Exercises — Math lexer

version #

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

Copyright

This document is for internal use at EPITA (website) only.

Contents

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

1 Math lexer

Files to submit:

- math-lexer/src/lexer/lexer.c
- math-lexer/meson.build

Provided files:

- math-lexer/meson.build
- math-lexer/src/utils/alloc.h
- math-lexer/src/ast/ast.c
- math-lexer/src/ast/ast.h
- math-lexer/src/eval/token_printer.c
- math-lexer/src/eval/rpn_print.c
- math-lexer/src/eval/ast_print.c
- math-lexer/src/lexer/lexer.h
- math-lexer/src/lexer/token.c
- math-lexer/src/lexer/token.h
- math-lexer/src/parser/parser.c
- math-lexer/src/parser/parser.h

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

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

1.1 Preamble

This exercice is **fundamental**. It gives you the basic understanding of how to start with *42sh* and specially how to begin with the two main parts : the `lexer` and `parser`. We **strongly** advise you to read **all** the given files. A lot of comments are available in each one of them.

1.2 Goal

The goal of this exercise is to write a simple lexer for arithmetic expressions (similarly to MyFind's ast-evalutation exercise). Your lexer will be given a string containing an arithmetic expression, composed of the operators `(`, `)`, `+`, `-`, `*`, `/`. If a given token does not belong to the previous list and is not a number, your lexer must print a message in `stderr`. The content of this message will not be tested.

1.3 Lexer

A lexer's purpose is to read an input, and split it into atomic blocks. These blocks are called tokens and are often associated with a type such as a value or an operator.

1.4 What to implement?

As you may have already seen, there is some given code for your lexer to have a pretty-print easily available. You have to implement the functions described in `lexer/lexer.h`. Do not worry, the header is documented in order to give you instructions about the purpose of each function.

Keep in mind that the lexer must generate tokens as the parser ask for them. In other word, your lexer analyze the input and generate a token because the parser ask for one. You cannot generate tokens in advance as it prevents interactive input of your program. It might be fine for this example but will not be for *42sh*.

1.5 Expression Parser

In your *42sh* you will be asked to implement a parser. This parser can be implemented by following the same technique we use for this exercise's given parser.

The given expression parser implements the following *LL grammar*:

```
input:      exp EOF
        |   EOF

exp:        sexp ('+' sexp | '-' sexp)*

sexp:       texp ('*' texp | '/' texp)*

texp:       NUMBER
        |   '-' NUMBER
        |   '(' exp ')'
        |   '-' '(' exp ')'
```

1.6 Example

```
42sh$ ./ast_print "1 + 3 / 4 * 9" | cat -e
(1+((3/4)*9))$
42sh$ ./rpn_print "1 + 3 / 4 * 9" | cat -e
1 3 4 / 9 * + $
42sh$ ./token_printer "1 + 3 / 4 * 9" | cat -e
1$
+$
3$
/$
4$
*$
9$
EOF$
42sh$ ./token_printer "1 + 3 % 5" 1> out 2> err
42sh$ cat -e out
1$
+$
3$
42sh$ cat -e err
lexer: token % is not valid$
```

It is my job to make sure you do yours.