Computer Science and Information System Department

Compiler Design

3rd Class

2019-2020

Operator precedence parser –

An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in any parser except operator precedence parser.

There are two methods for determining what precedence relations should hold between a pair of terminals: Use the conventional associativity and precedence of operator.

The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: **⋖, ≐, ⋗**

a ⋖ b This means a “yields precedence to” b.

a ⋗ b This means a “takes precedence over” b.

a ≐ b This means a “has same precedence as” b.

Example:

It is observed that this parser depends upon a parsing schedule that it must recommend while analyzing a given input string.
For example:
E->E+T | T
T->T*F | F
F->id
Firstly find the first terminal and last terminal of each non terminal

First terminal and last terminal of each non-terminal

Non-Terminal	First Terminal	Last Terminal
E	+,*,id	+,*,id
T	*,id	*,id
F	Id	id

Now find the precedence relation between terminals:

Same Precedence: There is no same precedence between any terminals in this grammar.

Yields precedence:

E->E+T then

+ <◦ *

+ <◦ id

T->T*F then

*<◦ id

$ <◦ +

$ <◦ *

$ <◦ id

Takes precedence:

E-> E+T then

+ ◦> +

*◦> +

id ◦> +

T->T*F then

*◦> *

id ◦> *

+ ◦> $

*◦> $

id ◦> $

For this grammar the precedence relation table is drawn below:

Precedence Relation

	id	+	*	$
id	-	∘>	∘>	∘>
+	<∘	∘>	<∘	∘>
*	<∘	∘>	∘>	∘>
$	<∘	<∘	<∘	-

Operator precedence parser utilizes precedence functions that represent terminal characters to whole numbers and so the precedence relations between the terminal characters are employed over integral analysis. So for this we design an algorithm by which we can derive precedence functions between terminals.

Algorithm:

i. Create functions fx for each grammar terminal x and for the end of string symbol ($).

ii. Partition the symbols in groups so that fx and gy are in the same group if x =◦ y. (There can be symbols in the same group even if they are not connected by this relation).

iii. Create an inclined sketch whose nodes are in the groups, next for each symbols x and y do:

place an edge from the group of gy to the group of fx if x<◦ y otherwise if x ◦> y place an edge from the group of fx to that of gy.

iv. If the constructed sketch has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fx and gy respectively.

So by this algorithm construct an inclined sketch by which the table II is encoded by two precedence functions f and g that represent terminal characters to whole numbers. Two functions f

and g where the following apply:

if x<◦ y then f(x) < g(y)

if x◦> y then f(x) > g(y)

if x=◦y then f(x)= g(y)

To frame these functions construct an inclined sketch with vertices fi and gi where i is the ith terminal adapting the given principles:

If x=◦ y then fxand fy are grouped together and gx and gy are grouped together.
If x◦> y then an inclined edge is drawn from fx to gy.
If x<◦ y then an inclined edge is drawn to fx from gy.
Applying these principles the precedence sketch for the above precedence table looks something like this:

Now it is easy to make a table for the f and g values of each terminal character by signifying f(x) as the long-drawn-out probable route in the sketch starting from fx such that individual vertex on this route has lower precedence in compare to its nearby preceding vertex, similarly for g(x) as well. The resulting precedence function or long-drawn-out route table driven from above sketch is:

Precedence Functions

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