Session 5: Use of CFGs for Parsing

I. OBJECTIVES:

We can think of using CFGs to parse various language constructs in the token streams freed from simple syntactic and semantic errors, as it is easier to describe the constructs with CFGs. But CFGs are hard to apply practically. In this session, we implement a simple recursive descent parser to parse a number of types of statements after exercising with simpler CFGs. We note that a recursive descent parser can be constructed from a CFG with reduced left recursion and ambiguity.

II. DEMONSTRATION OF USEFUL RESOURCES:

1. Observe the C code segments that implements the non-terminals of the following CFG.

$S \rightarrow b \mid AB$
$A \rightarrow a \mid aA$
$B \rightarrow b$

Language generated: {b, ab, aab, aaab, ...}

```
void S() {
        if (str[i] == 'b'){
                i++;
                f=1;
                return;
        }
        else {
        A();
        if (f) { B(); return;}
        }}
```

```
void A() {
        if (str[i] == 'a') {
                i++;
                f=1;
        }
        else {f=0; return;}
if (i<l-1) A();
}
```

```
void B() {
        if (str[i] == 'b') {
                i++;
                f=1;
                return;
        }
        else {f=0; return;}
}
```

2. A CFG to describe the syntax of simple arithmetic expressions may look like the one that follows:

<Exp>→<Term> + <Term> | <Term> - <Term> | <Term>

<Term>→<Factor> * <Factor> | <Factor> / <Factor> | <Factor>

<Factor>→(<Exp>) | ID | NUM

ID → a|b|c|d|e

NUM→ 0|1|2|…|9

Non-terminal symbols:

<Exp>, <Term>, <Factor>

Terminal symbols:

+, -, *, /, (,), a, b, c, d,e, 0, 1, 2, 3, ..., 9

Start symbol:

<Exp>

III. LAB EXERCISE:

1. Implement the following CFG, in the way shown above or with the help of a user defined stack, that is, in the way it may work in an implementation of the transition function of a PDA.

   $A \rightarrow aXd$
   $X \rightarrow bbX$
   $X \rightarrow bcX$
   $X \rightarrow \varepsilon$

2. Implement the CFG shown above for generating simple arithmetic expressions.

IV. ASSIGNMENT #5:

Implement the following grammar in C.

<stat>→<asgn_stat> | <dscn_stat> | <loop_stat>

<asgn_stat>→id = <expn>

<expn>→**<smpl_expn>** <extn>

<extn>→<relop> **<smpl_expn>** | ε

<dcsn_stat>→ if (<expn>) <stat> <extn1>

<extn1>→ else <stat> | ε

<loop_stat>→while (<expn>) <stat> | for (<asgn_stat> ; <expn> ; <asgn_stat>) <stat>

<relop>→ == | != | <= | >= | > | <

Note: <**smpl_expn**> can be implemented using the materials demonstrated in this session.