Workshop: Programming Languages

October 9, 2024

1 Introduction

In this workshop, you will be asked to familiarise yourself with a new programming language, learn its basics, history and use cases, and present the results to your peers. Your task is threefold:

- Write a program in your language
- Write a paper about your language
- Give a presentation on your language

Finally, there will be a (completely non-scientific) comparison of the performance of all the programs developed. This is not relevant for real development, but should give further insight into the differences between all languages. Of course, this performance comparison is irrelevant for grading.

2 Write a program in your language

To experience the development of your language in practice, you need to implement a simple program that recursively searches for a regular expression in all files of a folder (or just a single file). This program should behave similarly (though by no means completely) to the popular `grep` utility. The program must be able to run on a Linux system in an ANSI compatible terminal (and is not *required* to run on other platforms).

Given a pattern and a path, the program should recursively search the given path (which could be a single file or a folder) for the pattern. If found, it must print the path to the file, the line in the file, and the line itself where the pattern was found. It must also have the option to colour the terminal output, highlighting the text found in the line. Example output is shown below.

When you have finished implementing the program, depending on how difficult the implementation was, you should spend some time profiling your program. To do this, you should look at the performance profilers for the language (if they exist) and think about what can be done to speed up the search, given the features and limitations of the language.

Some parts of this specification are deliberately vague. It is part of your task to research how they could be solved.

Specification The program must meet the following requirements:

- it adheres to the command line interface given below
- it must also be able to print a similar help message
- it is able to search files with UTF-8 encodings (but no other encodings are required)
- it is able to exclude binary files from the search
- it passes the test-suite (see below)

Command line interface and sample output The command line interface should be able to support the following options:

```
usage: searcher [OPTIONS] PATTERN [PATH ...]
 -A,--after-context <arg>    prints the given number of following lines
                             for each match
 -B,--before-context <arg>   prints the given number of preceding lines
                             for each match
```

```
 -c,--color                 print with colors, highlighting the matched
                            phrase in the output
 -C,--context <arg>         prints the number of preceding and following
                            lines for each match. this is equivalent to
                            setting --before-context and --after-context
 -h,--hidden                search hidden files and folders
    --help                  print this message
 -i,--ignore-case           search case insensitive
    --no-heading            prints a single line including the filename
                            for each match, instead of grouping matches
                            by file
```

The exact syntax is not too important if your language (or rather the library you are using to parse command line arguments) does not support that particular syntax (or feature set), but the provided test suite will then need to be tweaked a bit.

The following snippets are (partial) outputs of the program. Of course, the colours do not have to match exactly.

```
~ $ searcher --color PM_RESUME test_data/linux
test_data/linux/include/uapi/linux/apm_bios.h
89:#define APM_RESUME_DISABLED  0x0d
test_data/linux/drivers/usb/mtu3/mtu3_hw_regs.h
429:#define LPM_RESUME_INTR   BIT(9)
test_data/linux/drivers/net/wwan/t7xx/t7xx_reg.h
93:#define T7XX_PCIE_PM_RESUME_STATE    0x0d0c
96: PM_RESUME_REG_STATE_L3,
97: PM_RESUME_REG_STATE_L1,
98: PM_RESUME_REG_STATE_INIT,
99: PM_RESUME_REG_STATE_EXP,
100:  PM_RESUME_REG_STATE_L2,
101:  PM_RESUME_REG_STATE_L2_EXP,
...

~ $ searcher --no-heading --color PM_RESUME test_data/linux
test_data/linux/include/uapi/linux/apm_bios.h:89:#define APM_RESUME_DISABLED    0x0d
test_data/linux/drivers/usb/mtu3/mtu3_hw_regs.h:429:#define LPM_RESUME_INTR     BIT(9)
test_data/linux/drivers/net/wwan/t7xx/t7xx_pci.c:59:    MTK_PM_RESUMED,
test_data/linux/drivers/net/wwan/t7xx/t7xx_reg.h:93:#define T7XX_PCIE_PM_RESUME_STATE       0x0d0c
test_data/linux/drivers/net/wwan/t7xx/t7xx_pci.c:124:   atomic_set(&t7xx_dev->md_pm_state, MTK_PM_RESUMED);
...

~ $ searcher --no-heading -C 5 PM_RESUME test_data/linux
...
test_data/linux/Documentation/dev-tools/sparse.rst-20-
test_data/linux/Documentation/dev-tools/sparse.rst-21-        typedef int __bitwise pm_request_t;
test_data/linux/Documentation/dev-tools/sparse.rst-22-
test_data/linux/Documentation/dev-tools/sparse.rst-23-        enum pm_request {
test_data/linux/Documentation/dev-tools/sparse.rst-24-                PM_SUSPEND = (__force pm_request_t) 1,
test_data/linux/Documentation/dev-tools/sparse.rst:25:                PM_RESUME = (__force pm_request_t) 2
test_data/linux/Documentation/dev-tools/sparse.rst-26-        };
test_data/linux/Documentation/dev-tools/sparse.rst-27-
test_data/linux/Documentation/dev-tools/sparse.rst:28:which makes PM_SUSPEND and PM_RESUME "bitwise" integers...
test_data/linux/Documentation/dev-tools/sparse.rst-29-there because sparse will complain about casting to/from...
test_data/linux/Documentation/dev-tools/sparse.rst-30-but in this case we really _do_ want to force the conversion)...
test_data/linux/Documentation/dev-tools/sparse.rst-31-the enum values are all the same type, now "enum pm_request"...
test_data/linux/Documentation/dev-tools/sparse.rst-32-type too.
test_data/linux/Documentation/dev-tools/sparse.rst-33-
--
test_data/linux/Documentation/dev-tools/sparse.rst-40-So the simpler way is to just do::
test_data/linux/Documentation/dev-tools/sparse.rst-41-
test_data/linux/Documentation/dev-tools/sparse.rst-42-        typedef int __bitwise pm_request_t;
test_data/linux/Documentation/dev-tools/sparse.rst-43-
test_data/linux/Documentation/dev-tools/sparse.rst-44-        #define PM_SUSPEND ((__force pm_request_t) 1)
```

```
test_data/linux/Documentation/dev-tools/sparse.rst:45:        #define PM_RESUME ((__force pm_request_t) 2)
test_data/linux/Documentation/dev-tools/sparse.rst-46-
test_data/linux/Documentation/dev-tools/sparse.rst-47-and you now have all the infrastructure needed...
test_data/linux/Documentation/dev-tools/sparse.rst-48-
test_data/linux/Documentation/dev-tools/sparse.rst-49-One small note: the constant integer "0" is special. You...
test_data/linux/Documentation/dev-tools/sparse.rst-50-constant zero as a bitwise integer type without sparse...
...
```

(the `...` was added everywhere the output did not fit the page). Note how in the snippet, context lines (lines that do not have a match) are denoted by a `-` around the line number, and a `--` is inserted between matches, if those matches are not in the same file (not shown here) or if there are omitted lines between the contexts. Also, when two matches and their context lines overlap, no line is printed twice. Instead the two matches are printed in one block (as seen in on lines 24 and 28).

These are only snippets and do not show full outputs. Full output examples will be provided for reference. These will not include colouring (because they are text files), but of course colouring should always be an option. In addition to the sample outputs, there is a "readme" file which gives some additional information about how the output should be structured.

Test suite You will be provided with a test suit to test your program. This includes a Python script and about 20GB of test data. The test suite can be used to automatically test your program against a reference implementation (i.e. `grep` or the much faster `ripgrep`). To run the test suite, use the following command (Python 3 must be available on your system):

```
$ python test.py -d <data-path> <command>
```

where `<data-path>` is the path to the test data (containing the `linux` folder and `subtitles.txt`) and `<command>` is the command used to call your program (this can be a single command, i.e. `./searcher`, or a more complex call, i.e. `java -jar target/searcher.jar`). Use `python test.py -h` to see some additional options for the test script. Although not necessary, it is much faster to use `ripgrep` as a reference implementation, so you should look at how to get it installed on your system.

3 Write a paper about your language

For your paper, you should write 5000 words about the language. This should include its history and development, its use cases and its design, but should not include a "tutorial" on the language or a long-winded explanation of its syntax. It must also include an account of how you developed your program, what aspects of the language proved useful, what libraries you used, and anything else you think is important that might not be important in other languages. Depending on how difficult it was to develop the program in your particular language, it should also include an account of how you optimised your program to be faster and/or more efficient.

The following questions (in no particular order) can be used as a guide for the paper, although they should not be considered as a checklist:

- when/why and by whom was the language developed?
  - this may include talking about what came before and what was missing
  - it might also be interesting to talk about how the language is developed today (i.e. is there a committee or foundation in charge, who contributes to it, and how is it organized)
- what niche is the language supposed to fill?
- what makes the language special or different? (at least at the time it was developed)
- where is the language used today?
  - this may include a short exposition of one or more prominent projects where the language is used
- how does the infrastructure around the language work? (build systems, package management, documentation, IDEs, etc.)
- what characterizes the language? (type system/compilation vs. interpretation/memory management/etc.)

– although no explanation of its syntax is expected, if your language has a particularly interesting or different syntax to other languages, you may include a short paragraph about it

The papers contents contain roughly

- 60% about the language
- 30% about the implementation
- 10% for the introduction and conclusion

4 Give a presentation on your language

At the end of the workshop, you should give a short presentation about your language. This presentation must be no longer than 20 minutes and should give a brief overview of the language itself. Similar to your paper, it should not be a "tutorial" or an explanation of its syntax, but should instead highlight its design, its strengths and weaknesses, its use cases (including perhaps some mentions of prominent projects) and its infrastructure. It should also highlight some particularly interesting aspect of the implementation of your program. After the 20 minute presentation, there will be another 10 minutes for questions.