Session 5: Use of CFGs for Parsing

I. OBJECTIVE:

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 implement 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;}
}
```

** 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.

<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.

Session 6: Predictive Parsing

I. OBJECTIVES:

Manual implementation of LL(1) and LR(1) parsing algorithms.

II. DEMONSTRATION OF USEFUL RESOURCES:

1. Computation of the FIRST and FOLLOW functions as described below.

❖ To Compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

a. If X is a terminal, then FIRST(X) is {X}.

b. If X is a non-terminal and $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place b in FIRST(X) if for some i, b is in FIRST(Y_i), and ε is in all of FIRST(Y_1), …, FIRST(Y_{i-1});
that is, $Y_1, \ldots, Y_{i-1}$ derives ε.

c. If ε is in FIRST(Y_j) for all j = 1, 2, …, k then add ε to FIRST(X).

Sample input and corresponding output:

```
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id
```

```
FIRST(E) = {(, id}
FIRST(T) = {(, id}
FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FIRST(F) = {(, id}
```

❖ To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

i. Place $ in FOLLOW(S), where S is the start symbol and $ is the right end-marker of an input.

ii. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B).

iii. If there is a production $A \rightarrow \alpha B$, then everything in FOLLOW(A) is in FOLLOW(B).

iv. If there is a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Sample input and corresponding output:

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

FOLLOW(E) = {$,)}
FOLLOW(T) = {+, $,)}
FOLLOW(E') = {), $}
FOLLOW(T') = {+,),$}
FOLLOW(F) = {*, +, $,)}

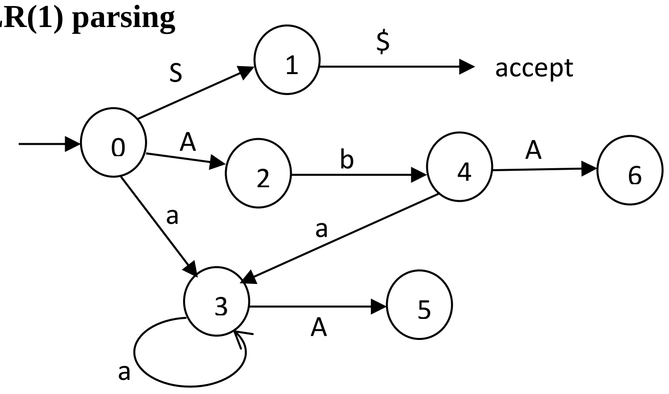
Table for LL(1) non-recursive predictive parsing with the given grammar:

Non-terminal	Input symbols					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *F T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

III. An example of construction of tools for LR(1) parsing

1. $S \to AbA$
2. $A \to aA$
3. $A \to a$

$S' \to S$
$S \to AbA$
$A \to aA$
$A \to a$



State	ACTION			GOTO	
	a	b	$	S	A
0	s3			1	2
1			acc		
2		s4			
3	s3	r3	r3		5
4	s3				6
5		r2	r2		
6			r1		

FOLLOW(S) = {$}

FOLLOW(A) = {b, $}

FIRST(S) = FIRST(A) ={a}

If $A \to \alpha\bullet$ is in I_i, then set ACTION(i, a) to "Reduce by $A \to \alpha$" for all a in FOLLOW(A).

I_3 contains $A \to a\bullet$; I_5contains $A \to aA\bullet$; I_6contains $S \to AbA\bullet$.

IV. LAB EXERCISE:

Perform the tasks1, 2, and 3 of the Assignment #6 which is described below.

V. ASSIGNMENT #6:

Suppose, you are given the following grammar and the input string ***abcd***.

$$
\boxed{
\begin{array}{l}
S \rightarrow aXd \\
X \rightarrow YZ \\
Y \rightarrow b \\
Y \rightarrow \varepsilon \\
Z \rightarrow cX \\
Z \rightarrow \varepsilon
\end{array}
}
$$

- ❑ You are required to perform the following tasks manually:
  1. Find the FIRST and FOLLOW sets of each of the non-terminals.
  2. Construct the predictive parsing table for LL(1) method.
  3. Demonstrate the moves of the LL(1) parser on the given input.
  4. Construct the LR(0) automaton for the grammar.
  5. Construct the parsing table for LR(1) parsing with the grammar.
  6. Demonstrate the moves of the LR(1) parser on the given input.