Not All Programming Languages are Created Equal: Using Functional Languages in Introductory Computer Science Classes.

Elena Machkasova

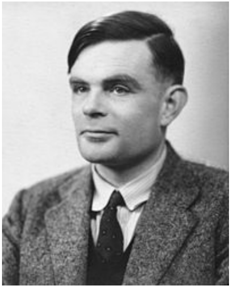
University of Minnesota, Morris

Thursday Afternoon Faculty Seminar, October 3 2013.

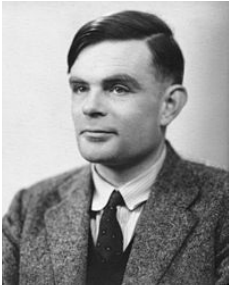
Curious facts about my research

- My research is in programming languages.
- In the past: some theoretical and practical work on how to make programs run faster while getting the same behavior (program optimization).
- Recently: **exploring use of a new programming language Clojure in introductory CS classes.**
- Clojure: a functional programming language released on October 16, 2007.

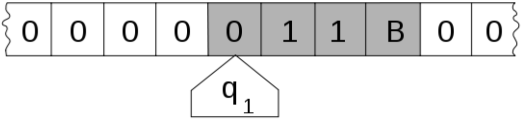
Foundations of computing systems: how it all started.



Source: http://en.wikipedia.org/wiki/Alan_Turing



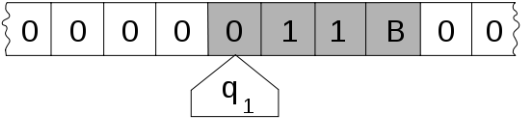
Source: `http://en.wikipedia.org/wiki/Alan_Turing`
Alan Turing 1936: *Turing machine*: an abstract computing device.



Source: `http://en.wikipedia.org/wiki/Turing_machine`

Turing machine: an abstract computing device.

Foundations of computing systems: Turing machine.



Source: `http://en.wikipedia.org/wiki/Turing_machine`

Turing machine: an abstract computing device.

Consists of an infinite tape (*memory*) and a movable read/write head that reads and writes symbols on the type: (*direct changes in memory*).

Executes rules, e.g. *if the current memory cell has 0, write 1 and move left.*

Internal architecture of a computer, such as CPU (central processing unit) and memory, are based on concepts of the Turing machine.

Source: `http://en.wikipedia.org/wiki/Alonzo_Church`

Foundations of computing systems: how it all started.

Source: `http://en.wikipedia.org/wiki/Alonzo_Church`
Alonzo Church (Alan Turing's thesis adviser).
1930s: *the lambda calculus (λ-calculus)*: a system in mathematical logic that describes computations.

A computation is a sequence of function applications, similar to $f(g(x))$.
Functions: $\lambda x.x + 1$ is a function that adds 1 to its argument.
λx means that a function takes one argument x, and $x + 1$ is the value that the function computes.
Does not deal with details of implementation or memory manipulation.

A computation is a sequence of function applications, similar to $f(g(x))$.
Functions: $\lambda x.x + 1$ is a function that adds 1 to its argument.
λx means that a function takes one argument x, and $x + 1$ is the value that the function computes.
Does not deal with details of implementation or memory manipulation.

Church-Turing thesis: the Turing machine and the λ-calculus can both perform all possible algorithms that can be described by computable functions.
Languages that are equivalent to the Turing machine are called *Turing complete*.
All general purpose programming language are Turing complete.

A computation is a sequence of function applications, similar to $f(g(x))$.
Functions: $\lambda x.x + 1$ is a function that adds 1 to its argument.
λx means that a function takes one argument x, and $x + 1$ is the value that the function computes.
Does not deal with details of implementation or memory manipulation.

Church-Turing thesis: the Turing machine and the λ-calculus can both perform all possible algorithms that can be described by computable functions.
Languages that are equivalent to the Turing machine are called *Turing complete*.
All general purpose programming language are Turing complete.
All languages are equivalent. Are they equal?.

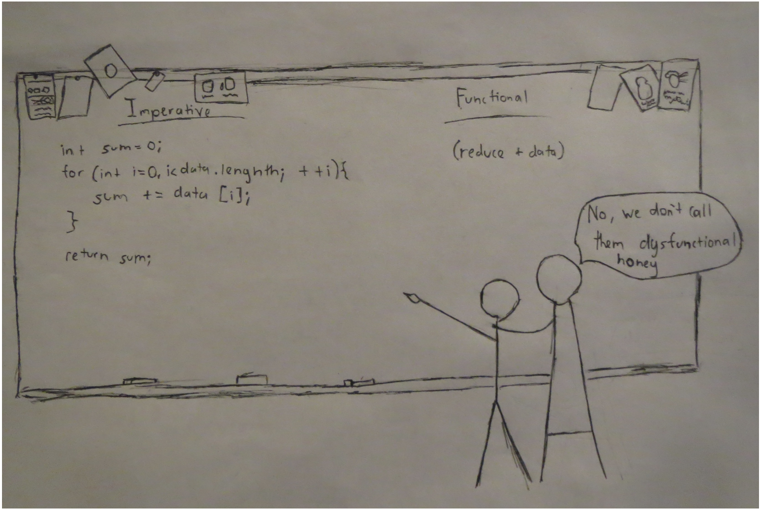
First programming languages.

First computers were programmed only by machine instructions.
High-level programming languages appeared so that programs can be understandable to humans.
First high-level languages:

- 1957: **Fortran** (**For**mula **Tran**slator), based on memory manipulation (like a Turing machine),
- 1958: **Lisp** (**Lis**t **P**rocessing), based on function composition (like the λ-calculus),
- 1959: **Cobol** (**Co**mmon **B**usiness-**O**riented **L**anguage), based on memory manipulation (like a Turing machine).

Languages that are like the λ-calculus are called *functional*.
Languages that are like a Turing machine are called....

Functional vs imperative.



Credit: Leonid Scott.

First programming languages.

First computers were programmed by machine instructions.
High-level programming languages appeared so that programs can be understandable to humans.
First high-level languages:

- 1957: **Fortran** (**For**mula **Tran**slator), based on memory manipulation (like a Turing machine),
- 1958: **Lisp** (**Lis**t **P**rocessing), based on function composition (like the λ-calculus),
- 1959: **Cobol** (**Co**mmon **B**usiness-**O**riented **L**anguage), based on memory manipulation (like a Turing machine).

Languages that are like the λ-calculus are called *functional*.
Languages that are like a Turing machine are called *imperative*.

Imperative	Functional
Most entities are changed (updated) in-place.	Most entities are immutable: a new entity is created upon an update.
Results are accumulated by updating a memory locations, often in a loop.	Results are accumulated as a sequence of function calls, e.g. recursion (a function calls itself).
Functions are defined before the program starts executing, encapsulate some functionality.	Functions are first-class citizens: can be anonymous, constructed "on-the-fly", passed to other functions, returned from other functions.
Built-in data structures are specified in terms of memory addresses (e.g. arrays: contiguous blocks of memory)	Built-in data structures are defined inductively, e.g. a part of a list is itself a list.

Sum up all the elements of a dataset `data`.

Imperative, a loop: `sum` is a variable to accumulate results, `i` is an index.

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Assume `data` is `5, 7, 3, ...`

Imperative vs functional: example

Sum up all the elements of a dataset `data`.

Imperative, a loop: `sum` is a variable to accumulate results, `i` is an index.

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Assume `data` is `5, 7, 3, ...` Then:

`i = 1, sum = 5` on the first iteration of the loop,

Imperative vs functional: example

Sum up all the elements of a dataset `data`.

Imperative, a loop: `sum` is a variable to accumulate results, `i` is an index.

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Assume `data` is `5, 7, 3, ...` Then:

`i = 1, sum = 5` on the first iteration of the loop,

`i = 2, sum = 12` on the second,

Sum up all the elements of a dataset `data`.

Imperative, a loop: `sum` is a variable to accumulate results, `i` is an index.

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Assume `data` is `5, 7, 3, ...` Then:

`i = 1, sum = 5` on the first iteration of the loop,

`i = 2, sum = 12` on the second,

`i = 3, sum = 15` on the third,.......

Imperative vs functional: example

Sum up all the elements of a dataset `data`.

Imperative, a loop: `sum` is a variable to accumulate results, `i` is an index.

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Assume `data` is `5, 7, 3, ...` Then:

`i = 1, sum = 5` on the first iteration of the loop,

`i = 2, sum = 12` on the second,

`i = 3, sum = 15` on the third,.......

Error-prone: when does the index change? Are we using the right element?

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Functional: *recursion*:

```
function sum (data):
  if isEmpty?(data) then 0
  else first(data) + sum(rest(data))
```

If data has no elements, the sum is 0, otherwise it's the result of adding the first element to the sum of the rest of data.

Imperative vs functional: example

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Functional: *higher order functions*:

```
(reduce + data)
```

+ is a *function* (not an operation, as in imperative languages), we *pass it* to a function `reduce` that applies it to all elements.

Imperative vs functional: example

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Functional: *higher order functions*:

`(reduce + data)`

`+` is a *function* (not an operation, as in imperative languages), we *pass it* to a function `reduce` that applies it to all elements.

`(reduce minimum data)`

`minimum` is a function that finds the smaller of two elements.

Imperative vs functional: example

```
sum = 0;
for (i = 1; i < length(data); i = i + 1) {
    sum = sum + data[i];
}
```

Functional: *higher order functions*:

`(reduce + data)`

+ is a *function* (not an operation, as in imperative languages), we *pass it* to a function `reduce` that applies it to all elements.

`(reduce minimum data)`

`minimum` is a function that finds the smaller of two elements.

(reduce (lambda x y. if $x < 0$ then 0 else $(x + y)$) data)
Anonymous function adds up positive data elements only.

Imperative vs functional: tradeoffs

How do functional languages store intermediate data? When a function starts, it allocates a workspace where its intermediate results are stored.

How do functional languages store intermediate data? When a function starts, it allocates a workspace where its intermediate results are stored.

1000 function calls = 1000 workspaces??? Isn't it slow and memory-intensive???

Imperative vs functional: tradeoffs

How do functional languages store intermediate data? When a function starts, it allocates a workspace where its intermediate results are stored.

1000 function calls = 1000 workspaces??? Isn't it slow and memory-intensive???

There are built-in optimizations in functional languages. Some degree of awareness of efficiency is required.

Imperative languages with a direct control of low-level memory access (e.g. C) can indeed be faster.

Imperative vs functional: tradeoffs (cont)

Functions as first-class citizens allow more modular and abstract code: no need to write a separate loop for adding all elements of a dataset and for multiplying since + and * (multiplication) can be passed to a function:

```
(reduce + list)
(reduce * list)
```

Functional languages promote writing shorter functions: it's easier to "separate concerns".

Imperative vs functional: tradeoffs (cont)

Effects of *immutability*:

- easier to develop correct programs: do not need to keep track of a changing state of entities in a program.
- more program optimization: knowing that a certain entity doesn't change allows remembering and pre-computing of parts of a program.
- functional programs are easier to run in parallel (multiple CPUs or distributed computation): since entities don't change, no need for access control to shared data since there's no risk of an accidental overwrite.
- functional languages explicitly make data mutable: easier to track.

Convenience for parallelization renewed interest in functional languages.

Current language landscape.

There are 100+ actively used programming languages.
Imperative are much more commonly used: *C*, *C++*, *Java*, *C#*, *python*, *JavaScript*, *php*, etc.
Several **functional** languages in active use: dialects of *Lisp (Common Lisp, Scheme, Clojure)*, *Haskell*, *Erlang*, *Scala*.
Functional languages are on the rise with parallel computation.
Mixed paradigm: many modern languages are imperative, but with some support for immutability and higher-order functions.

Current language landscape.

There are 100+ actively used programming languages.
Imperative are much more commonly used: *C, C++, Java, C#, python, JavaScript, php*, etc.
Several **functional** languages in active use: dialects of *Lisp (Common Lisp, Scheme, Clojure), Haskell, Erlang, Scala.*
Functional languages are on the rise with parallel computation.
Mixed paradigm: many modern languages are imperative, but with some support for immutability and higher-order functions.
Object-oriented languages: represent real-life entities as object in a program (modularity): Java, C++, C#, Scala.

Skills/background for CSci students.

Students entering computing industry...

- ...are expected to know at least one commonly used language (C++, Java, C#) well.
- ....will have to be learning new languages/paradigms as early as within the first week at work.
- ...will be working with multiple languages and systems at the same time.
- ...are expected to write well-organized clear program code.
- ...greatly benefit from knowing a variety of tools and approaches for collaborative software development process, testing, managing projects, professional communication skills.

Independent, self-directed learning, building upon a strong knowledge foundation and experience with collaborative work.

UMM curriculum structure.

Introductory courses. CSci 1201: python (imperative with functional elements) or CSci 1301 Racket (a functional language, dialect of Lisp).
Sophomore level. Course: CSci 2101 Data Structures: Java (imperative, object-oriented).
Sophomore/Junior level. Intro to operating systems, larger scale software development, algorithms development.
Junior/Senior level. Electives.

Why a functional language in an intro CS class?

Why teach a functional language in an intro class if mostly imperative languages are used later?

Why a functional language in an intro CS class?

Why teach a functional language in an intro class if mostly imperative languages are used later?

- Immutability makes design easier: focus on concepts.
- Focus on abstraction, generalization, and modularity.
- Focus on functions.
- Focus on compact, logical code design.
- Better understanding of recursion (useful for recursive data structures and algorithms later)

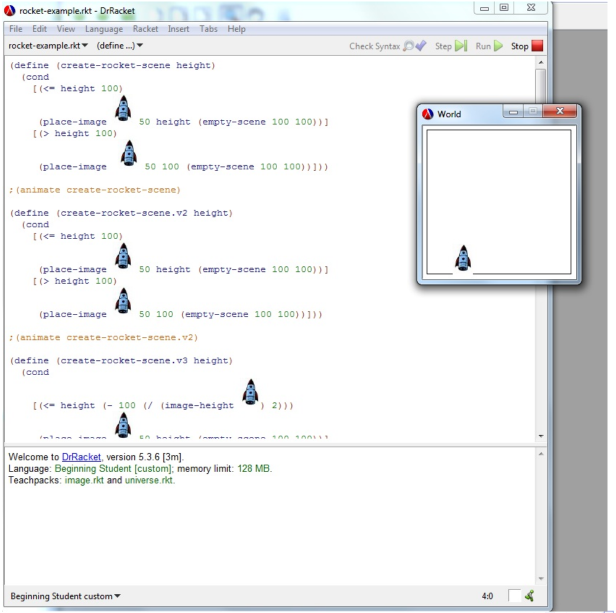
Why a functional language in an intro CS class?

Why teach a functional language in an intro class if mostly imperative languages are used later?

- Immutability makes design easier: focus on concepts.
- Focus on abstraction, generalization, and modularity.
- Focus on functions.
- Focus on compact, logical code design.
- Better understanding of recursion (useful for recursive data structures and algorithms later)

Students better learn more involved concepts (e.g. object-oriented approaches) with a strong base in general concepts.

Current CSci 1301: Racket programming language



.

- A dialect of Lisp, specifically designed for teaching.
- Comes with an environment that allows students to easily write and execute their code.
- Has multiple levels: Beginner Student, Intermediate, Advanced. Lower levels make only a subset of features available to students, to avoid accidentally executable, but incorrect code. Also change the way output is formatted.
- Incorporates a system for working with images and allows students to provide functions for interactive environments ("when a key is pressed, do this") without dealing with back-end structures.

UMM has been using Racket, and its predecessor Scheme, in intro classes for about 15 years. It worked well.

The Clojure programming language.

Clojure is a new language in the Lisp family.

- Built-in immutability, for safe sharing during parallel processing.
- Built-in support for several models of mutability for cases when data needs to be concurrently modified.
- Running in the Java Virtual Machine: a well-developed engine for running Java: can piggy-back on its optimizations, interactions with the operating systems, etc.
- Fully interoperable with Java: can use all Java libraries.
- Used in industry, has a large (and fast growing) community around it (conferences, meetups, open source projects,...)

The Clojure programming language.

Clojure is a new language in the Lisp family.

- Built-in immutability, for safe sharing during parallel processing.
- Built-in support for several models of mutability for cases when data needs to be concurrently modified.
- Running in the Java Virtual Machine: a well-developed engine for running Java: can piggy-back on its optimizations, interactions with the operating systems, etc.
- Fully interoperable with Java: can use all Java libraries.
- Used in industry, has a large (and fast growing) community around it (conferences, meetups, open source projects,...)
- It's a nice language!

Clojure at UMM: faculty, students, alums

- First introduced by Brian Goslinga (UMM CSci'11)
- Made an appearance in parallel/distributed computing class (2011 Elena, 2013 Nic) and Programming Languages (2012).
- Three projects: improving Clojure error messages (Brian Goslinga UROP, Eugene Butler LSAMP), interoperability between Java and Clojure (Stephen Adams UROP), parallelization in Clojure (Joe Einertson UROP).
- An idea came up that we should try using Clojure in an introductory class.
- Starting Fall 2012, a joint work with Stephen Adams (UMM CSci 2012), Joe Einertson (UMM CSci 2013), plus a directed study with Paul Schliep and Max Magnuson (UMM CSci 2015).
- A presentation at Trends in Functional Programming in Education (TFPIE), May 2013.

Clojure at UMM: faculty, students, alums

- First introduced by Brian Goslinga (UMM CSci'11)
- Made an appearance in parallel/distributed computing class (2011 Elena, 2013 Nic) and Programming Languages (2012).
- Three projects: improving Clojure error messages (Brian Goslinga UROP, Eugene Butler LSAMP), interoperability between Java and Clojure (Stephen Adams UROP), parallelization in Clojure (Joe Einertson UROP).
- An idea came up that we should try using Clojure in an introductory class.
- Starting Fall 2012, a joint work with Stephen Adams (UMM CSci 2012), Joe Einertson (UMM CSci 2013), plus a directed study with Paul Schliep and Max Magnuson (UMM CSci 2015).
- A presentation at Trends in Functional Programming in Education (TFPIE), May 2013.
- We now have a name for this project: **ClojurEd**

Potential benefits of using Clojure in intro classes.

- If done right, we can get most of the same benefits as from using Racket.
- Additionally, it integrates nicely into future classes in the way Racket cannot.
- There is a large community around it: opportunities to get help or to jump into a project. Also, issues are fixed fast and the language support doesn't depend on a dozen of individuals.
- Allows fast parallel execution.
- It's a real-life language done well, and there aren't that many out there.

What needs to be done?

What needs to be done before we can teach Clojure to intro students?

What needs to be done?

What needs to be done before we can teach Clojure to intro students? **A lot!**

What needs to be done?

What needs to be done before we can teach Clojure to intro students? **A lot!** Work in progress:

- Environment for beginners: looking for a beginner-friendly text editor, need to add a few things to make it automatically run Clojure with our libraries.
- Error messages in Clojure are not beginner-friendly. We have done some work on improvements.
- We are changing and adding Clojure functions to make them easier to use for beginners.
- Developing an approach to introducing Clojure abstractions to beginners.
- Exploring Clojure libraries for graphics that would allow students to work with images, similar to Racket.

Bits of the universe: a Clojure poem

```
nil
pop!
(time ())
empty?
atom
atom atom
atom atom atom atom atom
atom atom atom atom atom atom atom atom atom
(binding [atom [atom [atom [atom [atom]]]]])
make-hierarchy
repeat
repeatedly iterate sequence
constantly interleave
merge

replicate
parents
descendants
cycle
7000000000

true? false?
find identity
find name
symbol? number?
rational? odd?
resolve
future?
future-done?
future-cancelled?
reversible?
nil?
```