EVALEXPR — Subject

version **#**

Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.assistants.epita.fr

Obligations

Obligations are ***fundamental*** *rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: The coding-style needs to be respected at all times.

Obligation #6: **Global variables** are forbidden, unless they are **explicitly** authorized

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Obligation #8: Your code must compile with the flags:

```
-std=c99 -pedantic -Werror -Wall -Wextra
```

Advice

- ▷ Read the *whole* subject.
- ▷ If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **newsgroup** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

[1]If an executable file is required, please provide its sources **only**. We will compile it ourselves.

Files to submit:

- ./Makefile
- ./src/*
- ./tests/*

Makefile: Your makefile should define at least the following targets:

- all: Produces the `evalexpr` binary
- evalexpr: Produces the `evalexpr` binary
- check: Runs the testsuite
- clean: Deletes everything produced by make

Authorized headers: You are only allowed to use the functions defined in the following headers:

- assert.h
- stdlib.h
- errno.h
- err.h
- string.h
- ctype.h
- stdio.h
- stddef.h

1 Goal

`evalexpr` is a program that reads on the standard input an arithmetic expression and writes the result of that expression on the standard output. The input will be of arbitrary size and can be given in reverse polish notation (*RPN*) or standard (*infix*) notation.

Here is a basic example of what is expected:

```
42sh$ echo "1 + 1" | ./evalexpr
2
```

2 Requirements

Your program can either be called with no argument and handle infix notation or with `-rpn` and handle RPN. All other arguments are invalid.

All operations will be done on integer numbers.

Tips

There will be no test trying to overflow `int` max capacity.

We advise you to start the project with the evaluation of RPN expressions. Once this is done, infix notation can be converted to RPN using the shunting-yard algorithm and then evaluated.

2.1 Valid expressions in reverse polish notation

RPN expressions are easy to parse and interpret. Therefore, they are a good way to start the project. These expressions will contain:

- Whitespaces (as recognized by `isspace(3)`)
- Numbers in base 10 (digits from 0 to 9)
- The binary addition operator: +
- The binary subtraction operator: -
- The binary multiplication operator: *
- The binary division operator: /
- The binary modulo operator (the one from *C* language): %
- The binary exponentiation operator: ^

Here are some examples:

```
42sh$ echo "1 1 +" | ./evalexpr -rpn
2
42sh$ echo $?
0
42sh$ echo "5 2 2 ^ 3 + *" | ./evalexpr -rpn | cat -e
35$
```

2.2 Valid expressions in standard notation

The expression will be given using infix notation, with the properties that you already know. These expressions will contain:

- Whitespaces (as recognized by `isspace(3)`)
- Numbers in base 10 (digits from 0 to 9)
- The unary identity operator: +

- The unary negation operator: -
- The binary addition operator: +
- The binary subtraction operator: -
- The binary multiplication operator: *
- The binary division operator: /
- The binary modulo operator (the one from *C* language): %
- The binary exponentiation operator: ^
- Left and right parenthesis: (and)

Here is a list of these operators from highest to lowest priority:

- Parenthesis (and), non-associative
- Unary + and -, right-associative
- ^, right-associative
- *, /, %, left-associative
- Binary + and - left-associative

In case of equal priority, operations will be executed left to right for left-associative operators, and right to left for right-associative operators. You have to respect operators priority, with parenthesis having the highest priority.

Here are some examples:

```
42sh$ echo "1+1" | ./evalexpr
2
42sh$ echo $?
0
42sh$ echo "5*(2^2+3)" | ./evalexpr | cat -e
35$
```

You can have many unary operators like this:

```
42sh$ echo '86*--1' | ./evalexpr
86
42sh$ echo '3---+-+4' | ./evalexpr
7
```

2.3 Error handling

The return code for successful operations is 0. If any error occurs, nothing will be printed on the standard output and you will have to return an error code depending on cases listed below. Error output will not be tested.

If there is no expression, your program shall do nothing and return 0.

```
42sh$ echo | ./evalexpr 2>/dev/null
42sh$ echo $?
0
```

If there is a lexical error (invalid character), return 1.

```
42sh$ echo "a+1" | ./evalexpr 2>/dev/null
42sh$ echo $?
1
```

If there is a syntax error (invalid expression), return 2.

```
42sh$ echo "1(+1)" | ./evalexpr 2>/dev/null
42sh$ echo $?
2
```

If there is an arithmetical error (division or modulo by 0), return 3.

```
42sh$ echo "1%0" | ./evalexpr 2>/dev/null
42sh$ echo $?
3
```

For any other error (bad argument for example), return 4.

```
42sh$ ./evalexpr --toto 2>/dev/null
42sh$ echo $?
4
```

Tips

For any test that should produce an error, we will only look at the standard output and return value of your program. Feel free to display an error message on `stderr` if you want.

3 Testsuite

Your project must have a Criterion testsuite containing **multiple** unit tests to check the behavior of the datastructures you use, as well as functional tests in *shell* to check the global behavior of your program.

Tips

In order to write your functional tests, `bc(1)` and `dc(1)` may prove useful.

We must be able to run your testsuite with the `make check` command.

It is my job to make sure you do yours.