MODULE 12 – LEADING, TRAILING and Operator Precedence Table

In this module, we will learn to construct the next type of Bottom up parser, which is the operator precedence parser. The functions Leading and Trailing are computed to construct the Operator precedence parsing table. This module discusses the construction of Operator precedence parsing table using the functions leading and trailing.

12.1 Operator Precedence Grammar

The class of grammar parsed by the Operator precedence parser is the Operator Precedence Grammar. This is a small, but an important class of grammar. An operator precedence parser can be a powerful shift-reduce parser for this small class of operator precedence grammar. A grammar has to satisfy the following conditions for it to be parsed by an operator precedence parser:

- No ε-productions are permitted thus ensuring that the RHS of all production should be a combination of terminals and non-terminals.
- The RHS of all productions should be in such a way that no two non-terminals should be adjacent

Consider the following examples:

1. Grammar G1 is defined with the productions, E→AB, A → a, B →b. This grammar has the non-terminals A and B adjacent to each other and hence is not an operator precedence grammar.
2. Grammar G2 is defined with productions, E→EOE, E → id, O → + | * | / . This grammar is also not operator precedence grammar as E, O, E are non-terminals and they are adjacent to each other.
3. Grammar G3, is a modified version of G2 with productions, E→E+E | E*E | E/E | id which is a operator precedence grammar as it does not have ε productions and has no two non-terminals adjacent to each other in the RHS of the productions.

12.2 Operator precedence parser rules

Let G be an ∈-free operator grammar (No ∈-Production).For each terminal symbols a and b, the following conditions need to be satisfied. We define three symbols to define the precedence relations between two terminals ‘a’ and ‘b’ namely, the ≐ , <·, ·> to indicate same, lesser and greater precedence respectively. The same, lesser and greater precedence between any two terminals is defined based on the following rules:

1. The first rule is for the same precedence relation between two terminals ‘a’ and ‘b’. We say, a≐ b, if ∃ a production in RHS of the form αaβbγ, where β is either ∈ or a single non-Terminal. Consider the grammar with a production, S→iCtSeS. It can be observed

that this grammar is an operator precedence grammar. The symbols for α can be ε, C, S for three situations. Thus for the scenario where α is ε, we have 'a' is 'i' and 'b' is 't'. Thus we have the relation, i ≐ t. Considering an alternate situation, where α is 'C', we have 'a' as 't' and 'b' as 'e' which yields the precedence relation t ≐ e. When α is S, we don't have a symbol for β and thus only these two precedence relations could be derived.

2. The second rule is to design the lesser than relation between two terminals 'a' and 'b'. We say, a $<\cdot$ b if for some non-terminal A ∃ a production in RHS of the form A→ αaAβ, and A$\Rightarrow^{+}$ γbδ where γ is either ∈ or a single non-terminal. Consider the productions, S→iCtS and C $\Rightarrow^{+}$ b and where β has 'bδ' and hence, we deduce i $<\cdot$ b as non-terminal C derives 'b'

3. The third rule is to design the greater than relation between two terminals 'a' and 'b' and is to look at the derivation from the right of the RHS of the production. We say, a $\cdot>$ b if for some non-terminal A ∃ a production in RHS of the form A→ αAbβ, and A$\Rightarrow^{+}$ γaδ where δ is either ∈ or a single non-terminal. Thus, for the production, S→iCtS and C $\Rightarrow^{+}$ b the relation derived will be b $\cdot>$ t.

Example 12.1 Consider the following ambiguous expression grammar.

E→E+E | E*E | (E) | id

This grammar is not a Operator precedence Grammar since by rule no. 3 we have + $<\cdot$ + & + $\cdot>$ + as the grammar is ambiguous. However, we have the unambiguous expression grammar and is given below. This grammar has clear definition of the precedence relation. E→E+T | T, T→T*F | F, F→(E) | id

12.3 Operator precedence parsing

The operator precedence parser is based on the precedence rules that are defined between the terminals of a grammar. In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

- a $<\cdot$ b implies 'b' has higher precedence than 'a'
- a $=\cdot$ b implies 'b' has same precedence as 'a'
- a $\cdot>$ b implies 'b' has lower precedence than 'a'

The challenge lies in the determination of correct precedence relations between terminals which are used for constructing the Operator precedence parsing table. For now let us assume the existence of the operator precedence parsing table. The determination of precedence relations are based on the traditional notions of associativity and precedence of operators. In the expression grammar, the precedence relations could be identified between all pairs of operators based on associativity and precedence and unary minus alone cause's problem.

The intention of the precedence relations is to find the handle of a right-sentential form. The following conventions are used

- $<\cdot$ with marking the left end,
- $=\cdot$ appearing in the interior of the handle, and
- $\cdot>$ marking the right hand.

The input string is prefixed and suffixed with "$" which would look like, " $\$a_1a_2...a_n\$$ " and then we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Example 11.2 Consider the following ambiguous grammar with productions

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

Let us assume the precedence table as given in Table 12.1. The next section of this module will deal with the construction of this precedence table.

Table 12.1 Precedence table for ambiguous expression grammar

	Id	**+**	*****	**$**
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*****	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Then the input string id+id*id with the precedence relations inserted will look like

"$ $<\cdot$ id $\cdot>$ + $<\cdot$ id $\cdot>$ * $<\cdot$ id $\cdot>$ $"

The precedence relation is inserted between two terminals, by considering the first terminal in the row and the second terminal in the column. Thus, "$" and "id" has $<\cdot$ Thus, the next pair is "id" and "+" which has a $\cdot>$ relation and is also inserted

The operator precedence parsing algorithm is a two step process. The stack is looked and scanned for the following

1. Scan the string from left end until the first $\cdot>$ is encountered.
2. Then scan backwards (to the left) over any $=\cdot$ until a $<\cdot$ is encountered.

This algorithm also uses a stack where the initial stack contents are the modified input string with precedence added. The handle contains everything to left of the first $\cdot>$ and to the right of the $<\cdot$ is encountered. The parsing action by the operator precedence parser is given in Table 12.2

Table 12.2 Parsing action for the ambiguous expression grammar

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.

After discussing the overview of the operator precedence parsers, we will discuss each step of the operator precedence parser in detail. Steps involved in the construction of the parser are as follows.

1. Ensure the Grammar satisfies the pre-requisite
2. Compute the functions Leading and Trailing
3. Using the computed leading and trailing, construct the Operator precedence parsing table
4. Parse the string based on the algorithm

In this module, we will discuss the computation of Leading and Trailing followed by the construction of the parsing table.

12.3 LEADING and TRAILING computation

LEADING is defined for every non-terminal. It is defined for each non-terminal such that, terminals that can be the first terminal in a string derived from that non-terminal. Similarly, TRAILING for each non-terminal are those terminals that can be the last terminal in a string derived from that NT. Formally, the functions LEADING and TRAILING are defined as follows:

LEADING(A) = { a| A $\Rightarrow^{+}$ γaδ, where γ is ∈ or a single non-terminal., where => indicates derivation, + indicates in one or more steps, A is a non-terminal. Thus LEADING(A) can be interpreted as looking for the first terminal from the left, in the RHS of a production by applying all possible derivations for a production

TRAILING(A) = { a| A $\Rightarrow^{+}$ γaδ, where δ is ∈ or a single non-terminal., where => indicates derivation, + indicates in one or more steps, A is a non-terminal. Thus TRAILING(A) can be interpreted as looking for the first terminal from the right, in the RHS of a production by applying all possible derivations for a production.

The algorithm for finding LEADING(A) where A is a non-terminal is given in Algorithm 12.1

Algorithm 12.1

LEADING(A)

{

1. 'a' is in Leading(A) if A → γaδ where γ is ε or any Non-Terminal
2. If' 'a' is in Leading(B) and A → Bα, then a in Leading(A)

}

Step 1 of algorithm 12.1, indicates how to add the first terminal occurring in the RHS of every production directly. Step 2 of the algorithm indicates to add the first terminal, through another non-terminal B to be included indirectly to the LEADING() of every non-terminal.

Similarly the algorithm to find TRAILING (A) is given in algorithm 12.2

Algorithm 12.2

TRAILING (A)

{

1. a is in Trailing(A) if A → γaδ where δ is ε or any Non-Terminal
2. If a is in Trailing(B) and A → αB, then a in Trailing(A)

}

Algorithm 12.2 is similar to algorithm 12.1 and the only difference being, the symbol is looked from right to left as against left to right in algorithm 12.1. Step 1 of the algorithm 12.2, indicates looking for the first terminal occurring in the RHS of a production from right side and thus adds the direct first symbol. The second step looks for adding the indirect first symbol from the right of the RHS of the production.

Example 12.2 Consider the unambiguous version of the expression grammar.

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Let us consider the productions of F from productions 5, 6. From production 6, there is only one symbol in the RHS and that is a terminal. So, "id" will be in the LEADING(F). From production 5, the first symbol itself is a terminal "(" and hence that will be added to LEADING(F) as well. Further additions to LEADING(F) is not possible as the first symbol in the RHS of the productions involving F is a terminal. Thus

LEADING (F) = {(, id}

Let us consider the production of T as defined in production 3, 4. From production 3, the first symbol on the RHS is a non-terminal and the first terminal is "*" and thus "*" will be in LEADING(T). Then from production 4, there is only one symbol in the RHS and that is a non-terminal. So, the LEADING of the RHS non-terminal will be there in the LHS non-terminal also. Thus, LEADING(T) will include LEADING(F) in addition to "*".

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

A similar analogy as explained for non-terminal T is applied to define the LEADING(E). Thus LEADING(E) will include LEADING(T) from production 2 and "+" from production 1. Thus

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

Let us consider computation of TRAILING () where it is similar to LEADING() but the productions are scanned from right to left. Let us again start from F. From production 6, "id" will be in TRAILING(F) as it is the only symbol in the RHS. In addition, from production 5, ")" will be in the TRAILING(F) and will not have any more symbols as the first symbol from the right is a terminal in both productions.

TRAILING (F) = {), id}

Similarly from productions 3, 4, the TRAILING(T) would include "*" and TRAILING(F) from the two productions respectively. Thus,

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

Similarly from productions 1,2, the TRAILING(E) would include "+" and TRAILING(T) from these two productions. Thus,

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

12.4 Operator precedence parsing table construction

After computing the two functions LEADING() and TRAILING(), the operator precedence table is constructed between all the terminals in the grammar including the "$" symbol. The algorithm for computing this parsing table is given in Algorithm 12.3

Algorithm 12.3

PARSINGTABLE(Grammar G, LEADING(), TRAILING())

{

For each production A → $X_1X_2X_3 ...X_n$

for i = 1 to n-1

1. if X_i and X_{i+1} are terminals
   set $X_i = \cdot X_{i+1}$
2. if $i \leq$ n-2 and X_i and X_{i+2} are terminals and X_{i+1} is a non-terminal
   set $X_i = \cdot X_{i+2}$
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 $X_i <\cdot$ **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 $\cdot> X_{i+1}$

}

Example 12.3

Consider the unambiguous expression grammar involving as given in example 12.2. Step 1 is not to be encountered in the expression grammar where two terminals occur adjacent to each other. Step 2 of the algorithm looks for a Non-terminal between a pair of terminals and the RHS should have only three symbols, then we go for the same precedence. Using this step, the terminals "(" and ")" of the 5th production of the Expression grammar gets the same precedence and is given in Table 12.3. Since, it is the only production obeying this rule, the same precedence situation will not arise for any other pair of terminals.

Consider step 3 of the algorithm, where we are looking for a terminal followed by a non-terminal. Productions 1, 3, 5 will fall in this category. We are looking at "+" and "T" in

production 1. LEADING(T) = {*, (, id}. So we set + $\lessdot$ {*, (, id} and this is shown in row 1 of Table 12.3. Similarly, from production 3, we set * $\lessdot$ { (, id} and is given in row 2 of the table 12.3. So, is the case from production 5 we set ($\lessdot$ {+, *, (, id}

Consider step 4 of the algorithm, where we are looking for a terminal preceded by a non-terminal. Similarly productions 1, 3, 5 will be used for this step of the algorithm also. Consider production 1 so we set, TRAILING(E) $\gtrdot$ X_{i+1} . Therefore, {), id} $\gtrdot$ + and is shown in row 5, row 3 of the algorithm. Similarly, {*,), id} $\gtrdot$ *. From production 5, {+, *,), id} $\gtrdot$). This parsing table is later used by the operator precedence parser.

Table 12.3 Parsing table construction

	+	*	id	(	)	$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
id	$\gtrdot$	$\gtrdot$			$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$=\cdot$	
)	$\gtrdot$	$\gtrdot$			$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		

Summary: This module detailed on the pre-requisite for a grammar to be parsed using operator precedence parser with a brief note on the operator precedence parsing algorithm. This module also dealt with computing Leading, trailing and using which the operator parsing table is also constructed.