$
aABe	aABe	$S \rightarrow aABe$

S → start symbol

Example 2

$E \rightarrow E+T \mid T$

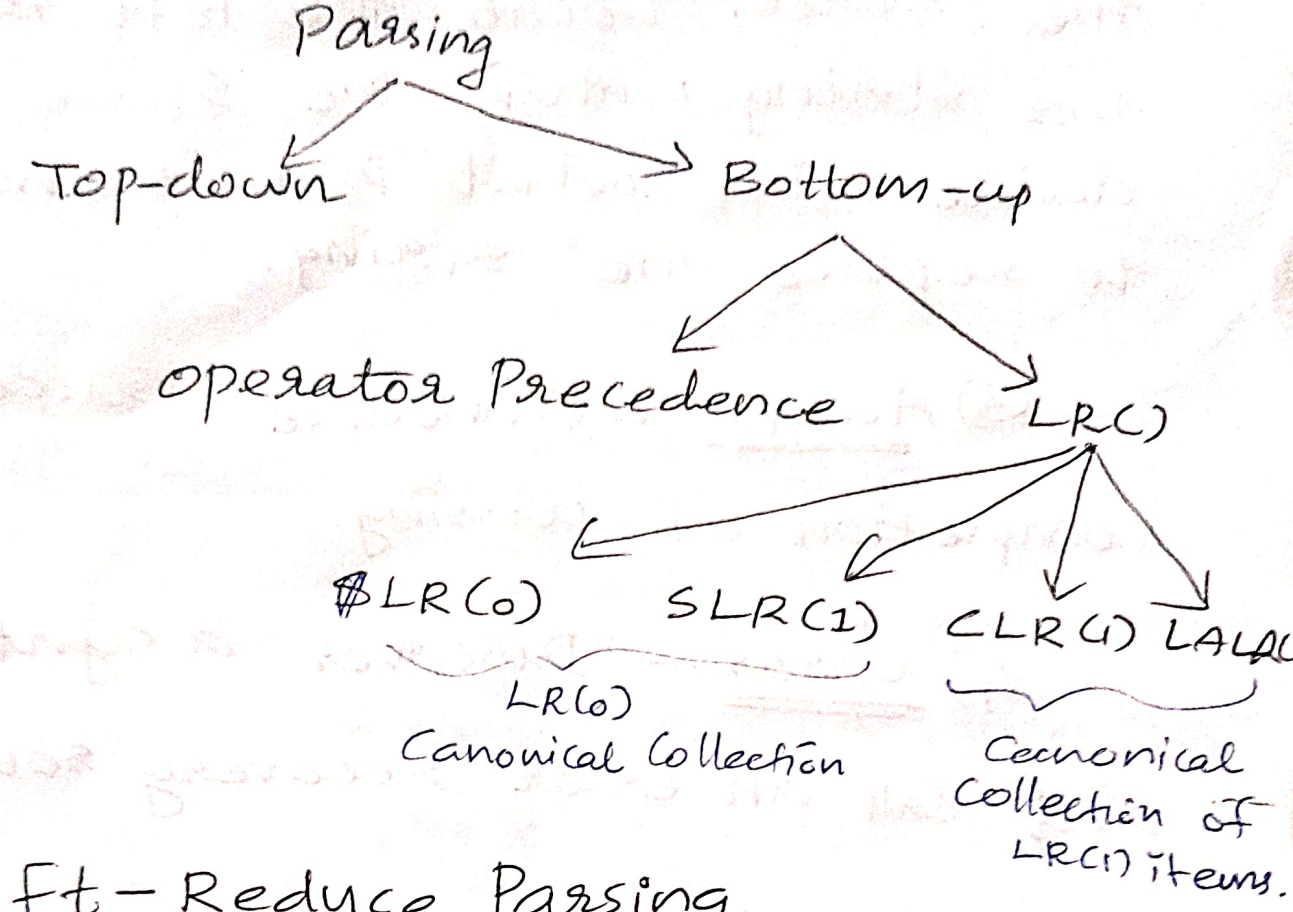
$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

input 'id * id'

Right Sentential form	Handle	Reducing Production
id * id	id	$F \rightarrow id$
F * id	F	$T \rightarrow F$
T * id	id	$F \rightarrow id$
T * F	T * F	~~$E \rightarrow E$~~ ~~$E \rightarrow T$~~ $T \rightarrow T*F$
T	T	$E \rightarrow T$
E		

→ start symbol.



Shift-Reduce Parsing.

S-R parsing is a form of bottom-up parsing in which stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

Mainly two data structures ← stack, input buffer.

~~Tuto~~ Actions of S-R parser

1) Shift – shift the next input symbol onto the top of the stack.

2) Reduce – The right end of the string to be reduced must be at

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

3) Accept – Announce successful completion of parsing.

4) Error – Discover a syntax error and call an error recovery routine.

Example 1

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

input string

" id * id "

Stack	Input buffer	Action
$	id * id $	Shift
$ id	* id $	Reduce F→id
$F	* id $	Reduce T→F
① $T	* id $	Shift
$T*	id $	Shift
$T*id	$	Reduce F→id
$T*F	$	Reduce T→T*F
② $T	$	Reduce E→T
$E	$	Accepted.

① shift-reduce conflict.

② reduce-reduce conflict.

Example 2

$S \longrightarrow (L) / a$

$L \longrightarrow L, S / S$

parse the input string $(a, (a, a))$ using shift–reduce parser.

Stack	Input buffer	Action
$	(a, (a,a))$	shift
$(	a, (a,a))$	shift
$(a	, (a,a))$	shift
$(a	, (a,a))$	Reduce S→a
$(S	, (a,a))$	Reduce S→S
$(L	, (a,a))$	shift
$(L,	(a,a))$	shift
$(L,(	a,a))$	shift
$(L,(a	,a))$	Reduce S→a
$(L,(S	,a))$	Reduce L→S
$(L,(L	,a))$	shift

$(L,(L,	a))$	shift
$(L,(L,a	))$	Reduce S→a
$(L,(L,S	))$	Reduce L→L,S
~~$(L,(L,L~~	~~))$~~	~~shift~~
~~$(L,(L,L)~~		
$(L,(L	))$	shift
$(L,(L)	)$	Reduce S→(L)
$(L,S	)$	Reduce L→L,S
$(L	)$	Shift
$(L)	$	Reduce S→(L)
$S	$	Accepted.

Operator Precedence Parsing.

→ It is applied to a small class of operator grammars.

→ A grammar is said to be Operator precedence grammar if it has 2 properties:

i) No RHS of any production has an ε.

ii) No two non-terminals are adjacent.

→ operator precedence can only established b/w the terminals of the grammar. It ignores the non-terminals.

Example:

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

is not an operator grammar, b'coz the right side EAE has two consecutive non-terminals.

If we substitute for A, each of its alternates, we obtain

$E \rightarrow E+E \mid E \rightarrow E \mid E * E \mid E \mid E \mid E \uparrow E \mid$
$(E) \mid -E \mid id$

→ There're certain disadvantages for this parsing technique, such as it is hard to handle tokens like the minus sign which has two different precedences.

→ only a small class of grammars can be parsed using this technique.

→ we use three disjoint precedence relations: $<$, $=$ and $>$, between certain pairs of terminals.

→ These precedence relations guide the selection of handles.

Rule 1:

→ If $a < b$: 'a yields precedence to b'

→ If $a = b$: 'a has same precedence as b'

→ If $a > b$: 'a takes precedence over b'.

Rule 2:

→ An identifier (id) is always given the higher precedence than any other symbol.

→ $ — always lower precedence.

Rule 3:

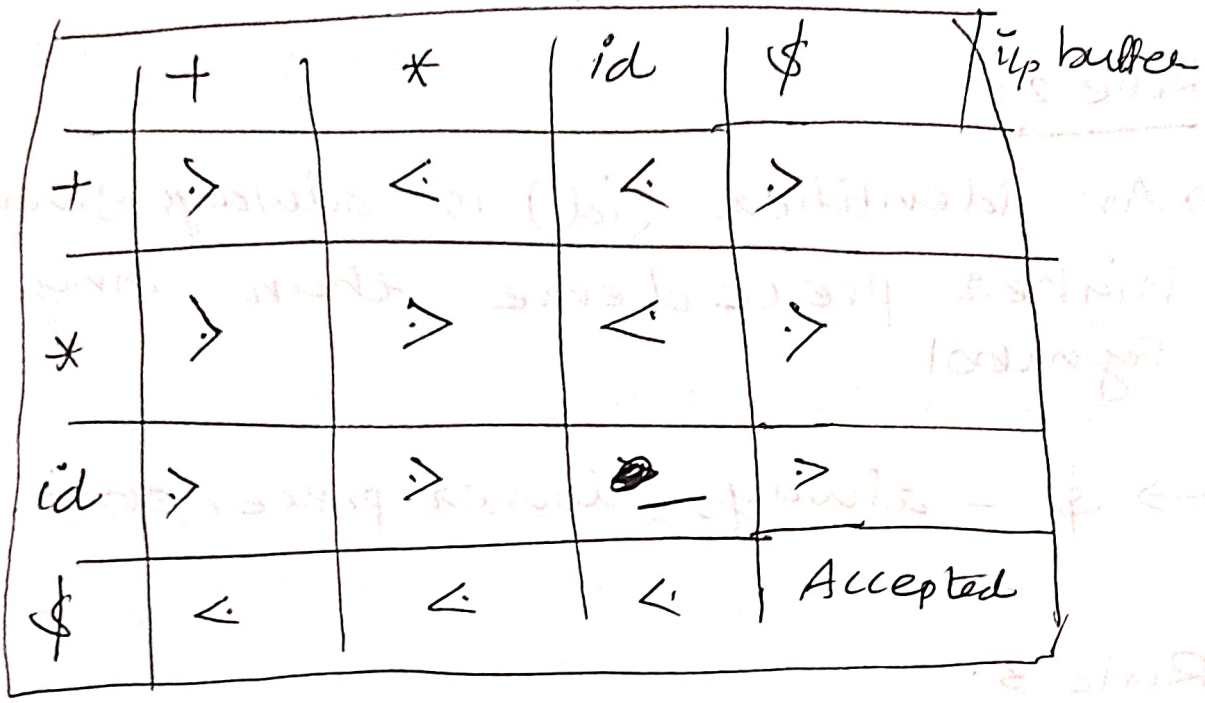
If two operators have the same precedence, check their associativity.

Left associative :— operators are evaluated from left to right.

Right associative — Operators are evaluated from right to left.

$E \rightarrow E+E \mid E*E \mid id$

operator precedence relation table:



id, a, b... — higher prec.

$ — lower precedence.

$+ > +$

associativity - from left to right

$+ < *$

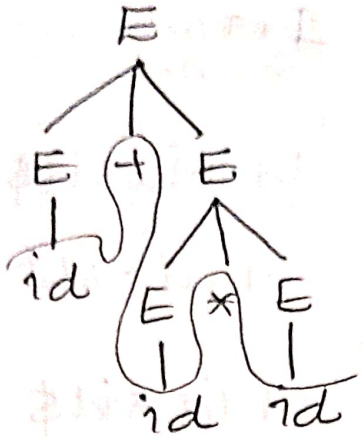
$* > *$

left associative.

Parsing the input string

Stack	Relation	Input string	operation
$	<	id+id*id$	shift id
$id	>	+id*id$	Reduce E→id
$E	<	+id*id$	shift +
$E+	<	id*id$	shift id
$E+id	>	*id$	Reduce E→id
$E+E	<	*id$	shift *
$E+E*	<	id$	shift id
$E+E*id	>	$	Reduce E→id
$E+E*E	>	$	Reduce E→E*E
$E+E	>	$	Reduce E→E+E
$E	-	$	Accepted

Parse tree



id + id * id

Precedence and Associativity

Parsers that evaluate expressions usually have to establish the order in which various operations are carried out.

Precedence —
It dictates which of two different operations is to be carried out first.

Eg:- A+B*C

Associativity –

It indicates which of two similar operations is to be carried out first.

Eg: A – B – C

can be operated as

(A – B) – C. // in C language.

A – (B – C) // in APL or FORTRAN.

Associativity in CFG

<expr> → <term> / <expr> + <term> /
<expr> – <term>

<term> → <factor> / <term> * <factor> /
<term> / <factor>

<factor> → <number> / (<expr>)

<number> → <digit> / <number> <digit>

<digit> → 0/1/2/3/4/5/6/7/8/9

All operators have same precedence here
And no associativity definitions also.

So all undefined categories are included in ambigous grammar.

(=)
(⋖ (
(⋖ id
$ ⋖ (
id ⋗)
id ⋗ $

$ ⋖ id
) ⋗ $
) ⋗)

⊖ ⋖ (
(⋖ ⊖
⊖ ⋗)
) ⋗ ⊖

P.Q → Jan 2024
Illustrate actions according to operator precedence parser for the input $id_1 + id_2 * id_3$ based on the given grammar $E \rightarrow E+E \mid E*E \mid (E) \mid id$