

Eastern Mediterranean University

CMSE318-CMPE410 Principles of Programming Languages  
Spring 2022-2023

Assignment 3

Parser Application

To be done in groups of two. Pick your partner!

We are given the following grammar for generating arithmetic expressions.

G1:

$G \rightarrow E$

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow ( E ) \mid M \mid N$

$M \rightarrow a \mid b \mid c \mid d$

$N \rightarrow 0 \mid 1 \mid 2 \mid 3$

This grammar is not suitable for top down recursive descent parsing because of left recursion.

Removing left recursion, we get the grammar G2:

G2:

$G \rightarrow E$

$E \rightarrow T R$

$R \rightarrow + T R \mid - T R \mid \epsilon$

$T \rightarrow F S$

$S \rightarrow * F S \mid / F S \mid \epsilon$

$F \rightarrow ( E ) \mid M \mid N$

$M \rightarrow a \mid b \mid c \mid d$

$N \rightarrow 0 \mid 1 \mid 2 \mid 3$

" $\epsilon$ " means the empty production, i.e. if  $T \epsilon \rightarrow$ , then T can derive the empty string.

Now, we can write recursive functions for parsing according to G2. Here is the pseudo-code:

```

/* G -> E */

Bool error = FALSE;
char next_token = '%';

void main(){
    /* open file for reading */
    .....
    G();
}

void G(){
    lex();
    print ("G -> E");
    E();
    if (next_token=='$' and !(error)) {
        print "success";
    }
    else {
        print ("failure: unconsumed input=%s", unconsumed_input());
    }
}

/* E -> T R */
void E(){
    if (error) return;
    print ("E -> T R");
    T();
    R();
}

/* R -> + T R | - T R | e */
void R(){
    if (error) return;
    if (next_token=='+') {
        print ("R -> + T R");
        lex();
        T();
        R();
    }
    else if (next_token=='-') {
        print ("R -> - T R");
        lex();
        T();
        R();
    }
}

```

```

    }
    else {
        print ("R->e");
    }
}

/* T -> F S */
void T(){
    if (error) return;
    print (" T -> F S");
    F();
    S();
}

/*      S -> * F S | / F S | e      */
void S(){
    if (error) return;
    if (next_token=='*') {
        print ("S -> * F S");
        lex();
        F();
        S();
    }
    else if (next_token=='/') {
        print "S -> / F S"
        lex();
        F();
        S();
    }
    else {
        print "S -> e"
    }
}

/*      F -> ( E ) | N      */
void F(){
    if (error) return;
    if (next_token=='(') {
        print "F->( E)";
        lex();
        E();
        if (next_token == ')') {
            lex();
        }
    }
    else { error=TRUE;
        print("error: unexptected token ", next_token);
    }
}

```

```

        print("unconsumed_input ", unconusmed_input());
        return;
    }
}
else if (next_token is one of 'a' or 'b' or 'c' or 'd') {
    print ("F->M");
    M();
}
else if (next_token is one of '0' or '1' or '2' or '3') {
    print ("F->N");
    N();
}

else {
    error=TRUE;
    print("error: unexptected token ", next_token);
    print("unconsumed_input ", unconusmed_input());
}
}

```

*/\* M → a | b | c | d \*/*

```

void M(){
    if (error) return;
    if (next_token is one of 'a' or 'b' or 'c' or 'd') {
        print ("M->", next_token);
        lex();
    } else {
        error=TRUE;
        print("error: unexptected token ", next_token);
        print("unconsumed_input ", unconusmed_input());
    }
}

```

*/\* N → 0 | 1 | 2 | 3 \*/*

```

void N(){
    if (error) return;
    if (next_token is one of '0' or '1' or '2' or '3') {
        print ("N->", next_token);
        lex();
    } else {
        error=TRUE;
        print("error: unexptected token ", next_token);
    }
}

```

```
        print("unconsumed_input ", unconusmed_input());  
    }  
}
```

OK, now that the parser is mostly written for you, here is what you will do.

Write parser program (based on the pseudo-code given above) *in Python* to parse expressions as defined by the grammar above. The input should be in a file. You should have global variables *error* (of type *Boolean*), and *next\_token* (of type *char*). Define a function *lex()* that gets the next character from the file and places it inside *next\_token*. *lex()* should skip any white spaces, such as newlines or the space character. The function *unconusmed\_input()* should return the remaining input in the file. The last character in the file should always be \$. Define functions (hint: class methods) *G()*, *E()*, *R()*, *T()*, *F()* and *N()*. In the *main()* function, open the file containing the expression and call *G()*.

### What to hand in:

Zip the following and upload to Teams.

1. A report containing
  - a. a description of the problem(what your program does)
  - b. description of your program (how it does what it does)
  - c. tutorial on running your program
  - d. 5 sample runs that parse correctly (i.e. 5 different inputs that produce no error)
  - e. 5 sample runs that produce errors (5 different inputs that produce errors)
2. Commented source code
3. Input files (files containing the inputs to the parser)

The filename of your uploaded file should be of the format:  
**studentID1\_studentID2\_CMSE318\_CMPE410\_Assignment3.zip**