Nicholas Jordan
CMPS 3500
LISP Interpreter
Professor Chengwei Lei

LISP Interpreter Screenshots

Notes about the design for each operation:

- All ‘space’ characters are ignored.

```
else if (*expr == ' ') {
  //ignore spaces
```

- Whenever a function that scans through the string returns, we need to catch back up to wherever that function left off.
    - ‘getNum()’ does this using a pass-by-pointer parameter ‘length’ so that we can iterate the char* to the end of the integer so that we don’t read integers that appear within our integer. For example, 342 contains integers 42 and 2 so we want to skip past these when 342 is read.

```
int getNum(char* expr, int* length) {
    int num;
    sscanf(expr++, "%d", &num);
    if (num == 0) {
        *length = 1;
    } else {
        *length = floor(log10(abs(num)))+1;
    }
    return num;
}
```

```
expr += length;
```

    - Nested calls to ‘eval()’ do this by using the syntax of operations to our advantage, but first we need to measure recursive call depth by using static global variables ‘depth’ and ‘max-depth’. Each recursive call to ‘eval()’ iterates the ‘depth’ variable, and returning from the recursive call subtracts from the ‘depth’ variable. So, after returning from each recursive call we can simply scan through n, where n = max_depth - depth, ‘)’ characters and we will be where the furthest nested call left off.
        - It should also be noted that returning to a ‘depth’ of 0 resets the ‘max_depth’ to 0.

```
depth++;
sum += (int)eval(expr);
for (int i = 0; i < max_depth - depth; i++) {
        while (*expr++ != ')') {}
}
if (depth == 0) max_depth = 0;
```

```
depth--;
return sum;
```

- ‘isdigit()’ is used to identify the start of an integer.

```
if (isdigit(*expr))
```

- The beginning of an operation is marked by the ‘(‘ character.
    - If a ‘(‘ is found inside an operation, then a recursive call to ‘eval()’ is made so that the nested operation is evaluated before the rest of the current operation.

```
else if (*expr == '(') {
  depth++;
  sum += (int)eval(expr);
```

- The end of an operation is marked by the ‘)’ character.
    - There is an error if a ‘\n’ or ‘\0’ character is found before the end of an operation.

```
while (*(expr++) != ')') {
    if (*expr == '\n' || *expr == '\0') {
        strcpy(err, "Unfinished operation");
        return 0;
```

    - There is an error if a ‘)’ is found before any numbers.

```
} else {
    //any other character is invalid
    strcpy(err, "Invalid character: ");
    strcat(err, &expr[0]);
    return 0;
}
```

- ‘eval()’ returns a float to accommodate for division operations.

```
float eval(char* expr) {
    if (*expr != '(') {
        strcpy(err, "Expression must begin with '('");
        return 0;
    }
    float result;
    while (*expr++ != '\n') {
        if (*expr == OP_ADD) {
            result = compute_add(expr++);
            break;
        } else if (*expr == OP_SUB) {
            result = compute_sub(expr++);
            break;
        } else if (*expr == OP_MUL) {
            result = compute_mul(expr++);
            break;
        } else if (*expr == OP_DIV) {
            result = compute_div(expr++);
            break;
        } else if (*expr == OP_MOD) {
            result = compute_mod(expr++);
            break;
        } else if (*expr == ' ') {
            //ignore spaces
        } else if (*expr == '\n' || *expr == ')') {
            strcpy(err, "Unfinished operation");
            return 0;
        } else {
            //any other character is invalid
            strcpy(err, "Invalid character: ");
            strcat(err, expr);
            return 0;
        }
    }
    return result;
}
```

Part 1:

Adding: The add operation was used as a proof of concept for the other operations, so once this function was finished the others were almost a copy and paste.

```
int compute_add(char* expr) {
    if (*(expr+1) == ')' || *(expr+1) == '\n' || *expr == '\0') {
        strcpy(err, "Unfinished operation");
        return 0;
    }

    if (depth > max_depth) {
        max_depth = depth;
    }
    int sum = 0;
    while (*(expr++) != ')') {
        if (*expr == '\n' || *expr == '\0') {    //needs to end with ')'
            strcpy(err, "Unfinished operation");
            return 0;
        } else if (isdigit(*expr)) {
            int length = 0;
            sum += getNum(expr, &length);
            expr += length;
        } else if (*expr == '(') {
            depth++;
            sum += (int)eval(expr);
            //after calling eval(), expr points to where it was prior to the call
            //so we need to catch back up to the last considered instance of ')'
            for (int i = 0; i < max_depth - depth; i++) {
                    while (*expr++ != ')') {}
            }
            if (depth == 0) max_depth = 0;
        } else if (*expr == ' ') {
            //ignore spaces
        } else if (*expr == ')') {
            break;
        } else {
            //any other character is invalid
            strcpy(err, "Invalid character: ");
            strcat(err, &expr[0]);
            return 0;
        }
    }
    depth--;
    return sum;
}
```

Part 2:
Subtraction:

```
int compute_sub(char* expr) {
    if (depth > max_depth) {
        max_depth = depth;
    }
    int diff;
    int first = 0;  //flag whether the first number has been acknowledged
    while (*expr++ != ')') {
        if (*expr == '\n' || *expr == '\0') {    //needs to end with ')'
            strcpy(err, "Unfinished operation");
            return 0;
        } else if (isdigit(*expr)) {
            int length = 0;
            int n = getNum(expr, &length);
            if (first == 0) {
                diff = n;
                first = 1;    //set flag
            } else {
                diff -= n;
            }
            expr += length;
        } else if (*expr == '(') {
            depth++;
            if (first == 0) {
                diff = (int)eval(expr);
                first = 1;    //set flag
            } else {
                diff -= (int)eval(expr);
            }
            //after calling eval(), expr points to where it was prior to the call
            //so we need to catch back up to the last considered instance of ')'
            for (int i = 0; i < max_depth - depth; i++) {
                    while (*expr++ != ')') {}
            }
            if (depth == 0) max_depth = 0;
        } else if (*expr == ' ') {
            //ignore spaces
        } else {
            //any other character is invalid
            strcpy(err, "Invalid character: ");
            strcat(err, expr);
            return 0;
        }
    }
    depth--;
    return diff;
}
```

Multiplication:

```
int compute_mul(char* expr) {
    if (depth > max_depth) {
        max_depth = depth;
    }
    int prod;
    int first = 0;  //flag whether the first number has been acknowledged
    while (*expr++ != ')') {
        if (*expr == '\n' || *expr == '\0') {    //needs to end with ')'
            strcpy(err, "Unfinished operation");
            return 0;
        } else if (isdigit(*expr)) {
            int length = 0;
            int n = getNum(expr, &length);
            if (first == 0) {
                prod = n;
                first = 1;    //set flag
            } else {
                prod *= n;
            }
            expr += length;
        } else if (*expr == '(') {
            depth++;
            if (first == 0) {
                prod = (int)eval(expr);
                first = 1;    //set flag
            } else {
                prod *= (int)eval(expr);
            }
            //after calling eval(), expr points to where it was prior to the call
            //so we need to catch back up to the last considered instance of ')'
            for (int i = 0; i < max_depth - depth; i++) {
                    while (*expr++ != ')') {}
            }
            if (depth == 0) max_depth = 0;
        } else if (*expr == ' ') {
            //ignore spaces
        } else {
            //any other character is invalid
            strcat(err, "Invalid character: ");
            strcat(err, expr);
            return 0;
        }
    }
    depth--;
    return prod;
}
```

Division:

```
float compute_div(char* expr) {
    if (depth > max_depth) {
        max_depth = depth;
    }
    float quot;
    int first = 0;  //flag whether the first number has been acknowledged
    while (*expr++ != ')') {
        if (*expr == '\n' || *expr == '\0') {    //needs to end with ')'
            strcpy(err, "Unfinished operation");
            return 0;
        } else if (isdigit(*expr)) {
            int length = 0;
            int n = getNum(expr, &length);
            if (first == 0) {
                quot = n;
                first++;    //set flag
            } else {
                if (n == 0) {
                    strcat(err, "You cannot divide by 0");
                    return 0;
                } else {
                    quot /= n;
                }
            }
            expr += length;
        } else if (*expr == '(') {
            depth++;
            if (first == 0) {
                quot = eval(expr);
                first = 1;    //set flag
            } else {
                quot /= eval(expr);
            }
            //after calling eval(), expr points to where it was prior to the call
            //so we need to catch back up to the last considered instance of ')'
            for (int i = 0; i < max_depth - depth; i++) {
                    while (*expr++ != ')') {}
            }
            if (depth == 0) max_depth = 0;
        } else if (*expr == ' ') {
            //ignore spaces
        } else {
            //any other character is invalid
            strcpy(err, "Invalid character: ");
            strcat(err, expr);
            return 0;
        }
    }
    depth--;
    return quot;
}
```

Modulus:

```
int compute_mod(char* expr) {
    if (depth > max_depth) {
        max_depth = depth;
    }
    int remainder;
    int count = 0;  //mod can only be between 2 numbers
    while (*expr++ != ')') {
        if (*expr == '\n' || *expr == '\0') {    //needs to end with ')'
            strcpy(err, "Unfinished operation");
            return 0;
        } else if (isdigit(*expr)) {
            int length = 0; //length of the number found in getNum
            int n = getNum(expr, &length);
            if (count == 0) {
                remainder = n;
                count++;
            } else if (count == 1) {
                if (n == 0) {
                    strcat(err, "You cannot modulous by 0");
                    return 0;
                } else {
                    remainder = remainder % n;
                    count++;
                }
            } else {
                strcat(err, "You can only modulous two numbers");
                return 0;
            }
            expr += length;
        } else if (*expr == '(') {
            //modulous cant be done with floats, type cast to int
            if (count == 0) {
                depth++;
                remainder = (int)eval(expr);
                count++;
            } else if (count == 1) {
                depth++;
                int n = (int)eval(expr);
                if (n == 0) {
                    strcat(err, "You cannot modulous by 0");
                    return 0;
                } else {
                    remainder = remainder % n;
                    count++;
                }
            } else {
                strcat(err, "You can only modulous two numbers");
                return 0;
            }
            //after calling eval(), expr points to where it was prior to the call
            //so we need to catch back up to the last considered instance of ')'
            for (int i = 0; i < max_depth - depth; i++) {
                while (*expr++ != ')') {}
            }
            if (depth == 0) max_depth = 0;
        } else if (*expr == ' ') {
            //ignore spaces
        } else {
            //any other character is invalid
            strcpy(err, "Invalid character: ");
            strcat(err, expr);
            return 0;
        }
    }
    depth--;
    return remainder;
}
```

Result: The below screenshot shows the following:

- Spaces are ignored
- Every operation works
- Every operation, except modulus, can evaluate more than two operands.
- Operations can be nested within each other.

```
soup@DESKTOP-OV0U0DF:/mnt/c/Users/Nicholas/Desktop/CMPS_3500_Final$ gcc lisp.c -lm -o lisp
soup@DESKTOP-OV0U0DF:/mnt/c/Users/Nicholas/Desktop/CMPS_3500_Final$ ./lisp
Lisp interpretor is now listening for input strings
NOTE: Operations can be nested. Ex: (+ 2 (* 4 2))
NOTE: The '%' operation takes the floor of nested division operations
Usage: (<+,-,*,/,%> <integer> <integer> ... )

(     +     2    2   )
Evaluating (     +     2    2   )
Result: 4.000000

(+ 2 2)
Evaluating (+ 2 2)
Result: 4.000000

(- 4 2)
Evaluating (- 4 2)
Result: 2.000000

(* 4 2)
Evaluating (* 4 2)
Result: 8.000000

(/ 8 3)
Evaluating (/ 8 3)
Result: 2.666667

(% 7 4)
Evaluating (% 7 4)
Result: 3.000000

(+ (* 5 (/ 10 2)) (% 5 3) (- 50 20 (+ 10 (/ 100 10))) (* 5 2))
Evaluating (+ (* 5 (/ 10 2)) (% 5 3) (- 50 20 (+ 10 (/ 100 10))) (* 5 2))
Result: 47.000000
```

I'm positive that this LISP interpreter has many problems that I have either not found or not thought of, but from what I have thought of and tested it works pretty well and even has its own primitive error checking.