Assignment 1: Reverse-Polish Notation

Date: To be determined
Deadline: To be determined

Objectives

You must implement a stack API and a conversion program that converts between two notational systems for mathematical expressions.

Requirements

You should implement the stack API described in the `stack.h` file. This data structure has not been covered in the lectures yet, but it is easy enough to understand. Note that for this assignment the size of the stack is limited to a fixed number, which is also defined in the `stack.h` file. If you are unfamiliar with stacks, check out the reference links at the end of the assignment. This part of the assignment will be a general implementation to store integers in a stack data structure.

Next, you should use this stack to store characters as part of a RPN conversion program. Your conversion program must be named `infix2rpn` and must accept a single mathematical expression using infix notation on the command-line, and output the following:

- on standard output, its representation in reverse-polish notation.
- then, on standard error, a summary of the stack operations needed to perform the conversion. The difference between standard output and standard error will be explained in the C lectures.

The program must terminate with exit code 1 if it encounters an invalid input, and exit code 0 when it succeeds.

You must submit your work as a tarball. The command `make tarball` will create a tarball for you named `infix2rpn_submit.tar.gz`.

Details on the input and output formats

- Input infix expressions are formed using the following rules:
  - a non-empty sequence of decimal digits forms an expression.
  - two expressions separated by a binary operator `+ - * /` form an expression.
  - one expression between parentheses `( )` forms an expression.
  - spaces surrounding operators, operands or parentheses are meaningless and can be ignored.
- Your program must support proper precedence: `3*1+2` and `3*(1+2)` are different!
- Output RPN expressions are a space-delimited sequence of operators and non-operators.
- On the standard error, the program must print the word "`stats`" followed by three numbers separated by spaces, in this order:
  - the total number of stack "push" operations;
  - the total number of stack "pop" operations;
  - the **maximum** size of the stack during the conversion.

Example:

```
$ ./infix2rpn "3+2"
3 2 +
stats 1 1 1

# Exit code is 0 in case of success.
$ ./infix2rpn "(3+2)/3"; echo $?
3 2 + 3 /
stats 3 3 2
0

# Results go to stdout.
$ ./infix2rpn "(3+2)/3"  2> /dev/null
3 2 + 3 /

# Stats go to stderr.
$ ./infix2rpn "(3+2)/3"  > /dev/null
stats 3 3 2

# Checking that the exit status is correct in case of error
$ ./infix2rpn "blabla"  > /dev/null 2>&1; echo $?
1
```

Automated Testing

The correctness of your programs will be determined by automatic grading scripts. To help you get used to this, part of the scripts for this week have also been provided. You can test your stack with `check_stack.c`, which contains a set of testcases for just the data structure functions. The script `check_infix2rpn.sh` contains some of the grading checks that will be run on your code.

The command `make check` will run all these tests in order. Note that only half of the requirements are tested here, and you should add your own tests in `test_expressions.sh` in order to verify all elements of your program function correctly.

Getting started

1. Unpack the provided source code archive; then run `make`.
2. Try out the generated `infix2rpn` and understand how input expressions are provided.
3. Read the file `stack.h` and study the interface of functions listed there.
4. Implement the data structure in `stack.c` according to the interface description.
5. Run `make check` to see if your stack implementation is correct. Reproduce errors found by the tests (you can view the code from each test in `check_stack.c`) and fix your stack where needed.
6. Implement the conversion algorithm in `infix2rpn.c` and test this with `make check` too.
7. Add tests to `test_expressions.sh` and check the parts of the assignment that are not covered in the provided grading script.

Hint

You will not need to analyze numbers or determine the value of each number on the input. Of course, you can do this, but it is not needed to achieve a correct solution. The simple algorithm can

look at characters individually and then forget about them. Check the links referenced at the end of the assignment!

Grading

Your grade starts from 0, and the following tests determine your grade:

- +1pt if you have submitted an archive in the right format, your source code builds without errors and you have modified both `stack.c` and `infix2rpn.c` in any way.
- +1pt if your stack API processes pushes and pops properly and detects stack overflow and underflow situations.
- +1pt if your converter processes expressions of any length, without parentheses and a single precedence level, properly.
- +1pt if your converter processes expressions of any length, without parentheses and multiple precedence levels, properly.
- +0.5pt if your converter detects invalid characters properly and reports a correct exit code.
- -1pt if your code produces any warnings using the flags `-Wpedantic -Wall -Wextra` when compiling.

And the following features are not included in the provided tests at all. You will have to validate the correctness of these yourself by writing your own tests.

- +1pt if your stack API counts **valid** operations properly (number of pushes, pops and max. size).
- +0.5pt if your converter properly ignores spaces in the input expressions.
- +1.5pt if your converter processes all expressions including parentheses properly.
- +0.5pt if your converter detects improperly matched parentheses and reports a correct exit code.
- +1pt if your converter also supports *right-associative* exponentiation at a higher precedence level than multiplication, that is, `2*2^3^4` is an expression and is equivalent to `2*(2^(3^4))`, and converts it appropriately.
- +1pt if your converter also supports *unary negation* in front of simple numbers and grouped expressions using the symbol ~ (not "`-`"!), for example `~123` or `~(3+2)`.
- -1pt if the address sanitizer or running valgrind reports errors while running your converter. *Note that you cannot test both of these at the same time.*. The address sanitizer is enabled by default in our makefile, to generate an executable that you can use with valgrind run the commands `make clean; make valgrind`

Summary of operators and precedence levels

Precedence	Operator	Associativity
1	Negation	Right
2	Exponentiation	Right
3	Division and multiplication	Left
4	Addition and subtraction	Left

Reference links

- Video "What is a stack data structure"

- Wikibook - Fundamentals of data structures - Stacks
- Infix to postfix algorithm video
- Dijkstra's shunting-yard algorithm explained