Why learning about the big ideas in Computer Science already in an introductory course matters - a pragmatic demonstration

by Imre Horvath, 09/03/15 (2nd Edition)

I tell my students, "the language in which you'll spend most of your working life hasn't been invented yet, so we can't teach it to you. Instead we have to give you the skills you need to learn new languages as they appear."

-- Brian Harvey
University of California, Berkeley

Acknowledgments

I'd like say thank you to the following people for reading it, and for giving me valuable feedback for further improvements. I'd like to thank Dr. Harvey for reading it and for suggesting that this writing would be more interesting if it would present solutions in different languages using different paradigms, rather than using just LISP-friendly languages.[1] I'd like to thank Jens Mönig for reading and suggesting an improvement regarding the Snap! example. Prof. Dr. Gabor Suto, Dr. Agnes Kiraly and Dr. Ferenc Pakodi all encouraged me and provided me with great guidance in the beginning preparing this writing.

1. Introduction

Solving problems by writing computer programs requires two kinds of knowledge. The first kind of knowledge is about the syntax and the semantics of the programming language being used. (Someone might want to include the libraries complementing the core language in this enumeration as well) The second kind of knowledge is more like a step back from the concrete language and is more about the programming process. It's about the big ideas in Computer Science (CS), like recursion, higher-order procedures, first-class procedures (closure) and first-class continuations, etc.

To prove that Teaching Professor Dr. Brian Harvey [1] is right, when he says, *"Once you learned the big ideas, they thought, and this is my experience also, learning another programming language isn't a big deal; it's a chore for a weekend."* in his writing *Why SICP matters* [3], I'm going to solve a simple problem in various programming languages using the same big ideas. I've learned all of the programming languages used in this writing after learning the big ideas

[1] In a follow-up writing I'm going to follow that way.

from SICP [2]. (The only exception to this is Scheme, which is the language of instruction in SICP) The learning was mostly about the notation I had to use, to make the language do what I wanted it to do. Using this approach, the effort I had to put in was extremely small.

Please note, that UCB decided to discontinue its SICP based CS introductory course after the retirement of Teaching Professor Dr. Brian Harvey. However they still offer the CS10 (BJC) course which also teaches important big ideas using Snap! which is Scheme in disguise.

Chapter 2 states the problem I'm going to solve.

Chapter 3 presents you with some solutions in different programming languages, all solving the same problem using the same big ideas.

Chapter 4 draws a conclusion.

Chapter 5 lists the references.

2. The problem

The problem to solve is, to compute acronyms from strings.

To solve the problem I need to split the input string into a list of words. Filter this list of words to keep only the "real" words. Map the resulting list using a function which computes the first letter of a word. Finally join the letters to get a single word as the result of the whole computational process.

This above, is an abstract description of the process I want the computer to carry out. I'm using some big ideas from the functional programming paradigm to provide you with a solution.

Let's see in the next chapter, how this abstract description is turned into concrete code in some programming languages, all supporting the functional paradigm.

Again, my only task was to get familiar with the notation of the language at hand…

3. Some solutions

Please note that the examples provided below might differ slightly in terms of implementation, but the underlying concept should be easy to see in all of them.

3.1 Solution in the Scheme programming language

```
(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))))))
```

Figure 3.1.1

```
scheme@(guile-user)> (acronym "University of California at Berkeley")
$4 = "UCB"
```

Figure 3.1.2

We define the procedure `acronym` as depicted above (Figure 3.1.1). When the procedure is applied to a string value, we first compute the uppercase version of that string. Then we tokenize the resulting string, which means we turn it into the list of its words. Next we filter the list of words for the values we want to omit. After this, we produce the list of characters of the first letters of the words remaining. Finally we apply the `string` procedure to the list of characters, effectively creating the string representation of the acronym we are looking for. Figure 3.1.2 shows the result of applying the procedure to the given string value.

Here, in this example we use the `filter` and the `map` higher-order procedures. We also use some string-handling procedures, which are available out-of-the-box in the Scheme implementation GNU Guile [9] we are using.

3.2 Solution in the Snap! programming language[2]

[2] The original Snap! example (based on the BJC course) I presented here was using the COMBINE higher-order procedure with the JOIN block, treating JOIN as a binary operator. However Jens Mönig (co-author of the Snap! visual programming language) has suggested to apply JOIN directly to the list since JOIN is variadic (n-ary). However if your combiner operator is binary, you can always use it with reduce. The reason they use COMBINE in the ACRONYM example is to demonstrate the usage of all the three commonly used higher-order procedures in one expression.

Figure 3.2.1

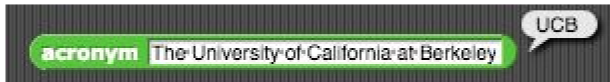

Figure 3.2.2

We define the `acronym` block as shown above (Figure 3.2.1). When the block is called with a sentence argument value, the sentence is split into a list of words. We keep only the words which are none of the listed values like `the`, `of`, `at` and `in`. Then we create a list of the first letters of the filtered list of words. Finally we join them into a single word, which is the result we wanted to compute. Figure 3.2.2 shows an example usage of the block, with the reported value (result).

We use the `keep` and `map` higher-order procedures (represented by blocks in Snap! [4], like any other procedure) when implementing our functional-style solution.

The higher-order procedures used in this example hide a lot of complexity. They abstract these frequently appearing patterns of computation (like filtering and mapping lists), by capturing and generalizing them, using techniques from the functional paradigm.

Snap! [4] is an enhanced version of Scratch [6]. It's Scheme in disguise. It's lexically scoped, has first-class procedures (closure), lists are also first-class and has first-class continuations as well. The visual programming features like using blocks, makes it impossible for the learner to make syntax errors. This too eliminates some complexity when learning how to program. Snap! is used as the language of instruction in the Beauty and Joy of Computing (BJC) course at UC Berkeley [5]. Teaching Professor Dr. Dan Garcia [7] is a passionate, enthusiastic lecturer, who leads the CS10 [8] introductory CS course at UC Berkeley.

3.3 Solution in the Ruby programming language

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

Figure 3.3.1

```
irb(main):002:0> acronym 'The University of California at Berkeley'
=> "UCB"
```

Figure 3.3.2

We define the method `acronym` as seen above (Figure 3.3.1). In the method, we produce the uppercase version of the original string. Then we split it into an array of words. Next we reject the words we don't want to include in the further processing. Then we map through the list of the remaining words, creating a new array of one-letter strings. These one-letter strings are the first letters of the original words. Finally we join the array to produce a single string, which is the acronym we want to return as the result of our computation. Figure 3.3.2 shows us an example usage with its result value.

Here we send the messages `reject` and `map` (Ruby [10] has roots in functional programming) to the array object under consideration. The underlying system will call the corresponding methods on the objects. (Ruby has roots in Smalltalk too) We associate the given blocks with the method calls. This is essentially the same principle as discussed in the previous examples above when working with higher-order procedures. In Ruby, blocks are closures, but not first-class objects, though they can easily be converted into first-class objects, if needed.

These are method calls on objects as opposed to applying a function to arguments. However the same principle applies here as well. It's amazing to see how the object-oriented and functional paradigms mix.

3.4 Solution in JavaScript

```
function acronym(str) {
  return str.
    toUpperCase().
    split(/\s+/).
    filter(function(wd) {
      return ['THE', 'OF', 'AT', 'IN'].indexOf(wd) === -1;
    }).
    map(function(wd) {
      return wd.charAt(0);
    }).
    join('');
}
```

Figure 3.4.1

```
> acronym('University of California at Berkeley');
'UCB'
```

Figure 3.4.2

In this example I'm not going to repeat myself by providing you with a very similar description already provided in the previous examples above. Instead, go ahead and try to figure it out by yourself by looking at the definition in figure 3.4.1 and the application shown in figure 3.4.2!

3.5 Solution in Java 8

```
import java.util.*;

public class Acronym {

  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 3.5.1

```
Success time: 0.24 memory: 320896 signal:0
UCB
```

Figure 3.5.2

Looking at figure 3.5.1, we can see that the solution is pretty similar in Java 8. That's because Java 8 introduced Lambda Expressions and Aggregate Operations, to support functional style programming[3]. I don't want to go into details, but I'd like to point out that their approach is not quite the same as we are used to in other lexically scoped languages with lambdas. The most important thing is that their Lambda Expressions don't introduce a new level of scoping and also they can only have free variables in their bodies which are references to final or effectively final local variables of the enclosing scope. Despite these limitations we could provide you with a very similar solution and with the same result as shown in figure 3.5.2.

4. Conclusion

[3] There are libraries like Google Guava for Java 6+, which provides among other things functional approximations, and some functional style programming. However, its creators discourage people using it excessively.

Traditionally, introductory CS courses focus on teaching the details of a highly imperative programming language having different notation for declaring variables, branching, looping and calling procedures. All these details like the huge number of syntax rules, precedence rules of the operators and the relative order of the assignments, introduce extra complexity, a newcomer has to deal with. As a result, it takes a lot of time, till the student can get to the big ideas buried underneath. Learning other languages supporting different paradigms afterwards can often be challenging for students.

To the contrary, SICP [2] teaches us about all the mainstream programming paradigms, like imperative, object-oriented, functional and declarative (sometimes referred to as logic-programming) in one introductory course. Someone might be surprised to hear, but SICP uses only one programming language to teach all the paradigms. This language (Scheme) uses a simple, uniform notation for everything. Its notation makes the order explicit, so no precedence rules apply. The language is so minimalistic, it can be learned during the first lecture. After this short initial investment, the whole semester is free to learn about the big ideas, enabling students to focus on the more important, big-picture content much sooner. It's the perfect language to teach you the skills you need, to learn new languages.

One big idea used in all the solutions in Chapter 3, is to *treat procedures as data*. This idea has helped a lot, when I was using the higher-order procedures `filter` and `map` to provide an elegant solution. These higher-order procedures capture and generalize repeating patterns of computation found, when dealing with lists. Eg. `filter` is for creating a new list from an existing one, containing only the elements for those the predicate procedure passed, yields true. It's written once, and can be used for each and every filtering task. It's important to see, that this kind of generalization creates two parts. One of them is the "fixed" part, which is represented by the body of the higher-order procedure. In our example, this captures the concept of filtering a list. The other one is the "variable" part of the computation. This "variable" part captures the details which vary by the different use-cases, and is represented by the body of the procedure being passed as data. In our filtering example this is the predicate procedure which decides if we want to keep a particular element or not.

The other big idea, the *recursion* was not seen directly in these examples. But it's at the heart of the higher-order procedures `filter, map` and `reduce`. You can solve similar problems by using recursion or by using already existing higher-order procedures. The difference is in their flexibility. Recall that, higher-order procedures can be used, to generalize repeating patterns of computation. If your problem fits in this pattern, then you can go for using the higher-order procedure. If not, then you can invent your custom tailored higher-order procedure, and implement it in terms of recursion.

Thinking about the programming process, in a language-independent fashion, enabled me to provide an abstract solution. The only remaining task was, to learn the notation of the concrete languages to be able to write the working code. This was much easier for me this way. Provided

that, I had no clue about the big ideas in CS, I would have had a really hard time learning all those languages to solve the problem above.

Please don't take me wrong and think this writing is only about functional programming. Although the provided examples are all functional in style, it's about the big ideas. It's important to see, that I only cover a few of them in this writing. There are much more to read about in SICP [2], and I do encourage you to read about them!

Learning about the big ideas from the textbook SICP [2], provided me with the necessary skills, to learn new languages as they appear.

That's why I say, learning about the big ideas in CS already in an introductory course matters.

5. References

1. Homepage of Teaching Professor Dr. Brian Harvey, University of California, Berkeley - http://www.cs.berkeley.edu/~bh/index.html

2. Structure and Interpretation of Computer Programs, 2nd Edition, Harold Abelson and Gerald Jay Sussman with Julie Sussman, The MIT Press, 1996 - https://mitpress.mit.edu/sicp/full-text/book/book.html

3. Why Structure and Interpretation of Computer Programs matters, Brian Harvey, University of California, Berkeley, 2011 - http://www.cs.berkeley.edu/~bh/sicp.html

4. Snap! Build Your Own Blocks - A web-based visual programming environment for teaching introductory CS courses - http://snap.berkeley.edu/snapsource/snap.html#open:http://snap.berkeley.edu/snapsource/tools.xml

5. Beauty and Joy of Computing course at UC Berkeley - http://bjc.berkeley.edu/

6. Scratch - https://scratch.mit.edu/

7. Homepage of Teaching Professor Dr. Dan Garcia, University of California, Berkeley - http://www.cs.berkeley.edu/~ddgarcia/

8. About the CS10 course at UC Berkeley - http://cs10.org/

9. GNU Guile Scheme - https://www.gnu.org/software/guile/

10. Ruby - https://www.ruby-lang.org/en/