OPERATOR PRECEDENCE PARSING

OPERATOR GRAMMAR

- No Ɛ-transition.
- No two adjacent non-terminals.

Eg.

$E \rightarrow E \; op \; E \mid id$

$op \rightarrow + \mid *$

$E \rightarrow EF \mid id$

$F \rightarrow FG \mid Y$

$G \rightarrow HI \mid J$

The above grammar is not an operator grammar but:

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

OPERATOR PRECEDENCE

- If a has higher precedence over b; a .> b
- If a has lower precedence over b; a <. b
- If a and b have equal precedence; a =. b

Note:

- id has higher precedence than any other symbol
- $ has lowest precedence.
- if two operators have equal precedence, then we check the **Associativity** of

•But it is an important class because of its widespread applications.
•It represents a small class of grammar.

A grammar that satisfies the following 2 conditions is called as Operator Precedence Grammar–
•There exists no production rule which contains ε on its RHS.
•There exists no production rule which contains two non-terminals adjacent to each other on its RHS.

- **Designing Operator Precedence Parser-**
- In operator precedence parsing,
- Firstly, we define precedence relations between every pair of terminal symbols.
- Secondly, we construct an operator precedence table.

- **Defining Precedence Relations-**
- The precedence relations are defined using the following rules-
-

- **Rule-01:**
- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

- **Rule-02:**
- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

- **Rule-03:**
- If two operators have the same precedence, then we go by checking their associativity.

- **Parsing A Given String-**
- The given input string is parsed using the following steps-
- **Step-01:**
- Insert the following-
- $ symbol at the beginning and ending of the input string.
- Precedence operator between every two symbols of the string by referring the operator precedence table.

- **Step-02:**
- Start scanning the string from LHS in the forward direction until > symbol is encountered.
- Keep a pointer on that location.

- **Step-03:**
- Start scanning the string from RHS in the backward direction until < symbol is encountered.
- Keep a pointer on that location.

- **Step-04:**
- Everything that lies in the middle of < and > forms the handle.
- Replace the handle with the head of the respective production.

- **Step-05:**
- Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

Operator-Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

- Ex:

$E \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$
not operator grammar

$E \rightarrow EOE$
$E \rightarrow id$
$O \rightarrow + | * | /$
not operator grammar

E→E+E |
E ->E*E |
E-> E/E| id
operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a <· b	b has higher precedence than a
a =· b	b has same precedence as a
a ·> b	b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,

  <· with marking the left end,

  =· appearing in the interior of the handle, and

  ·> marking the right hand.

- In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id

The partial operator-precedence table for this grammar

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

- Then the input string id+id*id with the precedence relations inserted will be:

  $ <· id ·> + <· id ·> * <· id ·> $

To Find The Handles

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· is encountered.

$ <· id ·> + <· id ·> * <· id ·> $	E → id		$ id + id * id $
$ <· + <· id ·> * <· id ·> $		E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id		$ E + E * id $
$ <· + <· * ·> $		E → E*E	$ E + E * ·E $
$ <· + ·> $	E → E+E		$ E + E $
$ $			$ E $

Operator-Precedence Parsing Algorithm

- The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and p points to $ ) then return
   else {
      let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
      if ( a <· b or a =· b ) then {          /* SHIFT */
         push b onto the stack;
         advance p to the next input symbol;
      }
      else if ( a ·> b ) then                  /* REDUCE */
         repeat pop stack
         until ( the top of stack terminal is related by <· to the terminal most recently popped );
      else error();
   }
```

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator O_1 has higher precedence than operator O_2,
   ➔ $O_1 \gtrdot O_2$ and $O_2 \lessdot O_1$

2. If operator O_1 and operator O_2 have equal precedence,
   they are left-associative ➔ $O_1 \gtrdot O_2$ and $O_2 \gtrdot O_1$
   they are right-associative ➔ $O_1 \lessdot O_2$ and $O_2 \lessdot O_1$

3. For all operators O, O
   $\lessdot$ id, id $\gtrdot$ O, O $\lessdot$ (, ($\lessdot$ O, O $\gtrdot$),) $\gtrdot$ O, O $\gtrdot$ \$, and \$ $\lessdot$ O

4. Also, let
   ($\doteq$) \$ $\lessdot$ (id $\gtrdot$)) $\gtrdot$ \$
   ($\lessdot$ (\$ $\lessdot$ id id $\gtrdot$ \$) $\gtrdot$)
   ($\lessdot$ id

Operator-Precedence Parsing Algorithm -- Example

stack	***input***	***action***		
$	id+id*id$ $ <· id	shift		
$id	+id*id$	id ·> +	reduce	E $\rightarrow$ id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id ·> *	reduce	E $\rightarrow$ id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id ·> $	reduce	E $\rightarrow$ id
$+*	$	* ·> $	reduce	E $\rightarrow$ E*E
$+	$	+ ·> $	reduce	E $\rightarrow$ E+E
$	$	accept		

Operator-Precedence Relations

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

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - The lexical analyzer will need a lookhead to distinguish the binary minus from the unary minus.
- Then, we make

O $\lessdot$ unary-minus	for any operator
unary-minus $\gtrdot$ O	if unary-minus has higher precedence than O
unary-minus $\lessdot$ O	if unary-minus has lower (or equal) precedence than O

Operator-Precedance Grammars

Let G be an ∈-free operator grammar(No ∈-Production).For each terminal symbols *a* and *b*, the following conditions are satisfies.

1. $a \doteq b$, if ∃ a production in RHS of the form $\alpha a \beta b \gamma$, where β is either ∈ or a single non Terminal. <u>*Ex*</u> *S→iCtSeS implies* $i \doteq t$ *and* $t \doteq e$.

2. $a \lessdot b$ if for some non-terminal A ∃ a production in RHS of the form $A \rightarrow \alpha a A \beta$, and $A \Rightarrow^{+} \gamma b \delta$ where γ is either ∈ or a single non-terminal. <u>*Ex*</u> *S→iCtS and* $C \Rightarrow^{+} b$ *implies* $i \lessdot b$.

3. $a \gtrdot b$ if for some non-terminal A ∃ a production in RHS of the form $A \rightarrow \alpha A b \beta$, and $A \Rightarrow^{+} \gamma a \delta$ where δ is either ∈ or a single non-terminal. <u>*Ex*</u> *S→iCtS and* $C \Rightarrow^{+} b$ *implies* $b \gtrdot t$.

<u>Example</u>:E→E+E | E*E | (E) | id is not a Operator precedence Grammar

By Rule no. 3 we have $+ \lessdot +$ & $+ \gtrdot +$. Where as we can modify the Grammar is as follow

E→E+T | T, T→T*F | F, F→(E) | id

Operator Precedence Relations.

To find the Table we have to find the last & first terminal for each non-terminal as follows:

<u>Non terminal</u>	<u>First terminal</u>	<u>Last terminal</u>
E	*, +, (, **id**	*, +,), **id**
T	*, (, **id**	*,), **id**
F	(, **id**	), **id**

By Applying the Rule of Operator Precedence Grammar

	+	*****	**(**	**)**	**id**	**$**
+	·>	<·	<·	·>	<·	·>
*****	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Operator Precedence Relations, Continue....

To produce the Table we have to follow the procedure as:

LEADING(A) = { a| A $\Rightarrow^{+} \gamma a \delta$, where γ is ∊ or a single non-terminal.}

TRAILING(A) = { a| A $\Rightarrow^{+} \gamma a \delta$, where δ is ∊ or a single non-terminal.}

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

$$f(a) < g(b) \quad \text{whenever} \quad a <\cdot\, b$$

$$f(a) = g(b) \quad \text{whenever} \quad a =\cdot\, b$$

$$f(a) > g(b) \quad \text{whenever} \quad a \;\cdot\!> b$$

PRECEDENCE TABLE

	id	**+**	*****	**$**
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>

Example: w= $id + id * id$

$<.id.>+<.id.>*<.id.>$

BASIC PRINCIPLE

- Scan input string left to right, try to detect .> and put a pointer on its location.
- Now scan backwards till reaching <.
- String between <. And .> is our handle.
- Replace handle by the head of the respective production.
- REPEAT until reaching start symbol.

ALGORITHM

```
w ← input
a ← input symbol
b ← stack top
Repeat
{
        if(a is $ and b is $)
                return
        if(a .> b)
                push a into stack
                move input pointer
        else if(a <. b)
                c ← pop stack
                until(c .> b)
        else
                error()
}
```

EXAMPLE

STACK	INPUT	ACTION/REMARK
$	id + id * id$	$ <. Id
$ id	+ id * id$	id >. +
$	+ id * id$	$ <. +
$ +	id * id$	+ <. Id
$ + id	* id$	id .> *
$ +	* id$	+ <. *
$ + *	id$	* <. Id
$ + * id	$	id .> $
$ + *	$	* .> $
$ +	$	+ .> $
$	$	accept

PRECEDENCE FUNCTIONS

- Operator precedence parsers use **precedence functions** that map terminal symbols to integers.

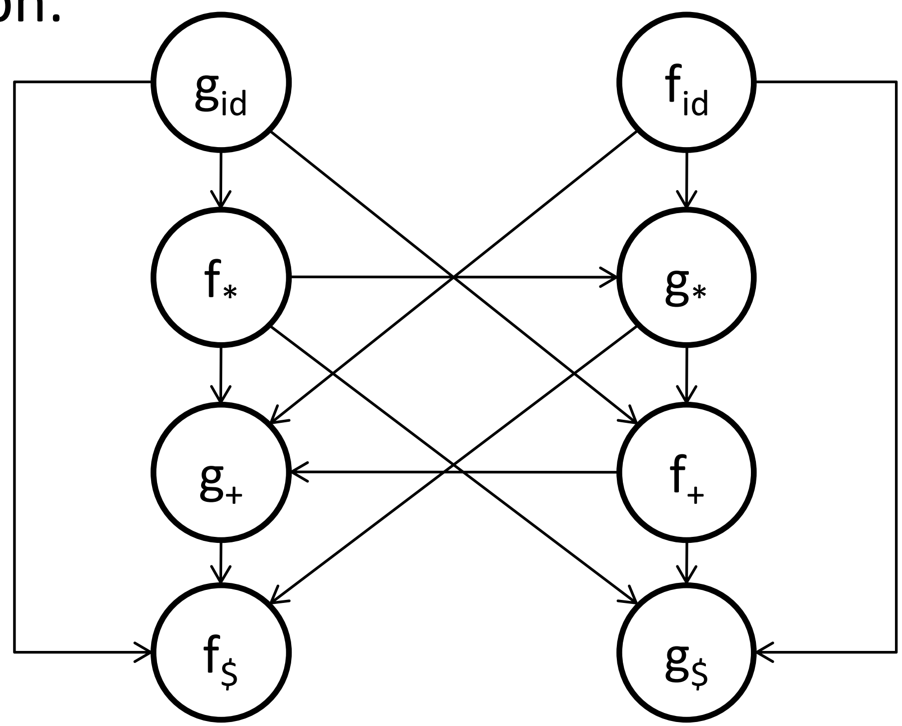
Algorithm for Constructing Precedence Functions

1. Create functions f_a for each grammar terminal *a* and for the end of string symbol.
2. Partition the symbols in groups so that f_a and g_b are in the same group if $a =\cdot b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b.
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

- Consider the following table:

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>

- Resulting graph:



- From the previous graph we extract the following precedence functions:

	id	+	*	$
f	4	2	4	0
id	5	1	3	0

Problem-01:

Consider the following grammar-

E → EAE | id

A → + | x

Construct the operator precedence parser and parse the string id + id x id.

Solution-

Step-01:

We convert the given grammar into operator precedence grammar.

The equivalent operator precedence grammar is-

E → E + E | E x E | id

Step-02:

The terminal symbols in the grammar are { id, + , x , $ }

We construct the operator precedence table as-

	id	+	x	$
id		>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	

Parsing Given String-

Given string to be parsed is **id + id x id**.

We follow the following steps to parse the given string-

Step-01:

We insert $ symbol at both ends of the string as-

$ id + id x id $

We insert precedence operators between the string symbols as-

$ < id > + < id > x < id > $

Step-02:

We scan and parse the string as-

$ **< id >** + < id > x < id > $

$ E + **< id >** x < id > $

$ E + E x **< id >** $

$ E + E x E $

$ + x $

$ < + **< x >** $

$ **< + >** $

$ $

Problem-02:

Consider the following grammar-

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L , S \mid S$$

Construct the operator precedence parser and parse the string (a , (a , a)).

Solution-

The terminal symbols in the grammar are { (,) , a , , }

We construct the operator precedence table as-

	a	(	)	,	\$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
\$	<	<	<	<	

Parsing Given String-

Given string to be parsed is **(a , (a , a))**.

We follow the following steps to parse the given string-

Step-01:

We insert \$ symbol at both ends of the string as-

$$\mathbf{\$ (a , (a , a)) \$}$$

We insert precedence operators between the string symbols as-

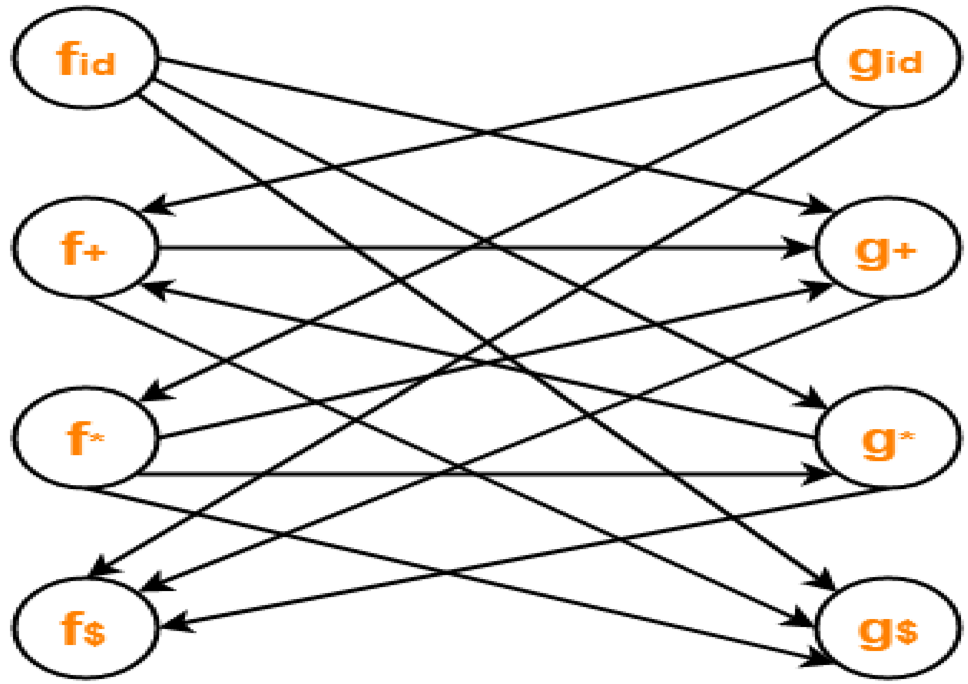
$$\mathbf{\$ < (< a > , < (< a > , < a >) >) > \$}$$

Step-02:

We scan and parse the string as-

$$\$ < (\underline{< \mathbf{a} >} , < (< a > , < a >) >) > \$$$
$$\$ < (S , < (\underline{< \mathbf{a} >} , < a >) >) > \$$$
$$\$ < (S , < (S , \underline{< \mathbf{a} >}) >) > \$$$
$$\$ < (S , \underline{< \mathbf{(S , S)} >}) > \$$$
$$\$ < (S , \underline{< \mathbf{(L , S)} >}) > \$$$
$$\$ < (S , \underline{< \mathbf{(L)} >}) > \$$$
$$\$ \underline{< \mathbf{(S , S)} >} \$$$
$$\$ \underline{< \mathbf{(L , S)} >} \$$$
$$\$ \underline{< \mathbf{(L)} >} \$$$
$$\$ \underline{< \mathbf{S} >} \$$$
$$\$ \$$$

Problem-03:



Consider the following grammar-

$E \rightarrow E + E \mid E \times E \mid id$

1.Construct Operator Precedence Parser.

2.Find the Operator Precedence Functions.

The terminal symbols in the grammar are

{ + , x , id , $ }

We construct the operator precedence table as-

Operator Precedence Table

		g →			
		id	+	x	$
	id		>	>	>
f ↓	+	<	>	<	>
	x	<	>	>	>
	$	<	<	<	

Here, the longest paths are-

- $f_{id} \rightarrow g_x \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$
- $g_{id} \rightarrow f_x \rightarrow g_x \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

The resulting precedence functions are-

	+	x	id	$
f	2	4	4	0
g	1	3	5	0

Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.