

## 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 → b | AB
A → a | aA
B → 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;}
}
```

**\*\* Find if there is any logical error in the sample code shown above.**

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.

$A \rightarrow aXd$

$X \rightarrow bbX$

$X \rightarrow bcX$

$X \rightarrow \varepsilon$

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

### IV. ASSIGNMENT #5:

Implement the following grammar in C.

$\langle \text{stat} \rangle \rightarrow \langle \text{asgn\_stat} \rangle \mid \langle \text{dscn\_stat} \rangle \mid \langle \text{loop\_stat} \rangle$

$\langle \text{asgn\_stat} \rangle \rightarrow \text{id} = \langle \text{expn} \rangle$

$\langle \text{expn} \rangle \rightarrow \langle \text{simpl\_expn} \rangle \langle \text{extn} \rangle$

$\langle \text{extn} \rangle \rightarrow \langle \text{relop} \rangle \langle \text{simpl\_expn} \rangle \mid \varepsilon$

$\langle \text{dcsn\_stat} \rangle \rightarrow \text{if} ( \langle \text{expn} \rangle ) \langle \text{stat} \rangle \langle \text{extn1} \rangle$

$\langle \text{extn1} \rangle \rightarrow \text{else} \langle \text{stat} \rangle \mid \varepsilon$

$\langle \text{loop\_stat} \rangle \rightarrow \text{while} ( \langle \text{expn} \rangle ) \langle \text{stat} \rangle \mid \text{for} ( \langle \text{asgn\_stat} \rangle ; \langle \text{expn} \rangle ; \langle \text{asgn\_stat} \rangle ) \langle \text{stat} \rangle$

$\langle \text{relop} \rangle \rightarrow == \mid != \mid <= \mid >= \mid > \mid <$

Note:  **$\langle \text{simpl\_expn} \rangle$**  can be implemented using the materials demonstrated in this session.