ualbany-csi402-s18 / **hw08** Private

No description, website, or topics provided.

2 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

azerrima correct filename — Latest commit 8a3b99c an hour ago

README.md — correct filename — an hour ago

README.md

Homework 08

- Link to create your repository: https://classroom.github.com/a/rh689XfF
- Due date: **May 7, 2018** (**no late submission**)
- Points: 25 pt.
- Final contents of your repository: (all in the root of your repository; no directories)
  - source files (`.l` / `.y`)
  - `makefile` or `Makefile`
  - the followings files are optional and not graded:
    - test scripts (`*.sh`)
    - `.gitignore`
    - your `README.md`

The purpose of this homework assignment is to learn and practice the basics of building compilers.

Note that this repository is read-only. You should follow the above link to create your own private repository for submitting your assignment.

Notes about your code and outputs

The grading will be automated. If your files cannot be found in the expected directories and with specified names, or your code cannot be compiled, you **will receive no points** for the corresponding task. Your grade in each task depends on the number of successfully-run test cases. Make sure to fully test your code by comparing your outputs against expected outputs (e.g., using `diff`) before you submit.

Make sure to follow the below guidelines about your source code and outputs. Failure to do so may result in loosing points especially in case of the expected outputs by our grading scripts.

- You must comment your code explaining how you achieve the desired goal(s) whenever asked to write a new piece of code.
- Test your program extensively. If you use pointers in your code make sure to run your code with test cases through valgrind too. That will save you from many hours of debugging later on.
- The `itsunix` machine should be your reference in terms of output and compilation environment. Make sure to test your programs and scripts there before submission.
- There has to be **no extra leading or trailing white spaces** in your output.

- You can use all functions provided by the GNU standard library. Your code should not rely on executing external programs unless explicitly instructed.
- Make sure to handle potential errors as result of invoking system calls. Recall that you can use perror() to print an appropriate error message.
- If you are printing your own error messages, note that they need to end with a newline character and must be printed to `stderr`.

Task 0: Review the readings

In case you did not get to completely read the assigned chapters for the past week(s), make sure you do it now. Remember that it is always more fun and fruitful when you actually practice what you are reading (try examples from the text; think about and implement other cool things you can do based on what you just learned.)

Task 1: Build your simple calculator parser

In the lectures, you learned about basic algorithms to manually build lexical analyzer and parser for a compiler. As we briefly discussed, there is also various tools to help you build new compilers by automatically generating the code for (parts of) the new compiler. One approach to build a compiler for your new language is to build a new frontend for GNU Compiler Collection (gcc). In this exercise, you learn how to use two basic tools provided by gcc for creating your own lexical analyzers and parsers.

First, carefully read the first three sections of the GCC Frontend HOWTO to learn the basics of building lexers and parsers. The first complete example in this tutorial is to build a simple calculator capable of " `+-*/` " operations. Your task is very simple. Extend the example in the tutorial by including parentheses in your grammar. For example, your new parser should able to resolve that `(12+4)*5` will be `80`.

Your executable should be named `calc`. The parser as given in the example will accept input from stdin, prints the parsed value, and waits for next input (you perhaps need to manually terminate the program using `ctrl-c`). In order to test your program you can either use it in the interactive mode or pipe your input to it. The latter is how we will test your program. For example:

```
echo (12+4)*5 | calc
```

The contents of your project should be

- `calc.l` : the input file to flex
- `calc.y` : the input file to bison
- `makefile` : at least the following targets
  - `calc` (default target): build the executable file `calc`
  - `lex.yy.o` : compiles `lex.yy.c` which is the result of running flex on your input
  - `calc.tab.o` : compiles `calc.tab.c` which is the result of bison on your input
  - `clean` : delete all but your input files and `makefile`

> **Hint:** I suggest that you start by running the tutorial example. But before that, make sure you have at least read the document once. Refer back to the document and manual pages to understand the details better and implement the requested feature.