Functional Programming

http://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2017/

Programming project – Grep in Haskell

February 14, 2018

`grep` is a UNIX utility that prints all the lines matching a given pattern in a set of files. Its arguments are a pattern (regular expression) followed by a list of files. Your task is to implement a (simple) `grep` clone in Haskell, `hgrep`, including a regular expression matcher, file system accesses, and a command line interface.

```
% hgrep "x[0-9]+" file1 file2 file3
```

1 Project

This project should be worked on by groups of two (or exceptionally three) students. The implementation should contain a file `Readme` describing how to compile and use the program. For the final version, each group should send a file "project-FP-<NAMES>.tar" to both Prof. Peter Thiemann and Gabriel Radanne by email. The email should be titled "project FP <NAMES>". The deadline for the project is the **February 19, 2018**.

The grade of the project will be used to improve the grade of the exam. Additionally, one of the exercises of the exam will refer to the project.

This project consists of two parts, a core section that all groups should complete and a set of potential extensions. Each group should pick at least two extensions.

2 Core

Each group should implement a fully-functional executable that takes as argument a regular expression using the POSIX syntax, a list of files, and prints all the lines matching the regular expression to the standard output. The implementation of regular expression matching should follow the description provided in section 2.1. The implementation of regular expression matching should also be testable stand alone, as described by section 2.2.

2.1 Regular expressions

For this project, we implement a simplified version of Posix Regular Expressions which we describe in fig. 1. We write ε for the regular expression matching the empty string and $\emptyset$ for the regular expression that does not match any string. For ease of reading, we write the sequence operation as $a \cdot b$, although the concrete syntax is simply the concatenation (ab, in this case). The alphabet, which is the set of printable ASCII characters, is noted $\mathcal{A}$.

We also present a grammar for regular expressions in fig. 2. Special characters such as "*", "|", "(" and ")" must be escaped to be used as regular characters. This syntax corresponds to `grep`'s "Extended Regular Expressions".

2.1.1 Derivatives

In order to decide if a string matches a given pattern, we will use regular expression derivatives. The derivative or a regular expressions r on a character 'a' is noted $\partial_{\mathrm{a}}(r)$ which is the regular expression for the language after matching 'a'. For example, given the regular expression $(a|b) \cdot c*$, if we match the char 'a', we can only match $c*$ afterwards hence $\partial_{\mathrm{a}}((a|b) \cdot c*) = c*$.

We first define the nullable operation, in fig. 3. nullable(r) is true if and only if the regular expression r matches the empty string ε. For example, $(a|b) \cdot c*$ is not nullable, but $(a|b*)$ is. Using this operation, we define the derivative of a regular expression in fig. 4.

Name	Example	Example matched
Void	$\emptyset$	
Empty	ε	`""`
Atom	a	`"a"`
Alternative	$a\|b$	`"a", "b"`
Sequence	$a \cdot b$	`"ab"`
Repetition	$(a\|b)*$	`"", "a", "abbabba", ...`

Figure 1: Regular expression constructions

$$
\begin{aligned}
\langle re0 \rangle &::= \langle re1 \rangle \text{ ('|' } \langle re1 \rangle \text{)*} \\
\langle re1 \rangle &::= \langle re2 \rangle + \\
\langle re2 \rangle &::= \langle atom \rangle \ \langle postfix \rangle \\
\langle postfix \rangle &::= \text{'*'} \mid \varepsilon \\
\langle atom \rangle &::= \mathcal{A} \mid \text{'('} \ \langle re0 \rangle \ \text{')'}
\end{aligned}
$$

Figure 2: Grammar of regular expressions

$$
\begin{aligned}
\text{nullable}(\emptyset) &= \text{false} \\
\text{nullable}(\varepsilon) &= \text{true} \\
\text{nullable}('a') &= \text{false} \\
\text{nullable}(r|r') &= \text{nullable}(r) \,|\, \text{nullable}(r') \\
\text{nullable}(r \cdot r') &= \text{nullable}(r) \,\&\, \text{nullable}(r') \\
\text{nullable}(r*) &= \text{true}
\end{aligned}
$$

Figure 3: Nullability of regular expressions

$$
\begin{aligned}
\partial_c(\emptyset) &= \emptyset \\
\partial_c(\varepsilon) &= \emptyset \\
\partial_c(c') &= \begin{cases} \varepsilon & \text{if } c = c' \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_c(r \,|\, r') &= \partial_c(r) \,|\, \partial_c(r') \\
\partial_c(r \cdot r') &= \begin{cases} (\partial_c(r) \cdot r') \,|\, \partial_c\, r' & \text{if nullable}(r) \\ \partial_c(r) \cdot r' & \text{otherwise} \end{cases} \\
\partial_c(r*) &= \partial_c(r) \cdot r*
\end{aligned}
$$

Figure 4: Derivative of regular expressions

Given these two definitions, we can finally define the matching operation. Matching of a regular expression r with a string s, noted $r \sim s$, is defined as follow.

$$r \sim \varepsilon \iff \text{nullable}(r) \qquad\qquad r \sim c \cdot s \iff \partial_c(r) \sim s$$

For more details on regular expression derivatives, you can consult Owens et al. [2009].

2.2 Tests

A test library is available on the course's page as the file `RegexTestLib.hs` to test your implementation. This test library provides a function `withFeatures` that take as argument the list of features that your library implements and generating test cases using QuickCheck. Here is a sample of a main file using the test library.

```
import Regex
import RegexTestLib as Test

basics = Test.Basics mkAtom mkSeq mkAlt mkStar -- Core regex library
features = -- Choose your supported feature
    [ Test.Match Regex.match
    , Test.Set mkSet
    , Test.Rep mkRep
    -- , Test.And mkAnd
    , Test.Parsing show parse
    , Test.Any mkAny
    , Test.Many mkPlus
    -- , Simplify simplify
    ]

main = Test.withFeatures basics features
```

In order to compile this file, you need to add the library "QuickCheck >= 2.9.2" to your cabal file. You can find a more detailed documentation at the beginning of the file.

3 Extensions

Here is a list of extensions to the basic `grep` implementation highlighted above, sorted by approximate difficulty. Each group should implement at least two extensions in this list. In additions to the two extensions, you are also free to create new extensions of your own invention. You can consult the `grep` manual (accessible with `man grep`) for inspiration.

3.1 Common regex operations

The `grep` executable implements more operations than the one described in section 2. Here is a list of very common regular expression operators.

Common operators The regular expression "`.`" matches any character. The regular expression `r+` matches at least one `r` and `r?` matches either `r` or ε.

Generalized repetition The regular expression `r{n,m}` matches `r` n to m times, for n and m non-negative integers.

Character set and classes The regex `[anbcd2]` matches any character 'a', 'b', 'c', 'd' or '2'. The complement of a set can be taken by prepending `^` as in `[^anbcd2]`.

You can also implement some of the regex operations supported by `grep`.

3.2 Uncommon regex operations

Here are some less common regex operators.

Intersection and complement The regular expression `r&r'` matches *both* `r` and `r'`. The regular expression `r!` matches anything except `r`.

Boundaries Most regular expression engines allow so-called "boundary" operators. For example `^` and `$` match the empty string at the very beginning and very end of the line. `\b` matches the empty string that is at the beginning or the end of a word.

3.3 Feature-full command line interface

`grep` has numerous options to alter its behavior such as `-invert-match`, which only show unmatched lines, or `-ignore-case` which ignore case differences. Implement a command line interface (including documentation!) for your `grep` clone using the optparse-applicative package[1]. (Using this package is mandatory).

3.4 Search with a fixed string

`grep` features the `-fixed-strings` options, which searches for a fixed string instead of a regular expression. For efficiency, it uses a dedicated algorithm for string search such as the Boyer-Moore algorithm. Implement a dedicated string search algorithm.

3.5 Match highlighting

`grep` can highlight the section of line that was matched by the pattern. Extended the matching algorithm to remember which positions were matched.

[1] https://hackage.haskell.org/package/optparse-applicative

3.6 Faster matching

The algorithm outlined in section 2.1.1 is not very fast. It is possible to speed it up using two methods.

Simplify regular expressions For example, $\emptyset|r$ is the same as r. While this pattern will not appear in practice, it might appear in derivatives. There are many such patterns that can be simplified.

Caching derivatives While matching, the same derivative will be computed many times. Instead of deriving the same regular expression again and again, we can simply store the derivative once it has been computed.

References

Scott Owens, John H. Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, 2009. doi: 10.1017/S0956796808007090. URL https://doi.org/10.1017/S0956796808007090.