Homework 3

Functional and Parallel Programming (Term III/2020)

built on 2021/06/30 at 10:14:32 *__due:__ Monday July 26 @ 11:59pm*

This part of the assignment will go over the content we learned from Scala.

IMPORTANT: Your implementation must strictly follow the functional style. As a rule of thumb, you cannot use features other than what we have done in class so far. In particular, **this means, no loops, no mutable variables** (cannot use `var`).

- You can define as many helper functions as necessary. Be mindful of what you should expose to outside your function.
- You are going to be graded on style as well as on correctness.
- Test your code!

Ham, Spam, Spaghetti (8 points)

For this task, save your code in `Spaghetti.scala`

This problem involves working with lazy evaluation and streams. Your implementation for this problem will go inside |object Spaghetti|.

(a) The spaghetti sequence is the sequence of integers: 1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ... The sequence starts with the number 1. To generate next element, we read off the digits of the previous member, counting the number of digits in groups of the same digit, like so: 1 is read "one 1" giving rise to 11. Then, 11 is read "two 1" giving rise to 21. Then, 21 is read "one 2 one 1", i.e., 1211. Next, 1211 is read "one 1 one 2 two 1", i.e., 111221.

You'll implement `def spaghetti: Stream[String]` that gives a stream of the spaghetti sequence where each element is represented as a string.

(b) An n-bit ham code is a sequence of n-bit strings constructed according to certain rules. For example,

`n = 1:` $H(1)=$`"0", "1"`
`n = 2:` $H(2)=$`"00", "01", "11", "10"`
`n = 3:` $H(3)=$`"000", "001", "011", "010", "110","111", "101", "100"`

Your task: find a pattern in the example and come up with the construction rules. Then, implement `def ham: Stream[String]` that gives a stream listing Ham codes for $n = 1, 2, 3, \ldots$. That is, the `ham` stream starting with `"0"`, `"1"`, `"00"`, `"01"`, `"11"`, `"10"`, `"000"`, `"001"`, `"011"`, `"010"`, `"110"`,`"111"`, `"101"`, `"100"`, ...

(Hint: $H(n)$ can be defined recursively using $H(n-1)$ and operations involving prepending a digit to the string and reversing a sequence.)

Parallel Prefix Sum (4 points)

For this task, save your code in `prefixsum.rs`

In class, we saw how the `filter` operation can be accomplished in $O(n)$ work and $O(\log n)$ depth, through the use of parallel prefix sum. But how does one really implement prefix sum? In this problem, you are to use Rayon to implement a function

```
fn parallel_prefix_sum(xs: &Vec<u64>) -> Vec<u64>
```

that computes the prefix sum of `xs` in $O(n)$ work and $O(\log n)$ depth, where n is the input length. Examples:

```
parallel_prefix_sum(&vec![1,3,5,2]) == vec![0, 1, 4, 9, 11]
parallel_prefix_sum(&vec![3]) == vec![0, 3]
parallel_prefix_sum(&vec![]) == vec![0]
```

You should assume that running a `map` on a parallel iterator takes $O(n)$ work and $O(1)$ depth. It is important to handle the case where n is odd.

Mining A Hay Stack (4 points)

For this task, save your code in `mining.rs`

You're given a data file containing all flight records for one year. An example for year 2008 is available for download at

`https://cs.muic.mahidol.ac.th/~ktangwon/2008.csv.zip`

This file has been zip compressed, so you should uncompress it before use. The input file (e.g., `2008.csv`) is comma-separated and the first line contains the field headers. The fields are explained here `http://stat-computing.org/dataexpo/2009/the-data.html`.

You will write a function that will give us a glimpse into this rich dataset. Implement a function

```
fn ontime_rank(filename: &str) -> Vec<(String, f64)>
```

that takes in a filename (e.g., `2008.csv`) and returns a vector of (airline code, ontime percentage), sorted from best to worst ontime percentage. A flight is ontime if the `ArrDelay` field is a negative number, 0, or not a number. The field `UniqueCarrier` provides the airline code. The ontime percentage is calculated as the percentage of ontime flights compared to all flights using that carrier code. Your percentage number is a number between 0 and 100 (inclusive).

For your reference, a standard sequential Rust program can go through the example file in about 5 seconds, with the file residing on a decent SSD drive.

Your goal is **use some form of parallelism** to make this computation fast. For starters, several places can be parallelized:

- Parsing each line takes time. Parsing several lines at once will save time. See how Rayon parallel map may be useful.
- Computing the on-time statistics is similiar to computing a histogram. This, too, can be parallelized. There are at least two options to try:
  - Option 1: Use a parallel sorting routine and work from there, or
  - Option 2: Use a concurrent hash map so you can do "group by" in parallel. There is a crate called `chashmap` that you might want to look into.

There will be a timing competition for (extra) points.

To get started, here's a piece of code that reads a given file entirely and derives a vector of `&str` lines:

```
use std::fs::File;
use std::io::prelude::*;

let header_count = 1;
let mut f = File::open(filename).unwrap();
let mut whole_file = String::new();
f.read_to_string(&mut whole_file);

// skip the header line
let all_lines: Vec<&str> = whole_file.lines().skip(1).collect();
```

Crawling the Web (12 points)

For this question, save your file as: **crawler.rs**

A few terms ago, you were tasked to write a crawler in Java. In this problem, you're going to do the same in Rust, with an additional feature that many pages will be crawled in parallel.

In this problem, you will download an entire Java docs by crawling the URL here: `https://cs.muic.mahidol.ac.th/courses/ooc/docs/`

Make sure you do not crawl any links outside this domain—otherwise, you're practically crawling a large portion of the whole Internet. At the end of it, you will display:

- the total number of files.

- the total number of directory
- the total number of unique file extensions.
- the list all unique file extensions. Do not list duplicates.
- the total number of files for each extension.
- the total number of words in all html files combined, excluding all html tags, attributes and html comments.

Several Tips and Tricks:

- There are crates such as `request` and `select`, which can help you with HTTP downloading and HTML parsing.
- There is a crate called `url` that can help sanitize/normalize URLs.
- You'll have to keep track of what has been visited already so you don't crawl it again. Much of the logic here will be similar to writing breadth-first serach.
- You will want to use a threadpool. Your goal is to minimize overhead while allowing multiple pages to be crawled simultaneously. Here's a quick demo of using Rayon's threadpool implementation:

```
let num_threads = 4usize;
let pool = rayon::ThreadPoolBuilder::new()
  .num_threads(num_threads)
  .build()?;

let total_tasks = 30;

pool.scope(|scope| {
  for task_number in 0..total_tasks {

    scope.spawn(move |_| {
      let my_task_number = task_number;
      let mut rng = rand::thread_rng();
      let random_duration: u64 = rng.gen_range(50, 1000);
      println!("[x] Starting Task {}", my_task_number);
      std::thread::sleep(Duration::from_millis(random_duration));
      println!("[x] Done task {}", my_task_number);
    });
  }
});
```

Basic Equation Solver (12 points)

You're building a numerical equation solver. The interface is primitive, providing

```
def solve(expString: String, varName: String, guess: Double): Double
```

where, for example, $solve("x^2-4.0", "x", 1.0)$ will solve for x in $x^2-4=0$ with a starting guess of $x=1.0$, and will eventually return 2.0 (or -2.0 depending on which answer your Newton's method likes more). For full credit, your submission will provide a functioning solver in $solver.Main.solve$ with the above signature that implements the Newton's method. Additionally, it will provide functions `differentiate` and `eval`, inside `solver.Process`, with the signature

```
def differentiate(e: Expression, varName: String): Expression
def eval(e: Expression, varAssn: Map[String, Double]): Double
```

You are to ***assume that the input expression always has at least a root.***

Parser: To help you get started, we're supplying a parser to turn a string expression into a recursive data type +Expression+ for convenient manipulation and consumption. Details can be found inside `Parser.scala`. For this assignment, however, just about the only thing you need to know is that if there's a string s that looks like an expression ($|s = "2^x + x + 4 * 3"|$), you can turn it into an Expression value by calling `solver.Parser(s)` or simply `Parser(s)` if you're inside the solver package or have imported it. The following is an example code listing:

```
// assuming we're inside the solver package
val s = "2^x + x + 4*3"
val e: Expression = Parser(s)
```

Supporting Utility: There's a simple routine for pretty-printing, practically converting an Expression value into a string representation for debugging, etc. This lives inside $Process.scala$.

Also housed in the file are functions to be implemented (you're welcome to ditch this skeleton as long as both $differentiate$ and $eval$ are implemented and match the required type signature):

- `def eval(e: Expression, varAssn: Map[String, Double]): Double` evaluates a given expression `e: Expression` using the variable settings in `varAssn: Map[String, Double]`, returning the evaluation result as a `Double`.
  Example: $eval(e, Map("x" -> 4.0))$ evaluates the expression with the variable x set to 4.0.
- `def differentiate(e: Expression, varName: String): Expression` symbolically differentiates an expression `e: Expression` with respect to variable `varName: String`, returning an `Expression` value.
  (*Hint:* The `Expression` type is a sum type. Ponder what it can be.)
- `def simplify(e: Expression): Expression` forms a new expression that simplifies the given expression `e: Expression`. The goal of this function is to produce an expression that is easier to evaluate and/or differentiate. If there's a canonical form you'd like to follow, use this function to put an expression in that form. The idea is to use `simplify` on the differentiated expression so that it is a nicer expression.

Symbolic Differentiation:

Your `differentiate` function attempts to compute the (partial) derivative of `e` with respect to a given variable name `varName`. For example, if `varName` is x, then `differentiate` returns an `Expression` that is equivalent to $\frac{d\mathtt{e}}{dx}$. Do not worry: You really don't have to remember much calculus. Here are some formulas for computing derivatives that you will use:

$$\begin{aligned}
\frac{d}{dx}\text{constant} &= 0\\
\frac{d}{dx}\big(f(x)+g(x)\big) &= \frac{d}{dx}f(x)+\frac{d}{dx}g(x)\\
\frac{d}{dx}\big(f(x)\times g(x)\big) &= \frac{df(x)}{dx}\cdot g(x)+f(x)\cdot\frac{dg(x)}{dx}\\
\frac{d}{dx}\Big([f(x)]^h\Big) &= h[f(x)]^{h-1}\frac{df(x)}{dx} && \text{where } h \text{ contains no variable.}\\
\frac{d}{dx}c^{f(x)} &= [\ln(c)]\cdot c^{f(x)} && \text{where } c \text{ is a nonzero constant.}
\end{aligned}$$

You don't need to support expressions beyond what can be differentiated using these formulas. Keep in mind that some expressions have to be "massaged" before they fit one of these forms.

Extra-Credit: Parsing Your Own Expression

After you're done with the functions above, your next task to write your own parser! Yes, that function that takes, e.g., `x^3.5 + 2*(5+x)` and returns a nice `Expression`, similar to what `solver.Parser` does. More precisely, you'll write the function

```
        def parse(input: String): Option[Expression]
```

which lives inside `MyParser.scala`.

Inside the same file, there is a tokenizer function that you may find helpful.

Notes

You should have an implementation of the Newton's method already. You can adapt your own code from lecture. All the restrictions for Scala apply to this question.