Computer Science and Engineering
IIIT Kalyani, West Bengal

Compiler Design (CS 501) Autumn:2019-2020
3^{rd} Year B.Tech.

LL(1) Recursive Descent Parser: An Example

We have not implemented any *recursive descent* LL(1) parser in the lab. Here is a very small example. Kindly note that this will be part of your theory and laboratory examination. I request you to go through it carefully (let me know if there is any error).

Consider the grammar $G = (\Sigma, N, R, S)$ where *terminals* are $\Sigma = \{+, *, \texttt{fc, \textbackslash n}\}$, *non-terminals* or *variables* are $\{S, E\}$, the *start symbol* is S and the *production rules* are as follows

$$\begin{aligned} S &\rightarrow E \text{ `}\texttt{\textbackslash n}' \\ E &\rightarrow E\ E\ + \mid E\ E\ * \mid \texttt{fc} \end{aligned}$$

The regular expression for `fc` is `[0-9]*\.[0-9]+`.

1. Transform the grammar to an $LL(1)$ grammar by removing the *left-recursion* and *left-factoring*.
2. Write a *scanner* (*lexical analyzer*) to generate tokens. You may use *flex* or write your own scanner.
3. Write *recursive descent* parser in C language.
4. If you are using *flex*, the parser code and `main()` should be in the *User code* part of the *flex* file.

Sample input/output:

```
$ a.out
1.0
Accept
$ a.out
1.0 2.0 +
Accept
$ a.out
1.0 20. *
Accept
$ a.out
1.0 2.0 + 3.5 *
Accept
```

```
$ a.out
1.5 + 2.5
Reject
$ a.out
1.5 2.5 3.5 + *
Accept
$ a.out
1.3 2.4 5.1 +
Reject
```

Ans. The production rules after the removal of *left-recursion* are

$$\begin{aligned} S &\rightarrow E \text{ `}\texttt{\textbackslash n}' \\ E &\rightarrow \texttt{fc}\ F \\ F &\rightarrow E\ +\ F \mid E\ *\ F \mid \varepsilon \end{aligned}$$

The production rules after the *left-factoring* are

$$\begin{aligned} S &\rightarrow E \text{ `}\texttt{\textbackslash n}' \\ E &\rightarrow \texttt{fc}\ F \\ F &\rightarrow E\ H \mid \varepsilon \\ H &\rightarrow +\ F \mid *\ F \end{aligned}$$

Non-terminal	First	Follow
S	{`fc`}	`eof`
E	{`fc`}	{`\n`, $+$, $*$}
F	{`fc`, ε}	{`\n`, $+$, $*$}
H	{$+$, $*$}	

```
/*
  example.l
  $ flex example.l
  $ cc -Wall lex.yy.c
  $ ./a.out
*/
%{

#include <stdio.h>
#define FC 301
#define OK 1
#define ERRTOK 1
#define ERR 0
#define UNDEF -1

int S();
int E();
int F();
int H();

int token=UNDEF;

%}

%option noyywrap

FC    [0-9]*\.[0-9]+

%%

"\n"         { return (int)'\n'; }
"*"          { return (int)'*'; }
"+"          { return (int)'+'; }
{FC}         { return FC; }
[ \t]        { ; }
.            { return ERRTOK; }
%%
int H(){
    if(token == UNDEF) token = yylex();
    if(token == '+' || token == '*') {
        token = UNDEF;
        return F();
    }
    return ERR;
}

int F(){
    if(token == UNDEF) token = yylex();
    if(token == '\n' ||
       token == '+' ||
       token == '*') return OK;
    if(token == FC && E() == OK) return H();
    else return ERR;
}

int E(){
    if(token == UNDEF) token = yylex();
    if(token == FC) {
      token = UNDEF;
      return F();
    }
    else return ERR;
}
```

```
int S(){
    if(token == UNDEF) token = yylex();
    if(token == FC) {
      if(E() == OK){
        if(token == UNDEF) token = yylex();
        if(token == '\n') return OK;
        else return ERR;
      }
    }
    return ERR;
}

int main(){
    if(S() == OK) printf("Accept\n");
    else printf("Reject\n");

    return 0;
}
```