# COMP 212 Fall 2024
# Final Non-Collaborative Challenge Problems

Remember that non-collaborative challenge problems are to be done independently. You are not allowed to communicate with anyone about the problems, and in particular are not allowed to discuss the problems with other students. Additionally, you are not allowed to search for help on the specific problem from any sources besides the course materials, including the Web or any generative AI tools. **There will be a zero-tolerance policy for violations. Anyone suspected of violating the honor code will be referred to the honor board, with a recommended penalty of a grade of F in COMP 212 if found responsible. If you can't solve a problem on your own, leave it blank.**

## 1 Text Generation

The code for this problem is in `generate.sml` and `sources-generate.cm`. Put these inside your `hw10-handout` folder because the problem also uses the support code from that homework.

In this problem, you will write a simple text generator, which takes a "prompt" and outputs a continuation of that prompt.

For example:

```
val g = Generate.make_generator
          (Generate.open_file_no_label "data/RCV1.medium_train.txt");

val test = Seq.tolist (Generate.generate (g, 7,
                                          SeqUtils.words "The"));
Might see: val test = ["The","company","said","the","first","quarter","of","the"]

val test2 = Seq.tolist (Generate.generate (g, 4,
                                           SeqUtils.words "Several players went"));
Might see: val test2 = ["Several","players","went","to","the","first","quarter"]
```

The first line trains a generator based on the Medium training set articles from Homework 10 (ignoring the category labels). `test` uses the generator to generate the next 7 words after "The". `test2` uses the generator to generate the next 4 words after "Several players went".

**Task 1.1** (45 pts). Write a function

```
make_generator : document MR.mapreducable -> (string Seq.seq -> string)
```

that takes a training dataset and outputs a function that, when given the sequence of words in a prompt, generates a next word.

You can follow the same general outline as in Homework 10: gather some statistiics from the training data, and then use those statistics to estimate the probability of the possible next words that the generator might generate, and then pick a word according to that probability.

For the first step, use the pairs of words that occur one after the other in the training data to estimate the probability of the next word after a given a word. (For extra credit, use more than a single word to pick the next word.)

For the second step, use the training data to determine what words might possibly follow a given word.

For the third step, as a baseline you can pick the word with the highest probability (like in Homework 10). But for full credit you should pick a word using some randomness. For example, if the possible next words after the word "frog" are "jumped" with probability 0.75 and "sat" with probability 0.25, your generator should pick "jumped" 3/4 of the time and "sat" 1/4 of the time. This way, less common words sometimes get picked too. You can use the function `randomReal()` to generate a random number in the range 0.0 though 1.0. Each time you call the function, it produces a different number. The numbers produced will be different every time you compile the program.

Upload your `generate.sml` to your handin folder.

# 2 Regular Expression Matching

The code for regular expression matching from this past week's lectures is in `regexp.sml` linked from this assignment.

**Task 2.1** (10 pts). Fill in the case for `OneOrMore r` for `fastmatch`.

**Task 2.2** (20 pts). Do the cases for `Lit c` and `OneOrMore r` of the following proof.

**Theorem 1.** *For all regular expressions* `r`*, stacks* `k`*, and strings* `s`*,*

```
fastmatch(r,k,s)
=
exists(fn (p1,p2) => match(r,p1) andalso matches(k,p2), splits(s))
```

*Proof.* The proof is by lexicographic induction on `s` and then `r`. This means you are allowed inductive hypotheses whenever (1) for the recursive call, the string `s` is smaller than the input to the original function call, or (2) in the recursive call, the string `s` stays the same as the input but the regular expression `r` is a child tree of the input regular expression.

- Case for `Or(r1,r2)`: We want to show that

```
fastmatch(Or(r1,r2),k,s)
=
exists(fn (p1,p2) => match(Or(r1,r2),p1) andalso matches(k,p2),
                                                 splits(s))
```

On the left-hand side

```
    fastmatch(Or(r1,r2),k,s)
↦   fastmatch(r1,k,s) orelse fastmatch(r2,k,s)
=   exists(fn (p1,p2) => match(r1,p1) andalso matches(k,p2), splits(s))
    orelse
    exists(fn (p1,p2) => match(r2,p1) andalso matches(k,p2), splits(s))
```

by the inductive hypothesis on `(s,r1)` and `(s,r2)`, which is valid because `s` is the same length and `r1` and `r2` are smaller.

On the right-hand side

```
    exists(fn (p1,p2) => match(Or(r1,r2),p1) andalso matches(k,p2), splits(s))
=   exists(fn (p1,p2) => (match(r1,p1) orelse match(r2,p1))
     andalso matches(k,p2), splits(s))
```

Suppose the left-hand side of the equation evaluate to true. Then either

```
exists(fn (p1,p2) => match(r1,p1) andalso matches(k,p2), splits(s))
```

or

```
exists(fn (p1,p2) => match(r2,p1) andalso matches(k,p2), splits(s))
```

evaluates to true. In the first case, because the `exists` evaluates to true, there is some splitting of `s` for which `match(r1,p1)` and `matches(k,p2)` evaluate to true. Therefore, there is some splitting for which

```
(match(r1,p1) orelse match(r2,p1)) andalso matches(k,p2)
```

evaluates to true, and the right-hand side is true as well. In the second case, there is some splitting of `s` for which `match(r2,p1)` and `matches(k,p2)`, so the right-hand side similarly evaluates to true.

Conversely, assume the right-hand side of the equation evaluates to true. Then there is some splitting of `s` for which

```
(match(r1,p1) orelse match(r2,p1)) andalso matches(k,p2)
```

evaluates to true. For this expression to be true, at least one of `match(r1,p1)` or `match(r2,p1)` must evaluate to true. When `match(r1,p1)` is true, we have that `(match(r1,p1) andalso matches(k,p2))` is true, and so is

```
exists(fn (p1,p2) => match(r1,p1) andalso matches(k,p2), splits(s))
```

When `match(r2,p1)` is true, so is

```
exists(fn (p1,p2) => match(r2,p1) andalso matches(k,p2), splits(s))
```

So in either case, the left-hand side is true.

- Case for `Then(r1,r2)`: We want to show that

```
fastmatch(Then(r1,r2),k,s)
=
exists(fn (q1,q2) => match(Then(r1,r2),q1) andalso matches(k,q2),
                     splits(s))
```

On the left-hand side

```
  fastmatch(Then(r1,r2),k,s)
↦ fastmatch(r1,r2::k,s)
= exists(fn (p1,p2) => match(r1,p1) andalso matches(r2::k,p2), splits(s))
```

by the inductive hypothesis on `(s,r1)`, which is valid because `s` is the same length and `r1` and `r2` are smaller. By definition of `matches`,

```
  matches(r2::k,p2)
↦ exists(fn (p21,p22) => match(r2,p21) andalso matches(k,p22), splits(p2))
```

So, all told, there exists a splitting of `s` into `p1` followed by `p21` followed by `p22` where `match(r1,p1)` and `match(r2,p21)` and `matches(k,p22)`.

On the right-hand side

```
  match(Then(r1,r2),q1)
↦ exists (fn (q11,q12) => match(r1,q11) andalso match(r2,q12), splits(q1))
```

So, all told, there exists a splitting of `s` into `q11` followed by `q12` followed by `q2` with `match(r1,q11)` and `match(r2,q12)` and `matches(k,q2)`.

Identifying `p1 = q11` and `p21 = q12` and `p22 = q2`, these are the same conditions.

(We are being informal about associativity of splitting here: if `(p1,p2)` is in `splits(s)` and `(p11,p12)` is in `splits(p1)`, then some `(p11, q)` is in `splits(s)` with `(p12,p2)` in `splits(q)`; and similarly for `p2`.)

- Case for `Lit c`: **Do this case.**
- Case for `OneOrMore r`: **Do this case.**

□

Upload `regexp.sml` and `final.pdf` with the answers to these tasks.