15-110: Principles of Computing

Recitation: Regular Expressions, FSMs, and File I/O (July 28, 2011)

In computer science, a **language** is defined to be a **set** of strings made up by characters from an **alphabet** Σ. A string s is determined to be in the language if they fit the membership requirements of the language. A **regular language** is a language where all the strings in the language match one or more given regular expressions. For example, the most trivial regular language is the language that contains only the empty string, ε. Another example of a regular language is the set of strings that match the regular expression `"Hello|World"`. Its members consist exactly of two words: `Hello` and `World`.

1. Defining and Proving Regular Languages

Problems related to regular languages are usually given as:

There exists a regular language L such that

$$L = \{x \mid x \text{ is a string of zero or more a's}\}$$

Prove that L is or is not regular.

To formally prove that a language is regular, one only needs to supply a regular expression that matches all of the strings in the language, or a FSM or DFA that matches the language. Sometimes it is easier to provide a regular expression to prove that a language is regular, sometimes it is easier to provide a FSM. In the above case, L is regular because the language can be defined by the regular expression `a*`.

Prove that the following languages are regular by supplying a regular expression that matches all strings in the language.

(a) $L = \{x \mid x$ **is made up of an even number of** `a`**'s**$\}$

(b) $L = \{x \mid x$ **starts and ends with two** `b`**'s**$\}$

(c) $L = \{x \mid x$ **is a binary number**$\}$**.**

(d) $L = \{x \mid x$ **is an English name that starts with a capital letter**$\}$

(e) $L = \{x \mid x$ **is a Roman numeral less than 9000**$\}$

(f) $L = \{x \mid x$ **is a bill item of the form** `Item: $DD.CC`**. e.g.** `Fried Rice: $123.45`$\}$

(g) Solve each of the above problems by supplying a FSM that matches the language.

2. Challenging Problems

(a) Prove that the language $L = \{x \mid x$ **is a binary representation of a non-negative multiple of 3**$\}$ **is regular. e.g.** `11`, `1001`, `100001`

(b) Prove that the language $L = \{ww \mid w$ **is any string**$\}$ **is not regular.**

(c) Write a regular expression that checks if an e-mail address is valid. Hint: Look for Mail::RFC822::Address

3. File Input/Output

In the computing world, files are an important part of everyday life. Programs read in data from files, and output computed results into other files. In some operating systems (e.g. UNIX), files are so important that everything (user input, Internet connections, etc.) is abstracted away as a collection of files! Thus, a necessary tool in a programmer's arsenal is the ability to work with files.

In Python, file I/O is done through **file objects**. When one opens a file in Python, they create the file object and can call several functions that will let the user read or write from the opened file.

When opening a file, one can supply Python with a **mode string**. This will tell Python what you want to do with the specific file. The modes available are:

- `r`: This mode opens the file in *read-only* mode. That is, the user will only be able to read from the file.
- `w`: This mode opens the file in *write* mode. That is, the user will be writing to the file. **WARNING**: If the file already exist, it will be erased.
- `a`: This mode opens the file in *write* mode but will not erase the file, and start writing at the *end* of the file.
- `r+`: This mode opens the file in both *read* and *write* mode. This lets the user read and write to the file at the same time. Useful for special files like databases.

Note: If you append a `b` to the mode string, e.g. `ab`, it opens the string in **binary mode**.

- `open(filename, mode)`: This function will open the file at the given mode. A successful call will return a file object.
- `close()`: This function will close the file object. It is good practice to close a file when one is done using it.
- `read(size)`: This function will read up to `size` bytes from the file and return them, and move the seek pointer by the appropriate number of bytes (meaning you have to set the seek pointer back to 0 if you want to read from the beginning of the file again). `size` is an optional argument so if you call `read()`, it will read and return all of the file's contents.
- `readline()`: This function reads and returns exactly one line of input from the file (up to and including the newline character `"\n"`). If it hits the end of the file and there is no newline character, it will not include the newline.
- `readlines(size)`: This function reads and returns all the lines in a files as a list of lines. `size` is an optional argument that when given, will make the function read up to `size` bytes in the file and return only the lines in those bytes. If a line is cut-off, it will return the whole line.
- `write(str)`: This function writes the string `str` onto the file starting from the current seek position.
- `seek(n)`: This function changes the seek point to the `n+1`'th byte in the file. e.g. `seek(0)` will move the seek pointer to the first byte in the file, `seek(1)` the second byte, and so on.
- `closed`: This value is a bool that will tell the user whether or not a file is closed.

(Example 1)

Let's try a simple example where we write out a list of ints from 0 to 999 to a file called `15110ints.txt`.

```
intList = range(0, 1000)
intFile = open("15110ints.txt", "w")
for myInt in intList:
    tempStr = str(myInt) + "\n"
    intList.write(tempStr)
intFile.close()
```

The above code block first converts a single int to a string (with an attached newline character) and then writes it to the file. Once it finished writing all of the ints to the file, it then closed the file.

(Example 2)

If we want to get the ints back from the file, we only have to do the following:

```
intFile = open("15110ints.txt", "r")
inputList = [ ]
for myInt in range(0, 1000):
    inputStr = intFile.readline()
    inputInt = int(inputStr)
    inputList += [inputInt]
intFile.close()
```

This opens the file, reads in a line, converts the string to an int, and then adds the int to a list, repeats the read-convert-add process up to a thousand times and then closes the file.

If we rewrite the code using the `readlines()` function, it will look like the following:

```
intFile = open("15110ints.txt", "r")
inputList = [ ]
for inputStr in intFile.readlines():
    inputInt = int(inputStr)
    inputList += [inputInt]
intFile.close()
```

The `readlines()` function is rather convenient when you have to read in a bunch of lines and do not know how many lines there are in the file.

Protips

```
inputlist = [ ]
with open("15110ints.txt", "r") as intFile:
    for inputStr in intFile.readlines():
        inputList += [int(inputStr)]
```

Python provides a neat feature where with the `with...as...` syntax, you would not have to worry about closing the file object outside of the block because Python does it automatically for you. The above code block is identical to the previous code block except that it uses the `with...as...` syntax.

Challenge: Go to `http://projecteuler.net/index.php?section=problems&id=22`, download `names.txt` and try solving the problem (ask a CA if your answer is correct).