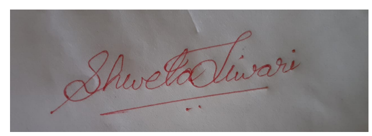
CD: COMPILER DESIGN
CD: UNIT-2 09/2022

SEPTEMBER 2022 / IT-3rd Year, V Semester
FALL SEMESTER, YEAR (5th, 3rd)
FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: SEPTEMBER, 2022

PREPARED FOR
Engineering Students
All Engineering College

CD: COMPILER DESIGN

INSTRUCTOR: Ms. SHWETA TIWARI
shwetatiwari08@ recabn.ac.in
shwetatiwari08aug@ gmail.com



September 2022

TOPIC On : UNIT-2 OPERATOR PRECEDENCE PARSER

By SHWETA TIWARI

Under On: Basic Parsing Techniques

TOPIC On : UNIT-2 OPERATOR PRECEDENCE PARSER

PRECEDENCE PARSER

Operator Precedence Parsing:

A grammar that is used to define mathematical operators is called an **operator grammar** or **operator precedence grammar**.

TWO RESTRICTION FOR GRAMMAR

1. Such grammars have the restriction that no production has either an **ε (empty)** right-hand side (null productions)
2. **No two adjacent non-terminals** in its right-hand side of production.

Examples –

This is an example of operator grammar:

E->E+E/E*E/id

However, the grammar given below is not an operator grammar because two non-terminals are adjacent to each other:

S->SAS/a

A->bSb/b

We can convert it into an operator grammar, though:

S->SbSbS/SbS/a

A->bSb/b

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:

1. Use the conventional associativity and precedence of operators.
2. 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.

Operator grammars have the property that no production right side is ε (empty) or has two adjacent non terminals. This property enables the implementation of efficient operator- precedence parsers.

Example: The following grammar for expressions:

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

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

This is not operator grammar, because the right side EAE has two consecutive non-terminals. However, if we substitute for A each of its alternate, we obtain the following operator grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid E \wedge E \mid - E \mid id$$

In operator-precedence parsing, we define three disjoint precedence relations between pairs of terminals. This parser relies on the following three precedence relations.

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*

Fig 1 Precedence Relations

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

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

Fig 2 Operator Precedence Relation Table

Example: The input string: $id_1 + id_2 * id_3$

After inserting precedence relations the string becomes:

$ <· id_1 ·> + <· id_2 ·> * <· id_3 ·> $

Having precedence relations allows identifying handles as follows:

1. Scan the string from left end until the leftmost ·> is encountered.
2. Then scan backwards over any ='s until a <· is encountered.
3. Everything between the two relations <· and ·> forms the handle.

Stack	Rule	Input	Comments
$ < id > + < id > * < id > $	E → id	$ id + id * id $	Here the first "id" is looked as the handle and since we were able to reduce, we reduce it in the input
$ < + < id > * < id > $	E → id	$ E + id * id $	The second handle is also "id" since that is available between a pair of lesser than and greater than precedences
$ < + < * < id > $	E → id	$ E + E * id $	The third handle is also "id".
$ < + < * > $	E → E*E	$ E + E * E $	The fourth handle is E *E, and is popped in the stack and we push the greater than symbol.
$ < + > $	E → E+E	$ E + E $	The last handle is E+E and that is also reduced.
$$			The stack is empty and has only the $ symbol, we say the string is accepted.

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.

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

Fig 3 Operator Precedence Relation Table

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

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

Fig 4 Stack Implementation

Implementation of Operator-Precedence Parser:

- An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.
- More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

Steps involved in Parsing:

1. Ensure the grammar satisfies the pre-requisite.
2. Computation of the function LEADING()
3. Computation of the function TRAILING()
4. Using the computed leading and trailing ,construct the operator Precedence Table
5. Parse the given input string based on the algorithm
6. Compute Precedence Function and graph.

Computation of LEADING:

- Leading is defined for every non-terminal.
- Terminals that can be the first terminal in a string derived from that non-terminal.
- LEADING(A)={ a| $A \Rightarrow^{+} \gamma a \delta$ },where γ is ε or any non-terminal, $\Rightarrow^{+}$ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for LEADING(A):

{

1. 'a' is in LEADING(A) is $A \rightarrow \gamma a \delta$ where γ is ε or any

non-terminal. 2.If 'a' is in LEADING(B) and $A \rightarrow B$, then 'a' is in

LEADING(A).

}

Computation of TRAILING:

- Trailing is defined for every non-terminal.
- Terminals that can be the last terminal in a string derived from that non-terminal.
- TRAILING(A)={ a| $A \Rightarrow^{+} \gamma a \delta$ },where δ is ε or any non-terminal, $\Rightarrow^{+}$ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for TRAILING(A):

{

1. 'a' is in TRAILING(A) is A→ γaδ where δ is ε or any

non-terminal. 2.If 'a' is in TRAILING(B) and A→B, then 'a' is in

TRAILING(A).

}

Example 1: Consider the unambiguous

grammar, E→E + T

E→T T→T

* F T→F

F→(E)

F→**id**

Step 1: Compute LEADING and TRAILING: LEADING(E)=

{ +,LEADING(T)} ={+ , * , (, id}

LEADING(T)= { *,LEADING(F)} ={* , (, id}

LEADING(F)= { (, id}

TRAILING(E)= { +, TRAILING(T)} ={+ , * ,) , id}

TRAILING(T)= { *, TRAILING(F)} ={* ,) , id}

TRAILING(F)= {) , id}

Step 2: After computing LEADING and TRAILING, the table is constructed between all the terminals in the grammar including the '$' symbol.

for each production $A \rightarrow X_1 X_2 X_3 \ldots X_n$

for i=1 to n-1

1. if X_i and X_{i+1} are terminals

   set $X_i \doteq X_{i+1}$

2. if $i \leq n-2$ and X_i and X_{i+2} are terminals and X_{i+1} is a non-terminal,

   set $X_i \doteq X_{i+2}$

3. if X_i is a terminal and X_{i+1} is a non-terminal ,then for all 'a' in LEADING(X_{i+1})

   set $X_i \lessdot$ a

4. if X_i is a non-terminal and X_{i+1} is a terminal ,then for all 'a' in TRAILING(X_i)

   set a $\gtrdot X_{i+1}$

5. Set \$ $\lessdot$ Leading(S) and Trailing(S) $\gtrdot$ \$,where S-start symbol.

Fig 5 Algorithm for constructing Precedence Relation Table

	+	*	**id**	**(**	**)**	**$**
+	>	<	<	<	>	>
*	>	>	<	<	>	>
id	>	>	e	e	>	>
(	<	<	<	<	=	e
)	>	>	e	e	>	>
$	<	<	<	<	e	**Accept**

Fig 6 Precedence Relation Table

Step 3: Parse the given input string **(id+id)*id$**

```
Set ip to point to the first symbol of w$
Repeat forever
    if $ is on the top of the stack and ip points to $ then return
    else begin
        Let a be the top terminal on the stack, and b the symbol pointed to by ip
                if a <· b or a =· b then
                        push b onto the stack
                        advance ip to the next input symbol
                end
                else if a ·> b then
                        repeat
                                pop the stack
                        until the top stack terminal is related by <·
                                to the terminal most recently popped
                else error()
    end
```

Fig 7 Parsing Algorithm

STACK	REL.	INPUT	ACTION
$	$ < (	(id+id)*id$	Shift (
$(	(< id	id+id)*id$	Shift id
$(id	id > +	+id)*id$	Pop id
$(	(< +	+id)*id$	Shift +
$(+	+ < id	id)*id$	Shift id
$(+id	id >)	)*id$	Pop id
$(+	+ >)	)*id$	Pop +
$(	(=)	)*id$	Shift)
$() $(	) > *	*id $	Pop) Pop (
$	$ < *	*id $	Shift *
$*	* < id	id$	Shift id
$*id	id > $	$	Pop id
$*	* > $	$	Pop *
$		$	Accept

Fig 8 Parse the input string (id+id)*id$

Precedence Functions:

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b.

1. $f(a) < g(b)$ whenever $a <\cdot b$.
2. $f(a) = g(b)$ whenever $a = b$. and
3. $f(a) > g(b)$ whenever $a \cdot> b$.

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 $a = b$ (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.

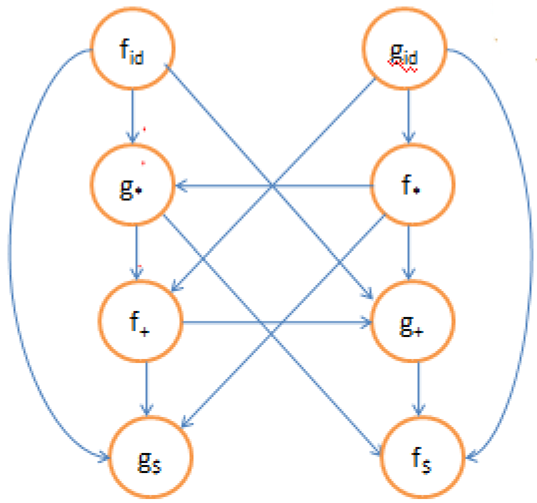


Fig 9 Precedence Graph

There are no cycles,so a precedence function exists. As f$ and g$ have no out edges,f($)=g($)=0.The longest path from g+ has length 1, so, g (+)=1.There is a path from g_{id} to f* to g* to f+ to g+ to f$, so, g (id)=5.The resulting precedence functions are:

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

Fig 10 Precedence Table

Example 2:

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i) *id=id (ii)id*id=id are successfully parsed or not?

S→L=R

S→R

L→*R

L→id

R→L

Solution:

1. **Computation of LEADING:**

LEADING(S) = {=, * , id}

LEADING(L) = {* , id}

LEADING(R) = {* , id}

2. **Computation of TRAILING:**

TRAILING(S) = {= , * , id}

TRAILING(L)= {* , id}

TRAILING(R)= {* , id}

3. **Precedence Table:**

	=	*	id	$
=	e	<·	<·	·>
*	·>	<·	<·	·>
id	·>	e	e	·>
$	<·	<·	<·	accept

* All undefined entries are error (e).

4. **Parsing the given input string:**

1. ***id = id**

STACK	INPUT STRING	ACTION
$	*id=id$	$<·* Push
$*	id=id$	*<·id Push
$*id	=id$	id·>= Pop
$*	=id$	*·>= Pop
$	=id$	$<·= Push
$=	id$	=<·id Push
$=id	$	id·>$ Pop
$=	$	=·>$ Pop
$	$	Accept

2. **id*id=id**

STACK	INPUT STRING	ACTION
$	id*id=id$	$<·idPush
$id	*id=id$	Error

Example 3: Check whether the following Grammar is an operator precedence grammar or not.

E→E+E

E→E*E

E→id

Solution:

1.Computation of LEADING:

LEADING(E) = {+, * , id}

2.Computation of TRAILING:

TRAILING(E) = {+, * , id}

3.Precedence Table:

	+	*****	**id**	**$**
+	<·/·>	<·/·>	<·	·>
*****	<·/·>	<·/·>	<·	·>
id	·>	·>	----------	·>
$	<·	<·	<·	---------- accept

All undefined entries are errors. Since the precedence table has multiple defined entries, the grammar is not an operator precedence grammar.

********************************THE END************************