CSCI 239—Discrete Structures of Computer Science
Lab 10—Pattern Matching

In logical proofs, we use pattern matching to apply equivalence and inference rules to logical statements. In this lab, we will explore two powerful ways to use pattern matching in computing applications: regular expressions with the *egrep* program and patterns in Haskell function definitions.

Objectives:

- to learn to form simple, useful regular expressions
- to explore how to use simple regular expressions to find patterns in text files
- to use pattern matching to define a Haskell function

Part 1: Regular expressions in the Linux shell

Regular expressions are a shorthand way of representing patterns that could occur in a string of characters. Linux shell programs have a built-in form of regular expressions. For example, suppose you had a set of strings and you wanted to know which strings had double *a*s (*aa*) in them. You could do this with the pattern:

```
.*aa.*
```

In this regular expression, a period (`.`) is used to represent any single character and an asterisk (`*`) represents zero or more repetitions of the preceding expression. The pattern `'.*'` occurs twice in the pattern above, where each time it represents zero or more repetitions of any character. The whole pattern `'.*aa.*'` then represents zero or more characters, followed by two *a*s, followed by zero or more additional characters, that is, as many characters as you want, as long as there are at least two consecutive *a*s. Here are some strings that match the pattern `'.*aa.*'`:

```
aa    abcaadef    1&,10aa1zen    aaPlusAnything    anythingPlusaa
```

Note that `*` can operate on any pattern. For example, `'a*'` means a string with zero or more *a*s; `'(abc)*'` means a string consisting of zero or more repetitions of the string *abc*. In the second example, the parentheses serve to group the pattern that the `*` operator is applied to. `'abc*'`, without the parentheses, means an *a*, a *b*, and then zero or more *c*s.

Let's try using the program *egrep*, which stands for *e*xtended *g*et *r*egular *exp*ression, to evaluate some regular expressions. We'll test our expressions using the file `/usr/share/dict/linux.words`, which is a file containing about half a million English words and Linux terms.

Open a Linux terminal window and type the command:

```
egrep '.*aa.*' /usr/share/dict/linux.words
```

You should see a list of all the words in the file containing at least one pair of consecutive as. *egrep* is a Linux program that takes two arguments; the first is a string representing a regular expression enclosed in single quotes; the second is the name of a text file. The program searches the file, looking for the pattern in each line; it outputs those lines that contain the pattern. If the pattern occurs more than once in the line, it only outputs the line once.

Our test file contains one word per line, so the output is all the words matching the pattern. If we use the program with a file containing more than one word per line, the output will be all lines that have a substring matching the pattern.

Note that the `'.*'` at the beginning and the one at the end of the pattern are actually unnecessary. If you type `egrep 'aa' /usr/share/dict/linux.words`, you will get the same result. This is because the *egrep* program looks for the pattern anywhere in the line rather than trying to match the pattern to the whole line. We'll see a bit further on that it is possible to match a pattern to a whole line as well.

Complete the following exercises using *egrep* on `/usr/share/dict/linux.words`. Record the *egrep* command you used for each part; only record a few lines of the output unless it fits in one window.

1. Find all words that have two *a*s somewhere in them, not necessarily next to each other.
2. Find all words that have two or more double *a*s in them. There must be at least one letter separating two double *a*s.
3. Find all words that have a *b* with a double *a* somewhere after the *b*.
4. Find all words that have an *a*, later a *b*, later a *c*, later a *d*, and still later an *e*.

There are many more ways to construct regular expressions for *egrep*; we'll look at a few of them here. To specify a selection among two or more alternative characters, we can use square brackets (`[]`). For example, `'[abcxq]'` represents a single letter that must be an *a*, a *b*, a *c*, an *x*, or a *q*; so the pattern `'.[abcxq].'` would specify a three-character string starting with any character followed by an *a*, *b*, *c*, *x*, or *q* followed by any one additional character. Alternatively, we can use square brackets to exclude characters by putting a caret character (`^`) right after the left bracket. For example, `'[^abc]'` represents a single character which can be any character *except a*, *b*, or *c*, and so the pattern `'[^abc]def'` would represent words that contain a character that is not an *a*, a *b*, or a *c* followed immediately be the string *def*.

If you want to make your pattern match only at the beginning of the line, you can do so by putting a caret (`^`) at the beginning of the pattern. (In this context, the caret has a different meaning than it does inside the square brackets, which is probably a bad decision on the part of the people who designed this version of regular expressions.) If you wanted to look for all the lines in a file that started with the string */#*, you could use the pattern `'^/#'`. Similarly, if you want your pattern to match only at the end of the line, end the pattern with the dollar sign (`$`), so the pattern `'[aeiou]$'` would match all lines ending in a lower-case vowel. Finally, you can match the whole line by putting a caret at the start of the pattern and a dollar sign at the end of the pattern; thus, if you use the pattern `'^a[^a]*a$'`, you would require that the line start with an *a*, end with another *a*, and that all the characters between these two *a*s (if any) not be *a*s.

Now do the next exercises, as before.

5. Find all words in `/usr/share/dict/linux.words` that have the letters *q* or *z* at least three times. (For example, *quizzes* has one *q* and two *z*s, so it should match the pattern.)
6. Find all words that have at least five vowels. (For this exercise, the vowels are *a*, *e*, *i*, *o*, *u*, and *y*.)
7. Find all words that have *exactly* four vowels. (This exercise is a little more complicated than the last.)

You can also specify ranges in the square bracket construction. For example, `[a-e]` means the same thing as `[abcde]`, and `[0-9A-Fa-f]` matches hexadecimal digits. The pattern `'[a-e][a-e]'` represents all lines having two of the letters from *a* to *e* next to each other.

8. Find all words in `/usr/share/dict/linux.words` that have a *b*, immediately followed by a letter from *a* to *q*, immediately followed by a letter from *r* to *z*.
9. Find all words that have a *c*, immediately followed by something that is not an *a*, immediately followed by an *r*.
10. Find all words that have a *t*, immediately followed by a letter that is not a vowel.

To choose among alternatives that are longer than a single character, put the alternatives in parentheses and separate them with the vertical bar character (`|`, which is above the enter key). For example, to match all lines containing *zz*, *bb*, or both, you could use the pattern `'(zz|bb)'`.

11. Find all words in `/usr/share/dict/linux.words` that have either *th* or *sc* followed by a non-vowel.
12. Find all words that have one or more of the properties in exercises 8–11. (This one requires nesting alternatives.)

This lab just gives a taste of the power of pattern matching using regular expressions. There are many applications for searching files for data that matches a desired pattern, and these applications usually use some form of regular expressions to increase the power of the algorithms.

Part 2: Pattern matching in Haskell function definitions

Another application of pattern matching is in programming languages, and (you guessed it) Haskell is a programming language that makes extensive use of pattern matching in the programs themselves. We've already seen a little of this, but we'll explore is a little further here.

Consider the following recursive definition of the *fibonacci* function:

$$fibonacci(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ fibonacci(n - 1) + fibonacci(n - 2) & n > 1 \end{cases}$$

Here is a Haskell function (also in the lab file) that implements the definition directly:

```
>    fibonacci :: (Eq a, Num a, Num a1) => a -> a1
>    fibonacci 0 = 1
>    fibonacci 1 = 1
>    fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

This function definition uses pattern matching on the parameters to define the base cases of the function.

Open *ghci* in a terminal window with the lab directory as the current directory. Try the *fibonacci* function with a few small values. Notice that as you increase the value of *n*, the function returns more and more slowly until you decide that you're tired of waiting for it. If that happens, you can use <*control*>–*Z* to kill the program. Why do you suppose this program runs so slowly?

Now consider another recursive function with a similar looking definition:

$$ackermann(m,n) = \begin{cases} n + 1 & m = 0 \\ ackermann(m - 1, 1) & m > 0, n = 0 \\ ackermann(m - 1, ackermann(m, n - 1)) & m > 0, n > 0 \end{cases}$$

Using the *fibonacci* function as a model, implement a Haskell version of this function. Test your function for the twenty-four cases where $0 \le m \le 3$ and $0 \le n \le 5$, and make a table of the values. You can check the Wikipedia page for *Ackermann function* to verify your table of results. Notice that this function gets enormously large for relatively small values of *m* and *n*. It also gets unacceptably slow for *ackermann* 3 9 and *ackermann* 4 1 and greater parameter values. Believe it or not, this function actually has value in the analysis of algorithms.