OPERATOR PRECEDENCE PARSING

Operator-Precedence Parsing (OPP)

The operator-precedence parser is a shift –reduce parser that can be easily constructed by hand. Operator precedence parser can be constructed from a small class of grammars which is called operator grammar. These grammars have the property (among other essential requirements)

- That no production right side is ε
- And no production right side has two adjacent nonterminal.

Example: The following grammar for expressions

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

Is not an **operator grammar**, because the right side **EAE** has two (in fact three) consecutive nonterminals. However, if we substitute for **A** each of its alternatives, we obtain the following operator grammar:

E ———→E+E / E-E / E*E / E÷E / E↑E / (E) / -E / id

We now describe an easy-to-implement parsing technique called operator-precedence parsing.

operator-precedence relation:

In operator-precedence parsing, there are three disjoint **precedence relations** namely:

<• - less than   **=•** - equal to   **•>** - greater than

The relations give the following meanings:

RELATION	MEANING
a* <• *b	***a*** *"yields precedence to"* ***b***
a* = *b	***a*** *"has the same precedence as"* ***b***
a* •> *b	***a*** *"takes precedence over"* ***b***

How to Create Operator-Precedence Relations:

- We use associativity and precedence relations among operators.

1. If operator θ1 has higher precedence than operator θ2, then make θ1. > θ2 and θ2 < . θ1

2.If operators θ1 and θ2, are of equal precedence, then make θ1. > θ2 and θ2. > θ1 if operators are left associative θ1 < . θ2 and θ2 < . θ1 if right associative

3. Make the following for all operators θ:

θ < . id , id . > θ

θ < . (, (< . θ

) . > θ , θ . >)

θ . > $, $ < . θ

4.Also make

(=) , (< . (,) . >) , (< . id , id . >) , $ < . id , id . > $, $ < . (,) . > $

These rules ensure that both id and (E) will be reduced to E. Also, $ serves as both the left and right endmarker, causing handles to be found between $'s wherever possible

Note:

- Id has higher precedence than any other symbol
- $ has lowest precedence.
- if two operators have equal precedence, then we check the Associativity of that particular operator.

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 X in the table denote error entries

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

Operator-precedence parsing algorithm:

Input: an input string w & table of precedence relations (holds precedence relations between certain terminals).

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:

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();
   }
```

Stack implementation of operator precedence parser:

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 (like shift-reduce technique but in the other manner).

The initial configuration of an operator precedence parsing is

Stack	Input
$	W$

Where W is the input string to be parsed the precedence and associativity of the rule on the top of stack, and the current token are used to determine whether to shift or reduce. this is done as follow:

When the relation between the top of stack and the leftmost of input word is .> this is mean perform reduce action, otherwise (when the relation < or =) the action is Shift .example 1 explain how use the Operator precedence for parse the an expression

Ex: - 1
Use Stack implementation of operator precedence parser to check this sentence id + id by this grammar: E→ E+E | E*E | id

Sol:

Stack		**Input**
$	<	id + id$
$ <id	>	+ id$
$<id>		+ id$
$<E+		+ id$
$<E+	<	id$
$<E+<id	>	$
$<E+<id>		$
$<E+E>		$
$ E		$
accept		

Ex2: Consider the following grammar

E → EOE | -E| (E)| id

O→ - | +| *|/| ↑

Using Operator precedence for parse the expression id1*(id2+id3) ↑ id

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

Stack		input	Action
$	<	id1*(id2+id3) ↑ id $	shift
$ id1	>	*(id2+id3) ↑ id $	Reduce E→id
$E	>	*(id2+id3) ↑ id $	shift
$E*	>	(id2+id3) ↑ id $	shift
$ E*(	<	id2+id3) ↑ id $	shift
$ E*(id2	>	+id3) ↑ id $	Reduce E→id
$ E*(E	<	+id3) ↑ id $	shift
$ E*(E+	<	id3) ↑ id $	shift
$ E*(E+ id3	>	) ↑ id $	Reduce E→id
$ E*(E+ E	>	) ↑ id $	Reduce E→E+E
$ E*(E	=	) ↑ id $	Shift
$E*(E)	>	↑ id$	Reduce E→(E)
$E * E	<	↑ id$	shift
$E*E↑	<	id$	shift
$E*E↑ id	>	$	Reduce E → id
$E*E↑ E	>	$	Reduce E → E ↑ E
$ E * E	>	$	Reduce E→E*E
$ E		$	Accept

H.W
Try input id*(id ↑ id)-id/id) using the same grammar in EX2