

Assignment 1: Reverse-Polish Notation

February 1st 2017

Deadline: February 15th 2017 23:59

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 datastructure 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 defined in the `stack.c` file. If you are unfamiliar with stacks, check out the reference links at the end of the assignment.

Your conversion program must be named `infix2rpn` and must accept a single expression using infix notation on the command-line, and output the following:

- on the standard output, its representation in [reverse-polish notation](#).
- then, on the standard error, a summary of the stack operations needed to perform the conversion.

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 tarball¹. Next to the source code, your archive must contain a text file named `"AUTHORS"` containing your name and Student ID.

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.

¹`make tarball` will create the tarball for you.

- one expression between parentheses () forms an expression.
- spacing characters 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

# Stats go to stderr, result to stdout.
$ ./infix2rpn "(3+2)/3" 2> /dev/null
3 2 + 3 /
$ ./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
```

Getting started

1. Unpack the provided source code archive; then run `make`.
2. Try out the generated `infix2rpn` and familiarize yourself with its interface.
3. Read the file `stack.h` and understand the interface.
4. Implement the data structure in `stack.c`.
5. Write a bunch of valid and invalid expressions to serve as test input for your program.
6. Implement the conversion algorithm in `infix2rpn.c`. Use your tests to check your work.

Hint

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

- +0,5pt if you have submitted an archive in the right format with an `AUTHORS` file.
- +0,5pt if your source code builds without errors and you have modified `stack.c` or `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 stack API counts operations properly (number of pushes, pops and max. stack size).
- +1pt if your converter processes expressions without grouping and a single precedence level properly.
- +1pt if your converter processes expressions with grouping and a single precedence level properly.
- +1pt if your converter processes expressions with multiple precedence levels properly.
- +0,5pt if your converter detects invalid characters and improperly matched parentheses properly and reports a correct exit code.
- -1pt if `valgrind` reports errors while running your converter.
- -1pt if `clang -W -Wall` reports warnings when compiling your code.

The following extra features will be tested to obtain higher grades, but only if you have obtained a minimum of 5 points on the list above already:

- +1pt if your converter properly ignores spaces in the input.
- +1pt if your converter also supports *left-associative* exponentiation at a higher precedence level than multiplication, that is, $2 * a^b^c$ is an expression and is equivalent to $2 * ((a^b)^c)$, and converts it appropriately.
- +0,5pt if your converter also supports *postfix function application* at the highest precedence level, that is, $(x)_F$ is an expression (means apply function F to (x) , $3^{(4+2)_F}$ is equivalent to $3^{((4+2)_F)}$ and is converted to "`3 4 2 + F ^`"; only one-letter functions can be supported).
- +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)`. If you choose to implement both this feature and the previous one, ensure that function application has a higher precedence than negation, that is, $\sim (3)_F$ is equivalent to $\sim ((3)_F)$.

Summary of desired precedence levels:

Level	Operator
1	Function application
2	Negation
3	Exponentiation
4	Division and multiplication
5	Addition and subtraction

See also

- Video "What is a stack data structure" <https://www.youtube.com/watch?v=FNZ5o9S9prU>
- Wikibook "Programming Concepts: Stacks" https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Stacks
- Infix to postfix algorithm video: <https://www.youtube.com/watch?v=LQ-iW8jm6Mk>
- Rosetta code: infix -> rpn: https://rosettacode.org/wiki/Parsing/Shunting-yard_algorithm
- Dijkstra's shunting-yard algorithm explained: https://en.wikipedia.org/wiki/Shunting-yard_algorithm