Computing Acronyms From Strings

by Imre Horvath, 09/04/15

1. Introduction

I'd like to demonstrate how to write a procedure/function/method in various programming languages, using different styles, implementing the acronym function. The acronym function maps a string to a word created by combining of the first letters of the "real" words of the original string.

2. Code Examples

2.1 Java

2.1.1 Java - Imperative Style

```java
import java.util.List;
import java.util.Arrays;

public class AcronymImper {

  private static final List<String> UNWANTED = Arrays.asList("THE", "OF", "AT", "IN");

  public static String acronym(String str) {
    String up = str.toUpperCase();
    String[] words = up.split("\\s+");
    String result = "";

    for (String wd : words) {
      if (!UNWANTED.contains(wd)) {
        result += wd.substring(0, 1);
      }
    }

    return result;
  }

  public static void main(String[] args) {
    System.out.println(acronym(args[0]));
  }
}
```

Figure 1

This highly imperative implementation uses a for loop, assignment and blends all the different computations like filtering a collection, transforming elements, combining results into one monolithic chunk of code.

2.1.2 Java - Functional Approximation - Pre Java 8

```
import java.util.List;
import java.util.Arrays;

import util.FuncLib;
import util.Predicate;
import util.Function;
import util.Combiner;

public class AcronymFuncAppr {

  private static final List<String> UNWANTED = Arrays.asList("THE", "OF", "AT", "IN");

  public static String acronym(String str) {
    List<String> words = Arrays.asList(str.toUpperCase().split("\\s+"));

    List<String> filteredWords = (List<String>) FuncLib.filter(words, new Predicate<String>() {
      public boolean apply(String s) {
        return !UNWANTED.contains(s);
      }
    });

    List<String> mappedWords = (List<String>) FuncLib.map(filteredWords, new Function<String, String>() {
      public String apply(String s) {
        return s.substring(0, 1);
      }
    });

    String theAcronym = (String) FuncLib.reduce(mappedWords, new Combiner<String>() {
      public String apply(String a, String b) {
        return a + b;
      }
    }, "");

    return theAcronym;
  }

  public static void main(String[] args) {
    System.out.println(acronym(args[0]));
  }
}
```

Figure 2

This example uses functional approximations created using Anonymous Classes and Functional Interfaces. The usage of Anonymous Classes is cumbersome and verbose. Also, they don't create closures. The free variables in their bodies can only refer to final or effectively final local variables from the outer scope. There are libraries like Google Guava for Java 6+ for providing functional approximations a bit more professional way than the hand-written example above. Please note that Java 8 has introduced support for functional style programming.

2.1.3 Java - Functional Style - Java 8

```java
import java.util.*;

public class AcronymJava8 {

  private static final List<String> UNWANTED = Arrays.asList("THE", "OF", "AT", "IN");

  public static String acronym(String str) {

    String[] words = str.toUpperCase().split("\\s+");

    return Arrays.stream(words)
      .filter(w -> !UNWANTED.contains(w))
      .map(w -> w.substring(0, 1))
      .reduce("", String::concat);
  }

  public static void main(String[] args) {
    System.out.println(acronym("University of California at Berkeley"));
  }
}
```

Figure 3

This Java 8 version example uses Aggregate Operations and Lambda Expressions. Please note that Lambda Expressions doesn't introduce a new level of scoping and also can't refer to non-final or non-"effectively final" local variables in the outer scope.

2.2 JavaScript

2.2.1 JavaScript - Functional Style

```javascript
// Functional style solution
module.exports = function(str) {
  return str.toUpperCase().split(/\s+/).filter(function(w) {
    return ['THE', 'OF', 'AT', 'IN'].indexOf(w) === -1;
  }).map(function(w) {
    return w[0];
  }).join('');
};

// Cheap way of testing
console.log(module.exports(process.argv[2]));
```

Figure 4

This functional style code example uses methods of the Array object like filter, map and join. Filter and map accepts code as data as argument! We pass the code to decide which elements

to keep and to transform elements, as arguments in this example. The filter and map methods hides complexity and generalize repeating patterns of computation.

2.2.2 JavaScript - Imperative Style

```
// Imperative style solution
module.exports = function(str) {
  var words = str.toUpperCase().split(/\s+/);
  var result = '';
  for (var k in words) {
    if (['THE', 'OF', 'AT', 'IN'].indexOf(words[k]) === -1) {
      result += words[k][0];
    }
  }
  return result;
};

// Cheap way of testing
console.log(module.exports(process.argv[2]));
```

Figure 5

This example is highly imperative, containing a for loop and assignment.

2.3 Python

```
import sys

def acronym(s):
  return ''.join(w[0] for w in s.upper().split() if w not in ['THE', 'OF', 'AT', 'IN'])

print(acronym(sys.argv[1]))
```

Figure 6

This example in Python uses special syntax for generating the result. Although the list generator expression is readable, joining in Python is quite awkward. Python favors fix constructs (syntax) over encouraging the programmer to build his/her metalinguistic abstraction. On the one hand it’s convenient, but on the other hand it’s limiting.

2.4 Ruby

```ruby
#!/usr/bin/env ruby

# def acronym(str)
#   str
#     .upcase
#     .split
#     .reject { |wd| %w(THE OF AT IN).include? wd }
#     .map { |wd| wd[0] }
#     .join
# end

# p acronym('The United States of America')
# p acronym('University of California at Berkeley')

# Augment the built-in String class
class String
  def to_acronym
    upcase
      .split
      .reject { |wd| %w(THE OF AT IN).include? wd }
      .map { |wd| wd[0] }
      .join
  end
end

p 'University of California at Berkeley'.to_acronym
```

Figure 7

This Ruby example augments the standard String class, meaning ALL string object will have the functionality “to_acronym” after loading this in. It’s concise, readable and beautifully DYNAMIC.

2.5 Scheme

2.5.1 Scheme - Functional Style

```scheme
;; Functional style solution
(define (acronym str)
  (apply string
         (map (lambda (wd) (string-ref wd 0))
              (filter (lambda (wd) (not (member wd '("THE" "OF" "AT" "IN"))))
                      (string-tokenize (string-upcase str))))))

;; Cheap way of testing
(values (acronym "The United States of America")
        (acronym "University of California at Berkeley"))
```

Figure 8

It's a standard functional style solution in Scheme, using function composition and using some of the standard higher order procedures like filter and map.

2.5.2 Scheme - Imperative Style

```
;; Imperative style solution
(define (acronym str)
  (let ((w (string-tokenize (string-upcase str)))
        (n '("THE" "OF" "AT" "IN")))
    (do ((w w (cdr w))
         (x '() (if (not (member (car w) n))
                    (cons (string-ref (car w) 0) x)
                    x)))
        ((null? w) (reverse-list->string x)))))

;; Cheap way of testing
(values (acronym "The United States of America")
        (acronym "University of California at Berkeley"))
```

Figure 9

This imperative style solution in Scheme uses the "do" iteration construct. In this example I don't use the part of the construct which is for effect and also since the variables are bound to fresh locations in each and every iteration, one might say it's not that imperative after all. But the looping and blending everything into one is more like the imperative approach.