**CD Practical 14:**

**To Study about Yet Another Compiler-Compiler(YACC).**

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

E -> E + E

E -> E \* E

E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E \* E (r2)

-> E \* z (r3)

-> E + E \* z (r1)

-> E + y \* z (r3)

-> x + y \* z (r3)

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

1 . x + y \* z shift

2 x . + y \* z reduce(r3)

3 E . + y \* z shift

4 E + .y \* z shift

5 E + y . \* z reduce(r3)

6 E + E . \* z shift

7 E + E \* .z shift

8 E + E \* z .reduce(r3)

9 E + E \* E .reduce(r2) emit multiply

10 E + E .reduce(r1) emit add

11 E .accept

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction.

Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a shift-reduce conflict. Our grammar is ambiguous because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a reduce-reduce conflict. With an id on the stack we may reduce to T, or E.

E -> T

E -> id

T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.