CSC324: Principles of Programming Languages

Lecture 0

Wednesday 6 May, 2020

Welcome to CSC324!

Glad you were all able to find the lecture hall.

Goals for today

- Introducing the teaching staff (5 minutes)
- Walking through the syllabus and other administrivia (10 minutes)
- Course overview (35 minutes)

Teaching team

I'm Nathan Taylor `<nb.taylor@utoronto.ca>`.

- B.Sc: University of Alberta
- Grad school: UBC Systems & Security lab
- Spent ~8 years in Silicon Valley, working on distributed systems, language runtimes, and compilers
  - Glad to be back home again!

Also: we have three great TAs to help!

Learning team

Who are you?

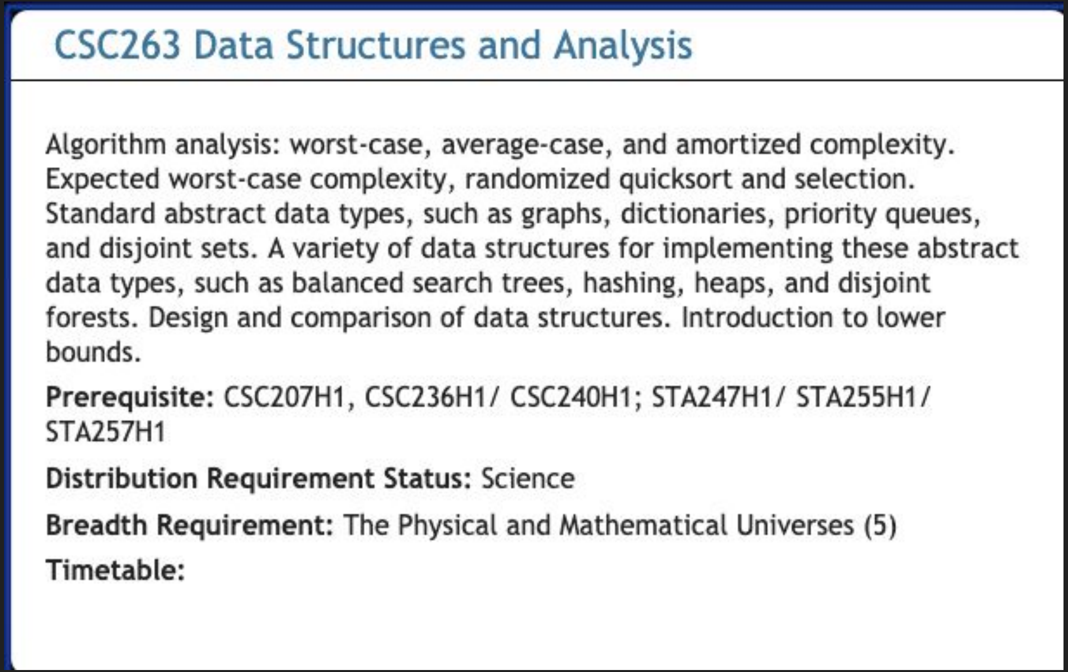
Please fill out the form linked to on the discussion forum to tell us about yourselves; this will help us adjust the course material based on your interests.

Administrivia: Lectures and Lab times

- Lectures: Wednesdays and Fridays at **12:10-13:00 EST**
  - **Lectures will be livestreamed, with recordings posted "at some point in the future"**
- Labs: Mondays at **12:10-13:00 EST**
  - **Run as drop-in TA office hours.**
- **Office hours: Thursdays 15:00-17:00 EST** or by appointment

Administrivia: Formal Prerequisites

CSC263/CSC265 (or equivalent)



CSC263 Data Structures and Analysis

Algorithm analysis: worst-case, average-case, and amortized complexity. Expected worst-case complexity, randomized quicksort and selection. Standard abstract data types, such as graphs, dictionaries, priority queues, and disjoint sets. A variety of data structures for implementing these abstract data types, such as balanced search trees, hashing, heaps, and disjoint forests. Design and comparison of data structures. Introduction to lower bounds.

Prerequisite: CSC207H1, CSC236H1/ CSC240H1; STA247H1/ STA255H1/ STA257H1

Distribution Requirement Status: Science

Breadth Requirement: The Physical and Mathematical Universes (5)

Timetable:

Administrivia: Informal Prerequisites

What do you actually need to know to succeed in this class?

The study of programming languages is unique in that it blends needing:

- practical programming ability
- theoretical underpinnings and formalism from theoretical CS.

Administrivia: Informal Prerequisites

Specifically, this translates to you needing to have:

- Familiarity with common data structures such as linked lists and trees;
- Competency in some prior programming language, such that you could implement one of the above data structures;
- Comfort with recursion, and understanding and writing proofs by induction.

Administrivia: Course work

- 10 approximately-weekly **exercises** (3% each)
    - Exercises are regular checkups intended to help you keep up with the course.
- Weekly **labs** (0%)
    - Labs are your opportunity to get hands-on experience with material covered in lecture.
    - Not graded, but completing them is still expected, on your own time or during the lab
    - TAs will gladly assist with any questions you have on the material covered.
- Two **assignments** (15% each)
    - Nontrivial programming projects that place the course material in a broader context
    - You may work individually or with a partner
- Two **quizzes** (10% each)
    - Takes the place of a traditional mid-term exam
- **Final** assessment (20%)
    - Takes the place of a traditional exam at the end of the course

Administrivia: Readings

- No formal textbook; we will be using David Liu's lecture notes, linked from the course webpage

- Slides and demo code will be posted on the course webpage.
    - After the first few lectures, most of our lecture time will be spent in a code editor. I will endevour to upload the code we write after lecture.

- Installing languages and documentation are also linked to on the course webpage.

Learning PL vs learning Racket and Haskell

In this class we use the programming languages Racket and Haskell to study notions of programming language design. This is not a class about programming in these languages, however.

Typically, I will lecture on a topic focusing on it with examples and live-programming in <u>either</u> Racket or Haskell. *It is your responsibility to learn how the topic works in the other language, or understand why the topic doesn't apply to the other language*.

Course policies - communications

- All announcements will be made on the course forum, if not in lecture

- Please post all course material-related questions on the forum
  - And, in the name of active learning, feel free to answer each others' questions, too!
  - Usual rules apply: do not post partial solutions to exercises or assignments

- For other kinds of questions, email me.

Course policies - assignment submissions

- Assignments to be submitted through MarkUs

- You are free to develop your solutions on your personal machines, but all submissions will be tested on the department teaching lab machines, and thus must work correctly in that environment

Course policies - working with a partner

- Weekly exercises, quizzes, and the final assessment must be done individually

- For the two assignments, we encourage you to work with a partner
    - To **form** a partnership, form a group on MarkUs.
    - You may only **dissolve** a partnership with the permission of the teaching staff.

Course policies - late assignments

- MarkUs may be slow right before a deadline; we encourage you to submit early!

- Each student will receive six **flex tokens** to extend a deadline by two hours
  - You may use any number of remaining tokens (ie. all six tokens will give you a 12h extension)

- For group work, tokens are deducted from **each** team member!

Course policies - Special consideration

If you are unable to complete homework or if you miss a test due to major illness or other circumstances completely outside of your control, get in touch with us immediately if you want to receive special consideration.

The University of Toronto is committed to accessibility. If you require accommodations or have any accessibility concerns, please visit Accessibility Services as soon as possible.

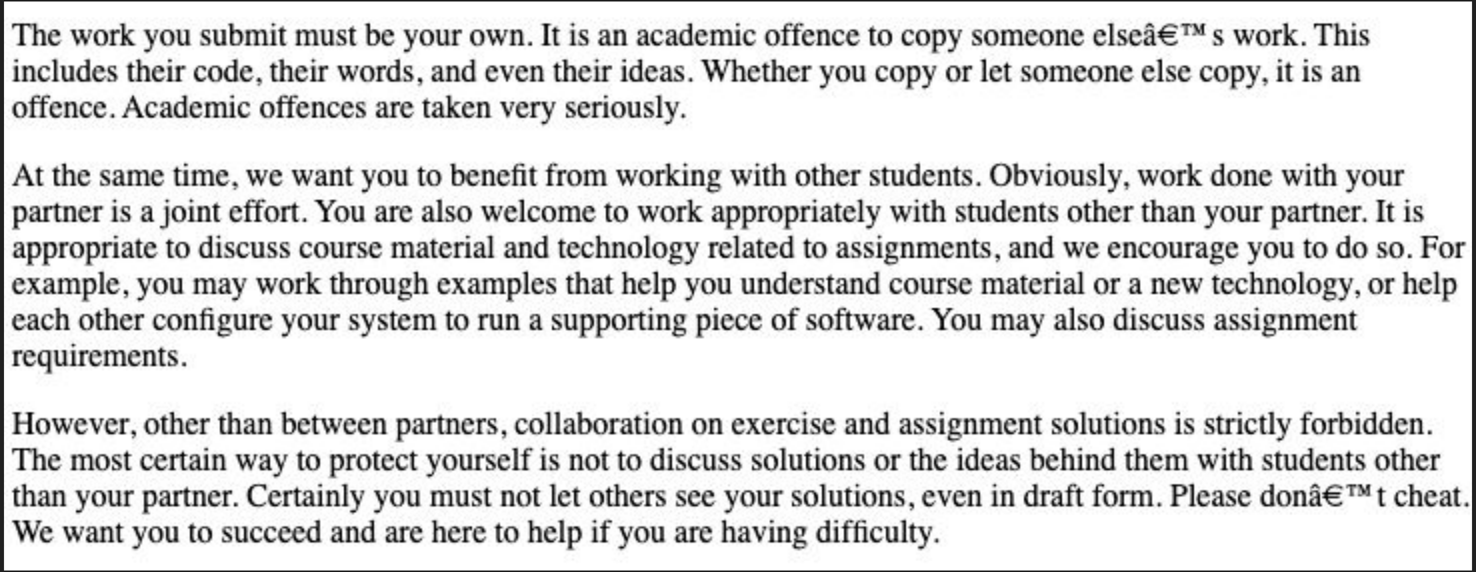
See the syllabus for more details.

Course policies - Remark requests

- The teaching staff is happy to meet with you to discuss areas of praise and improvement of your assignments and exercises. Get in touch at any point in the course if this is of interest.

- If you would like to request a regrade, email me within a week of the mark being returned to you, making your case for the regrade.
    - Warning: There is no guarantee that your remarked assignment will necessarily have an equal or greater score than the original!

Course policies - academic integrity


The work you submit must be your own. It is an academic offence to copy someone elseâ€™s work. This includes their code, their words, and even their ideas. Whether you copy or let someone else copy, it is an offence. Academic offences are taken very seriously.

At the same time, we want you to benefit from working with other students. Obviously, work done with your partner is a joint effort. You are also welcome to work appropriately with students other than your partner. It is appropriate to discuss course material and technology related to assignments, and we encourage you to do so. For example, you may work through examples that help you understand course material or a new technology, or help each other configure your system to run a supporting piece of software. You may also discuss assignment requirements.

However, other than between partners, collaboration on exercise and assignment solutions is strictly forbidden. The most certain way to protect yourself is not to discuss solutions or the ideas behind them with students other than your partner. Certainly you must not let others see your solutions, even in draft form. Please donâ€™t cheat. We want you to succeed and are here to help if you are having difficulty.

On to the course!

Motivation: Why should I care about any of this?

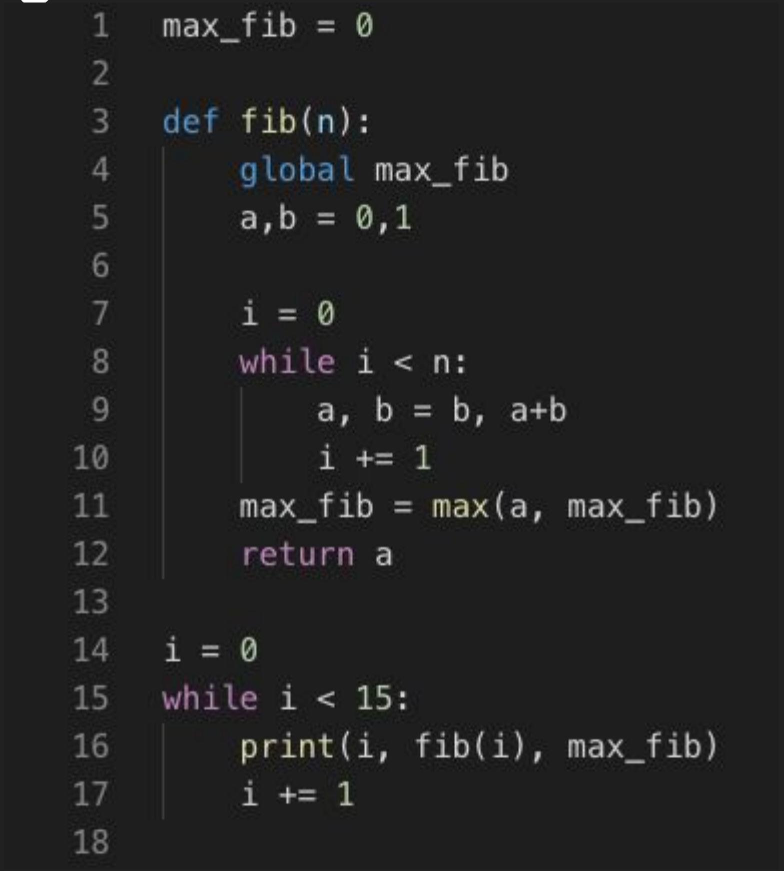
We'll wrap up this first lecture by discussing the principal topics of this class:

- The **iterative vs functional** programming models
- **Metaprogramming**: writing programs that modify programs
- Proving aspects of program correctness with **type theory**

This is meant to be a 10,000-foot view of the material!

1. The imperative programming model

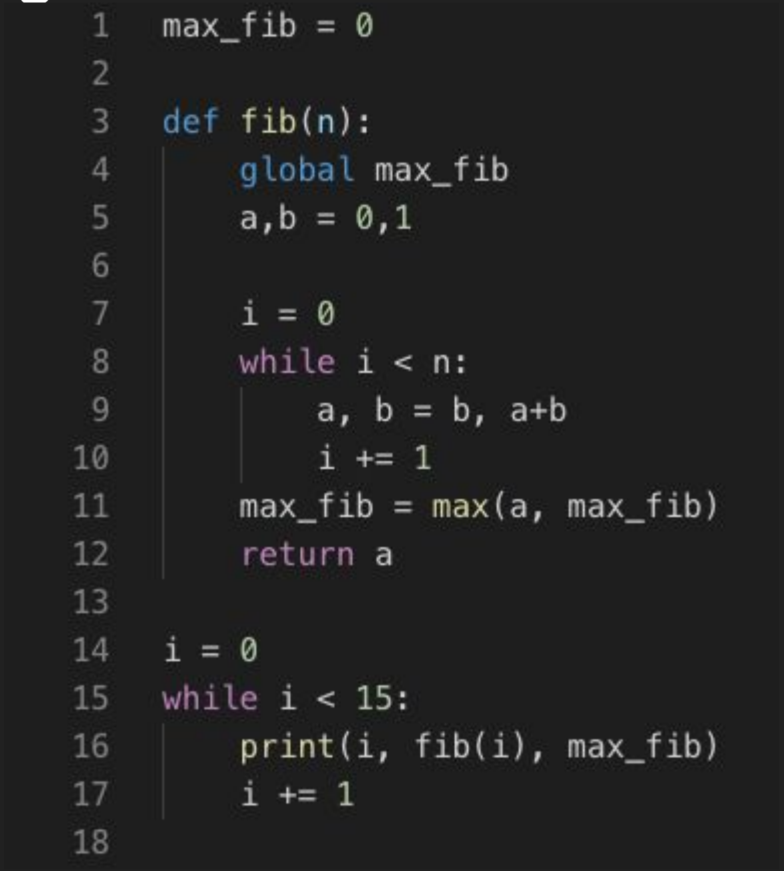
Here's a great program that I wrote!



```python
max_fib = 0

def fib(n):
    global max_fib
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    max_fib = max(a, max_fib)
    return a

i = 0
while i < 15:
    print(i, fib(i), max_fib)
    i += 1

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

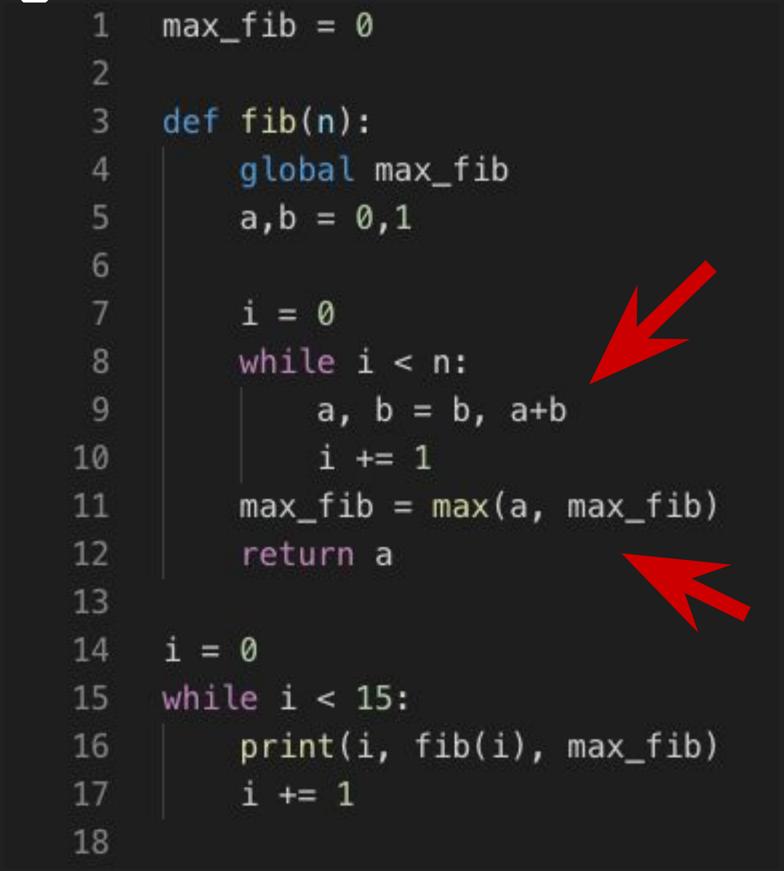


```python
max_fib = 0

def fib(n):
    global max_fib
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    max_fib = max(a, max_fib)
    return a

i = 0
while i < 15:
    print(i, fib(i), max_fib)
    i += 1

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

Such **statements** might **modify the value** of variables defined within the program as it runs.

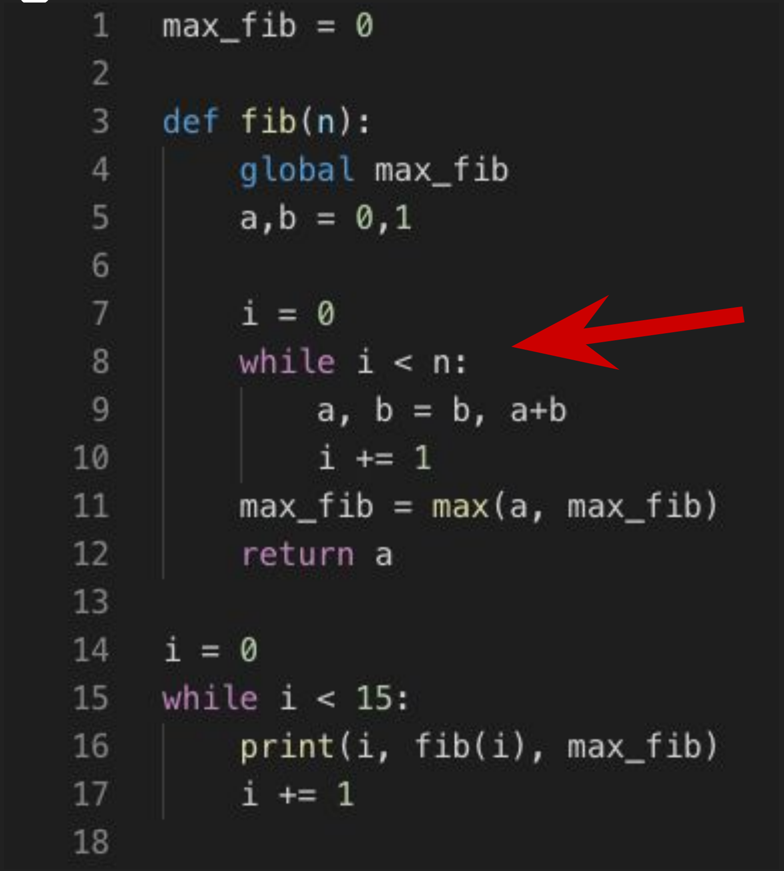


```python
max_fib = 0

def fib(n):
    global max_fib
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    max_fib = max(a, max_fib)
    return a

i = 0
while i < 15:
    print(i, fib(i), max_fib)
    i += 1

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

Such steps might **repeat** until a variable has been mutated to some value



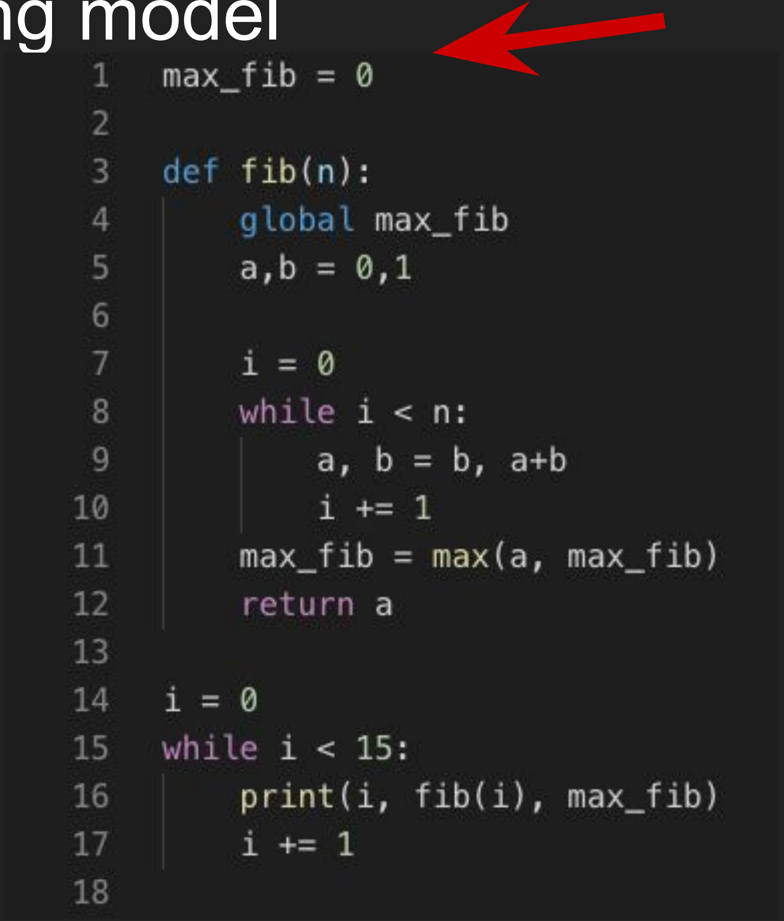
```python
max_fib = 0

def fib(n):
    global max_fib
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    max_fib = max(a, max_fib)
    return a

i = 0
while i < 15:
    print(i, fib(i), max_fib)
    i += 1

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

A step like calling a function might return a value, but could **also** result in a **side-effecting operation,** such as modifying a global variable.

We know global variables are "bad", but...

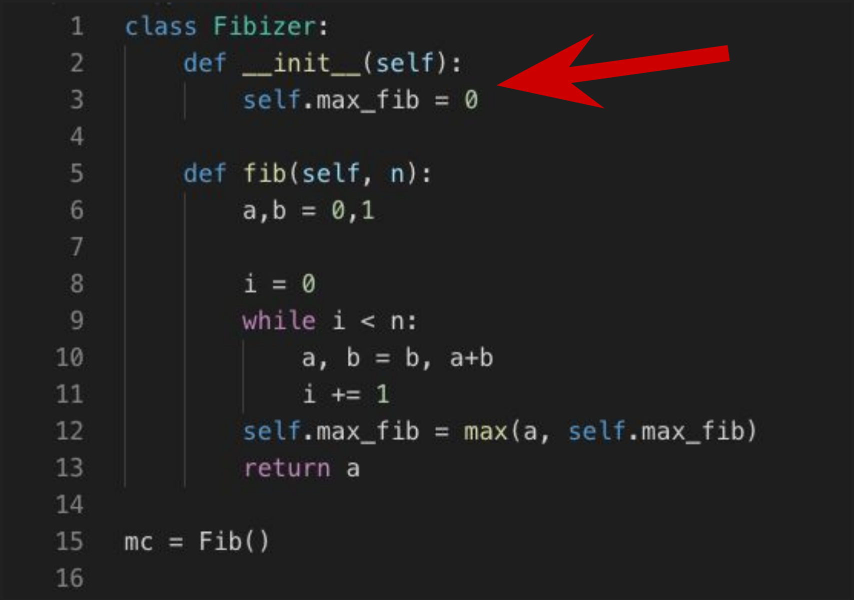


```python
max_fib = 0

def fib(n):
    global max_fib
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    max_fib = max(a, max_fib)
    return a

i = 0
while i < 15:
    print(i, fib(i), max_fib)
    i += 1

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

A side-effecting operation could also be setting an instance variable inside an object. This is morally equivalent to the global variable example; the scope of the "external state" is just smaller.



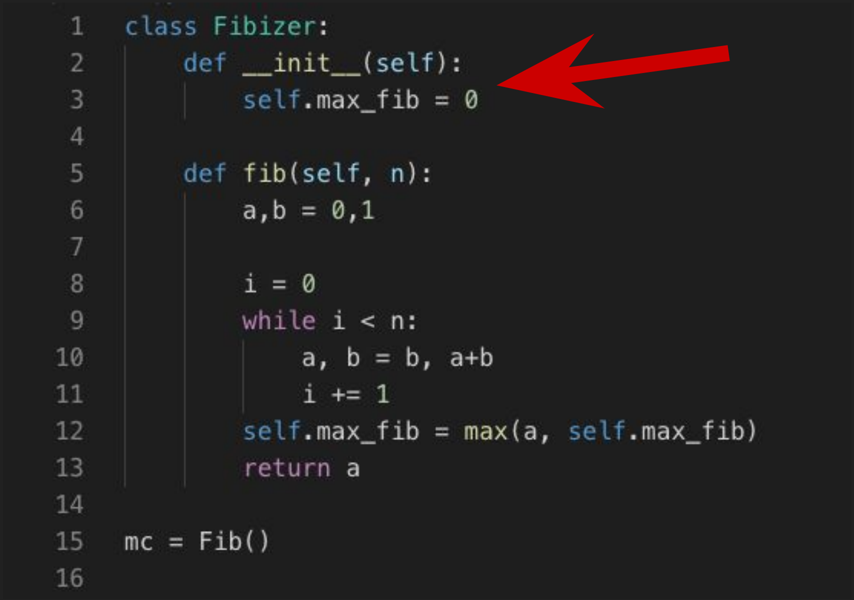
```python
class Fibizer:
    def __init__(self):
        self.max_fib = 0

    def fib(self, n):
        a,b = 0,1

        i = 0
        while i < n:
            a, b = b, a+b
            i += 1
        self.max_fib = max(a, self.max_fib)
        return a

mc = Fib()

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

<u>**Definition**</u>: A **side effect** is an external action that a function can do, apart from returning a value

- mutating an external variable
- Printing to the screen
- File or network I/O

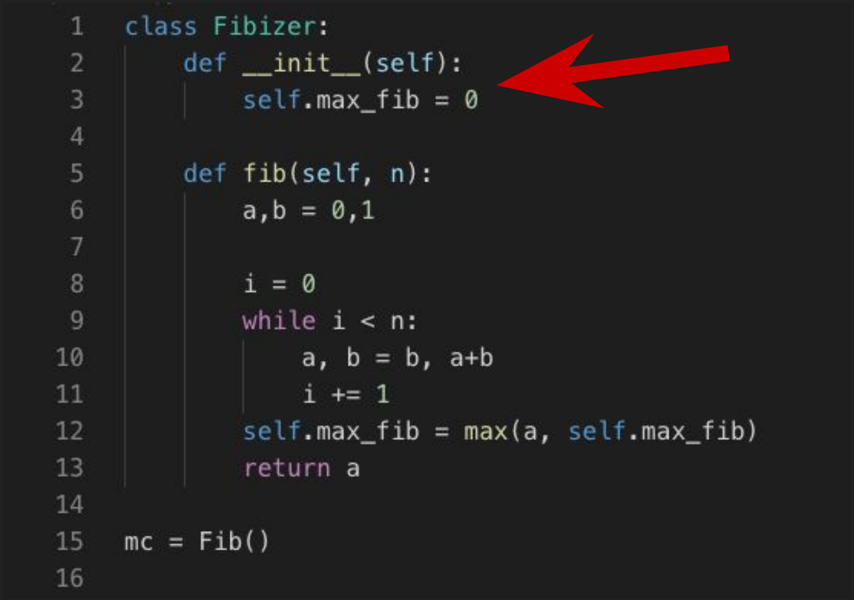


```python
class Fibizer:
    def __init__(self):
        self.max_fib = 0

    def fib(self, n):
        a,b = 0,1

        i = 0
        while i < n:
            a, b = b, a+b
            i += 1
        self.max_fib = max(a, self.max_fib)
        return a

mc = Fib()
```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

Definition: A function or method with a side-effect is called **impure**. If no side-effects occur, it is said to be **a pure function**.

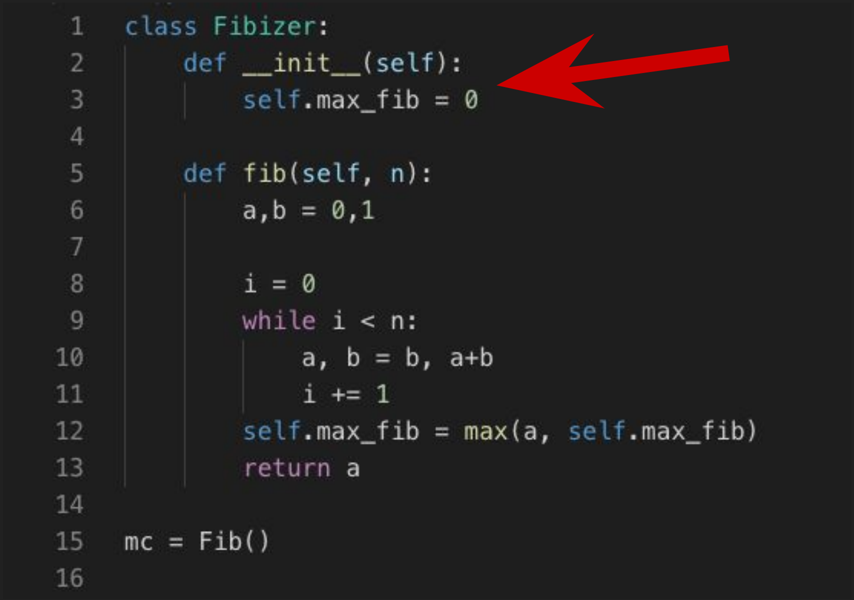


```python
class Fibizer:
    def __init__(self):
        self.max_fib = 0

    def fib(self, n):
        a,b = 0,1

        i = 0
        while i < n:
            a, b = b, a+b
            i += 1
        self.max_fib = max(a, self.max_fib)
        return a

mc = Fib()

```

1. The imperative programming model

Imperative programming is focused on **telling the computer what steps to perform and in what order**

A pure function's return value can only depend on its input values, since it by definition cannot depend on values of mutable state outside the function.

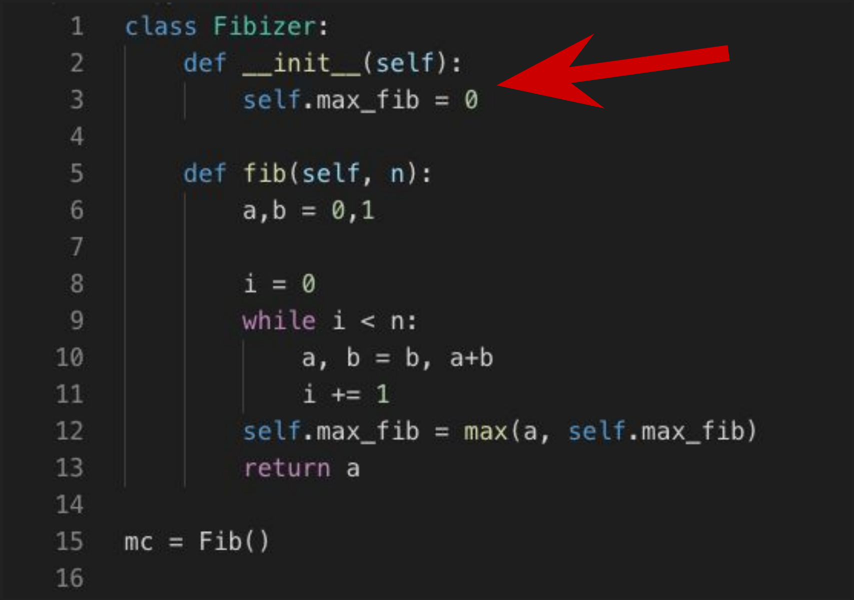


```python
class Fibizer:
    def __init__(self):
        self.max_fib = 0

    def fib(self, n):
        a,b = 0,1

        i = 0
        while i < n:
            a, b = b, a+b
            i += 1
        self.max_fib = max(a, self.max_fib)
        return a

mc = Fib()

```

1. The imperative programming model

The building blocks in imperative programming languages favour

- **statements** that **mutate state**
- **impure** or **side-effecting functions**



```python
class Fibizer:
    def __init__(self):
        self.max_fib = 0

    def fib(self, n):
        a,b = 0,1

        i = 0
        while i < n:
            a, b = b, a+b
            i += 1
        self.max_fib = max(a, self.max_fib)
        return a

mc = Fib()

```

TURING MACHINES

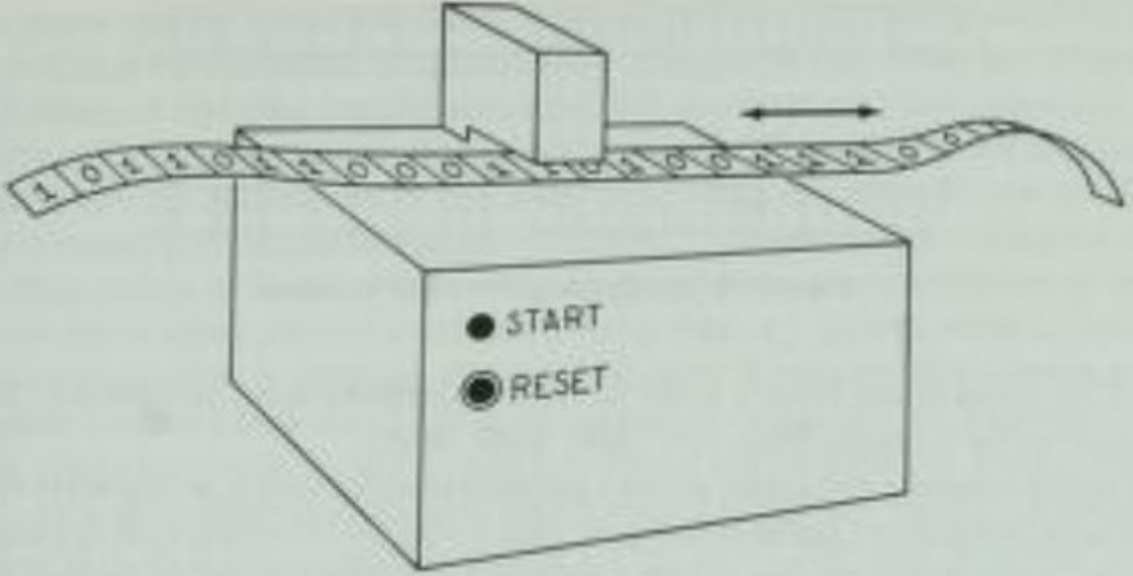


Figure 90 A Turing machine conceptualization

https://archive.org/details/turingomnibus61e00dewd

Difficulties with mutable state

Impure programs are **hard to maintain**:

"Do I fully understand under what circumstances this piece of mutable data needs to be changed?"

"Am I confident that this piece of mutable state is being set when it should be, and not being set when it shouldn't be?"

"If I get the answers to the above questions wrong, how long will it take before a bug manifests itself?"

Difficulties with mutable state

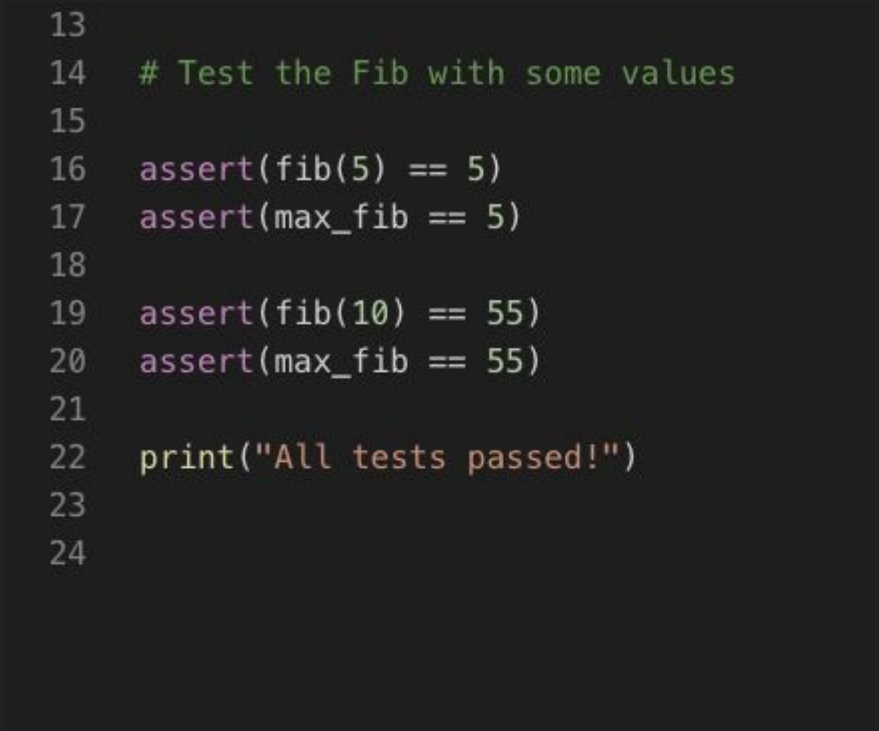
War story from industry: an ex-employer's codebase had a 2000-line C function called `void poke(void);` .

"Given the signature of poke, taking no arguments and returning no value, what can you conclude about what it does?"

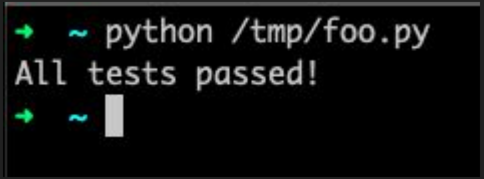
- It's a **pure function** that does nothing at all! (if only…)
- It's an **impure function** that modifies some mutable state, based on the current values of other pieces of state

Difficulties with mutable state

Impure programs are **hard to write tests for**:



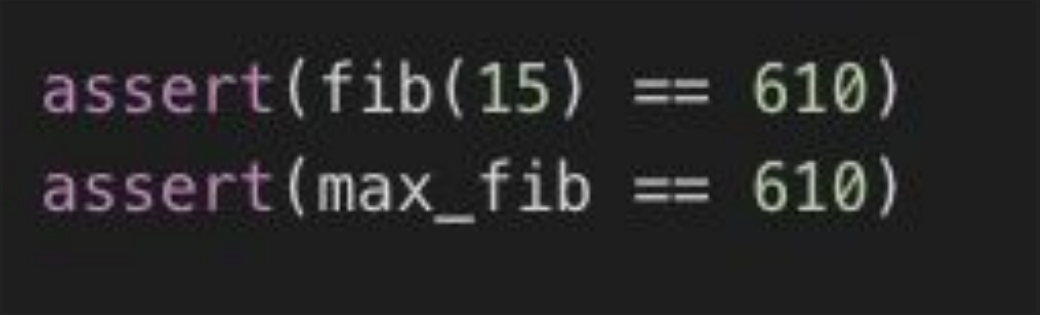
```

# Test the Fib with some values

assert(fib(5) == 5)
assert(max_fib == 5)

assert(fib(10) == 55)
assert(max_fib == 55)

print("All tests passed!")


```



```
➜  ~ python /tmp/foo.py
All tests passed!
➜  ~ 
```

Difficulties with mutable state

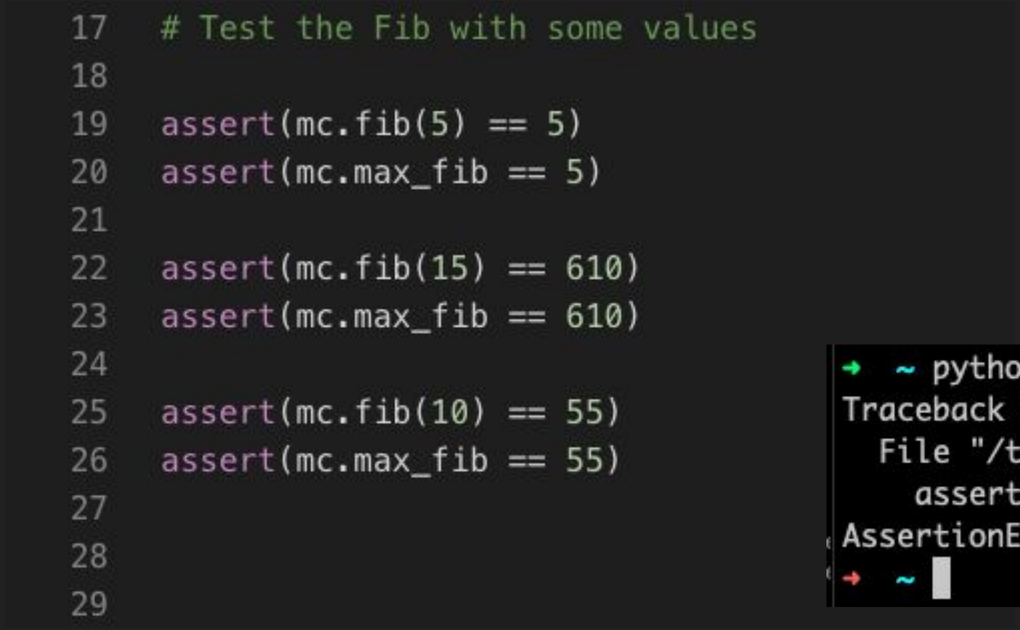
Impure programs are **hard to write tests for**:



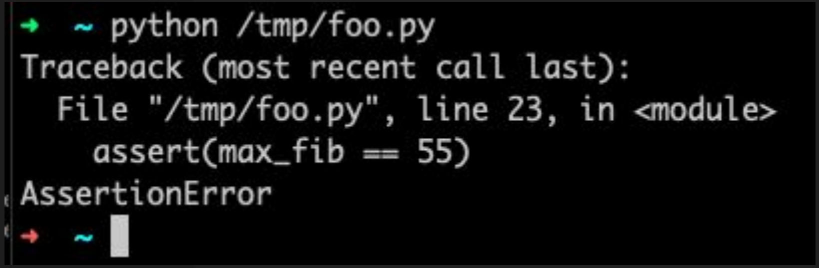
```
assert(fib(15) == 610)
assert(max_fib == 610)
```

Difficulties with mutable state

Impure programs are **hard to write tests for**:



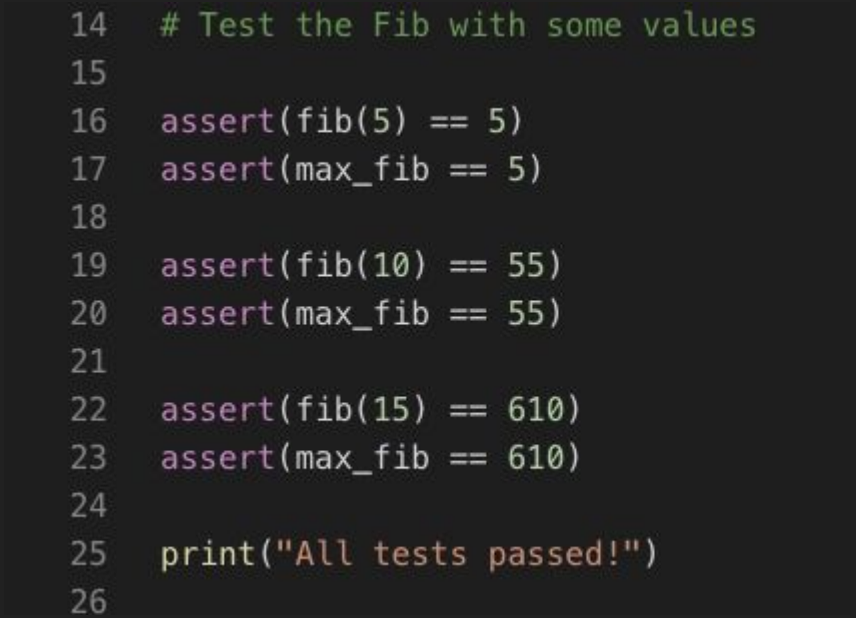
```
17  # Test the Fib with some values
18
19  assert(mc.fib(5) == 5)
20  assert(mc.max_fib == 5)
21
22  assert(mc.fib(15) == 610)
23  assert(mc.max_fib == 610)
24
25  assert(mc.fib(10) == 55)
26  assert(mc.max_fib == 55)
27
28
29
```



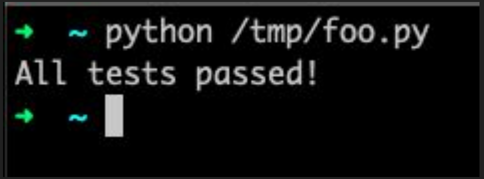
```
➜  ~ python /tmp/foo.py
Traceback (most recent call last):
  File "/tmp/foo.py", line 23, in <module>
    assert(max_fib == 55)
AssertionError
➜  ~
```

Difficulties with mutable state

Impure programs are **hard to write tests for**:



```python
# Test the Fib with some values

assert(fib(5) == 5)
assert(max_fib == 5)

assert(fib(10) == 55)
assert(max_fib == 55)

assert(fib(15) == 610)
assert(max_fib == 610)

print("All tests passed!")

```



```
➜  ~ python /tmp/foo.py
All tests passed!
➜  ~ 
```

Difficulties with mutable state

Impure programs are **hard to write tests for**:

Because fib() is an impure, side-effecting function, the order that the tests are run in depends on whether the tests pass. In industry, we call a test suite that sometimes passes or fails, depending on external state, a "flaky test".

Difficulties with mutable state

Impure programs are **hard to parallelize**

"How do I ensure that updates to this piece of mutable state are synchronized if multiple processors try to overwrite it at the same time?"

1. Imperative vs Functional programming

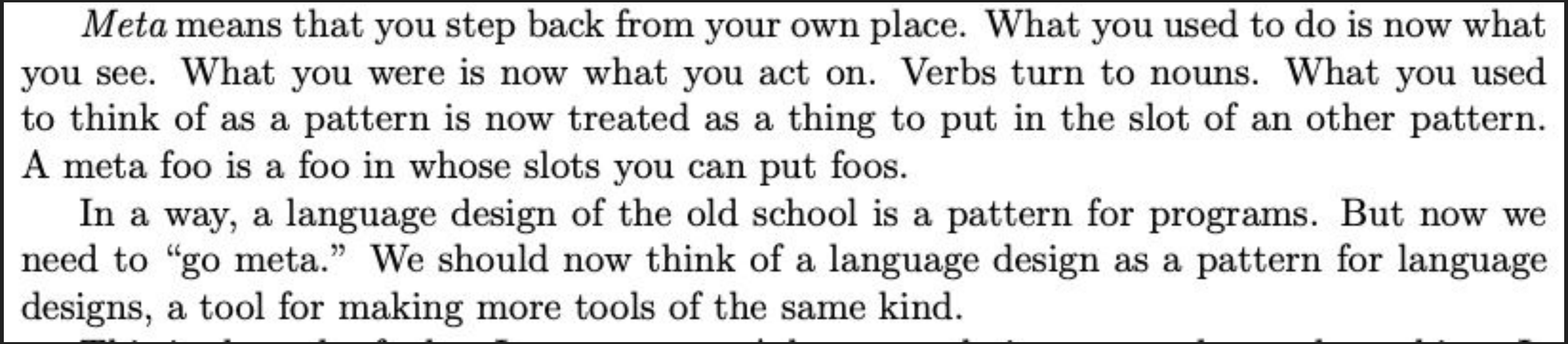
The building blocks in imperative programming languages favour

- **statements** that **mutate state**
- **impure** or **side-effecting functions**
  - **that operate on mutable state or objects**
  - **and compose (in OOP) via composition or inheritance**

The building blocks in **functional programming languages** favour

- **expressions** that **evaluate to values**
- **pure** or **side-effect free functions**
  - **that return new pieces of state or objects**
  - **and compose to make new, richer functions..!**

2. Metaprogramming: Modifying programs with programs



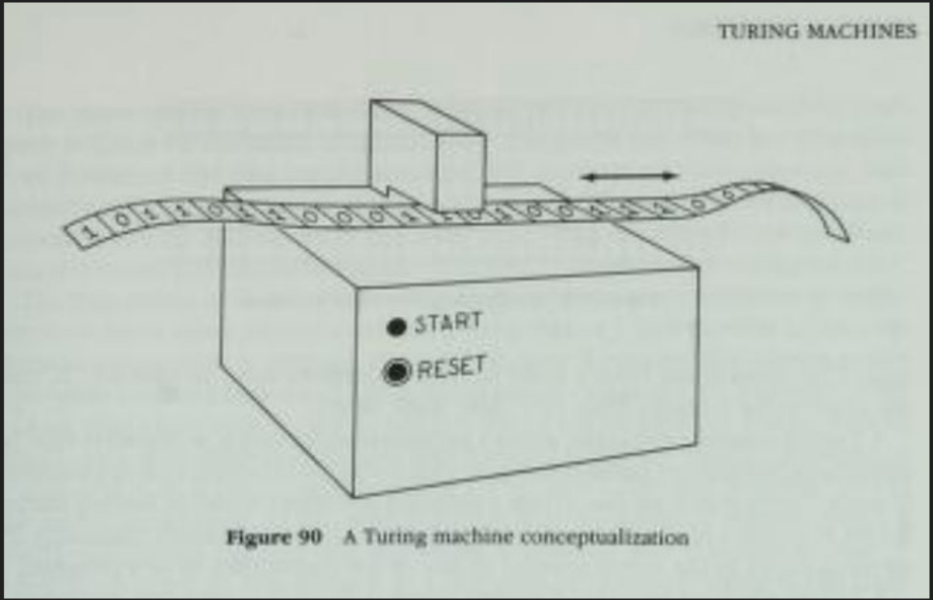
Meta means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of an other pattern. A meta foo is a foo in whose slots you can put foos.

In a way, a language design of the old school is a pattern for programs. But now we need to "go meta." We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind.

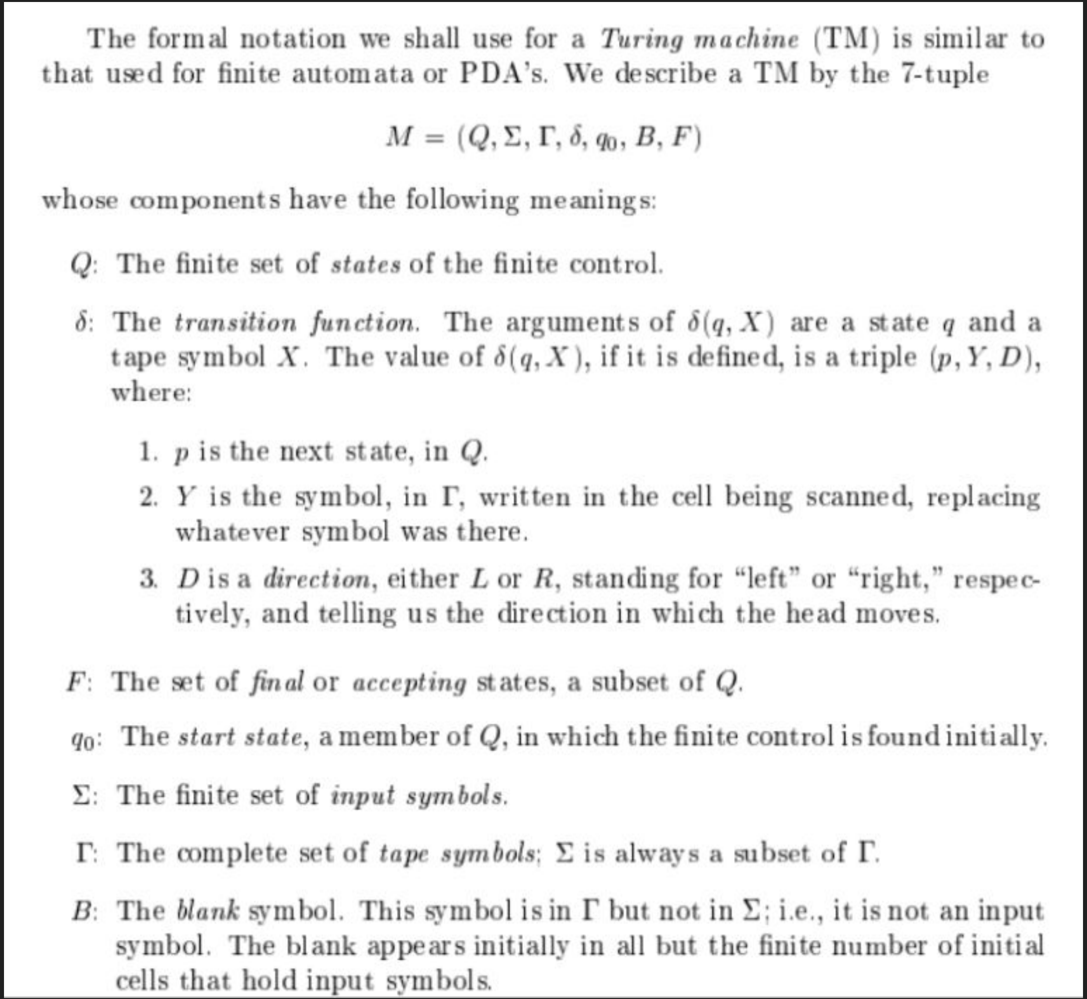
Guy Steele, "Growing A Language"

Another connection to Turing Machines...

In the Turing Machine model, the program modifies the data stored on its tape, but has no way to "reprogram" itself



https://archive.org/details/turingomnibus61e00dewd



The formal notation we shall use for a *Turing machine* (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

δ: The *transition function*. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

F: The set of *final* or *accepting* states, a subset of Q.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

Σ: The finite set of *input symbols*.

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

$\Bigg\}$ The machine's "**code**": components related to the Turing Machine's internal state machine

$\Bigg\}$ The machine's "**data**": components related to the state of the Turing Machine's tape

Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). Introduction to Automata Theory, Languages, and Computation (3rd ed.)., modified

2. Modifying programs with programs

Languages that inherit from the Turing Machine tradition treat program code and program data as completely separate

We will introduce **metaprogramming** in order to write programs that **consume a program and/or output another program**.

2. Modifying programs with programs

We can use metaprogramming to introduce new language features or build integrated mini-languages into an existing one

How would you implement "switch" in C or Java, if it wasn't already built into the language?

Whereas OOP-programmers think of building a program out of lots of small objects, metaprogrammers build programs out of lots of tiny languages!

3. An Introduction to Type Theory

3. An Introduction to Type Theory

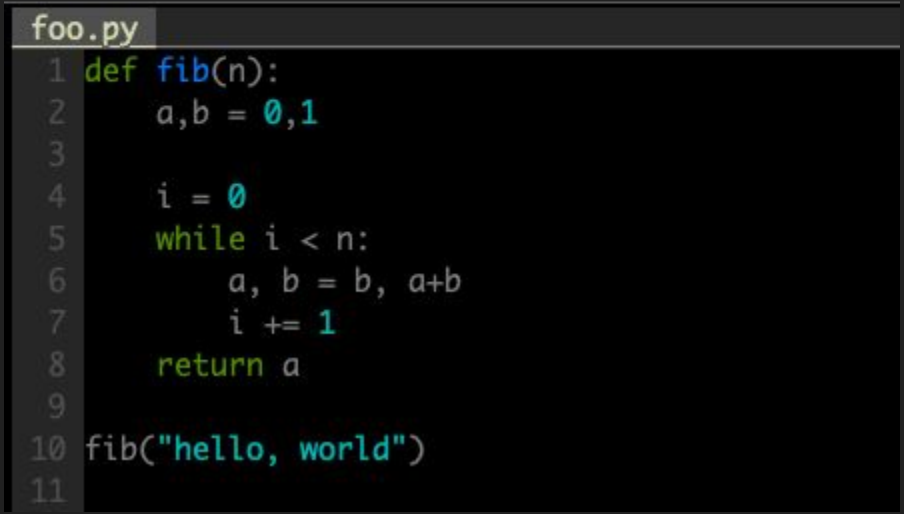
By now you will have cultivated an intuitive meaning of what a **type** is.

- A type is a "collection" of "things"
- A (pure) function with an argument of type A and a return type of B maps As to Bs

3. An Introduction to Type Theory

By now you will have cultivated an intuitive meaning of what a **type** is and in what ways the meaning of types affect a program's execution

In Python 2, this program results in an infinite loop...

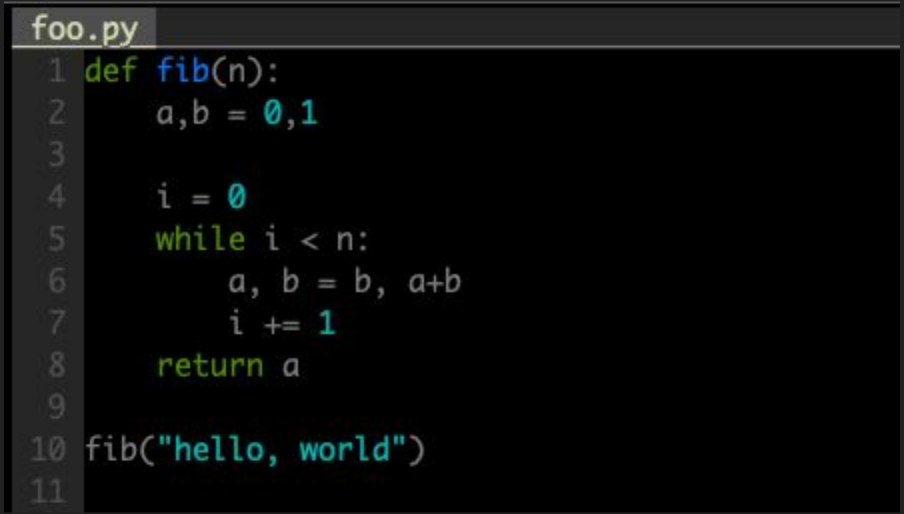


foo.py

```python
def fib(n):
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    return a

fib("hello, world")
```

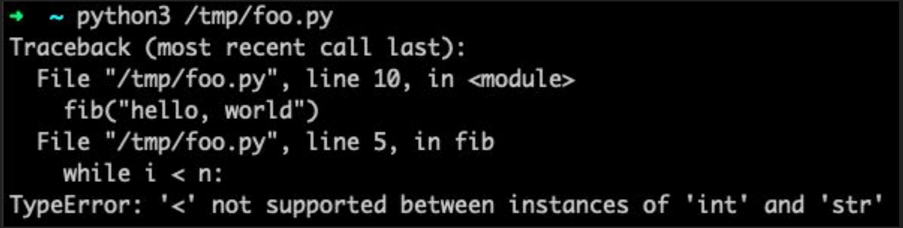
3. An Introduction to Type Theory

By now you will have cultivated an intuitive meaning of what a **type** is and in what ways the meaning of types affect a program's execution



```
foo.py
def fib(n):
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    return a

fib("hello, world")
```

In Python 3, this program at least throws an exception at runtime

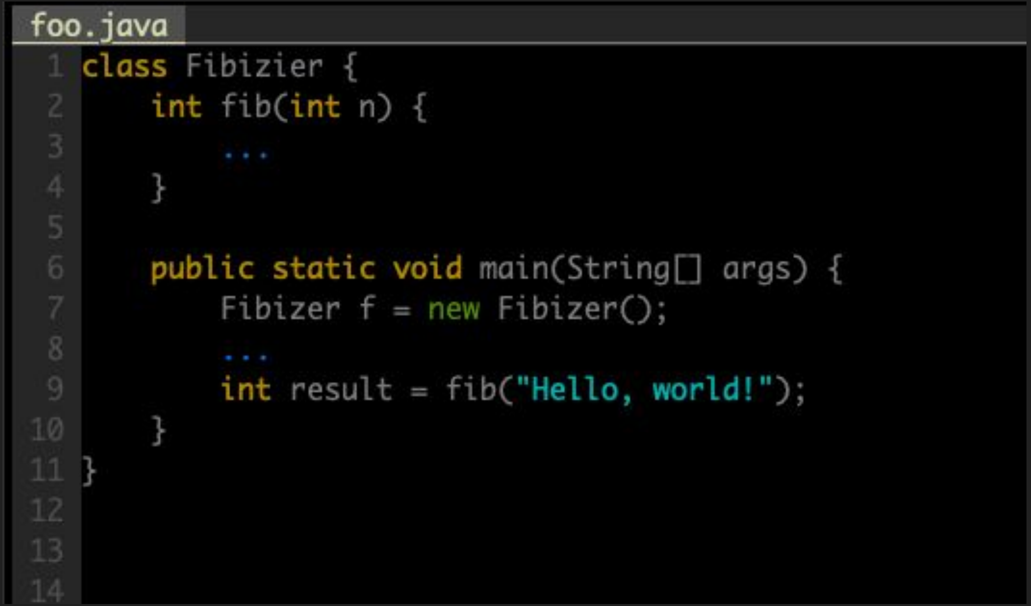


```
➜  ~ python3 /tmp/foo.py
Traceback (most recent call last):
  File "/tmp/foo.py", line 10, in <module>
    fib("hello, world")
  File "/tmp/foo.py", line 5, in fib
    while i < n:
TypeError: '<' not supported between instances of 'int' and 'str'
```

3. An Introduction to Type Theory

By now you will have cultivated an intuitive meaning of what a **type** is and in what ways the meaning of types affect a program's execution

...whereas languages with **static type systems** would reveal this programming error before the program is run.

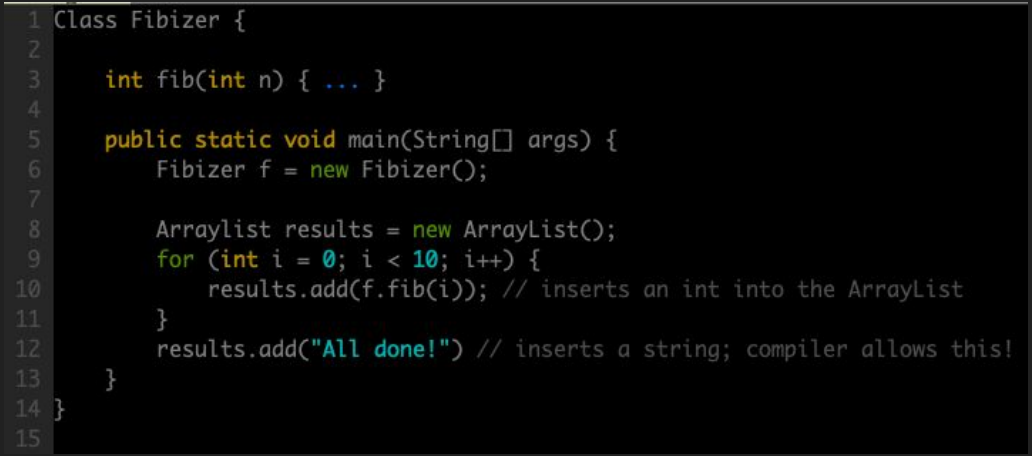

foo.java

```java
class Fibizier {
    int fib(int n) {
        ...
    }

    public static void main(String[] args) {
        Fibizer f = new Fibizer();
        ...
        int result = fib("Hello, world!");
    }
}
```

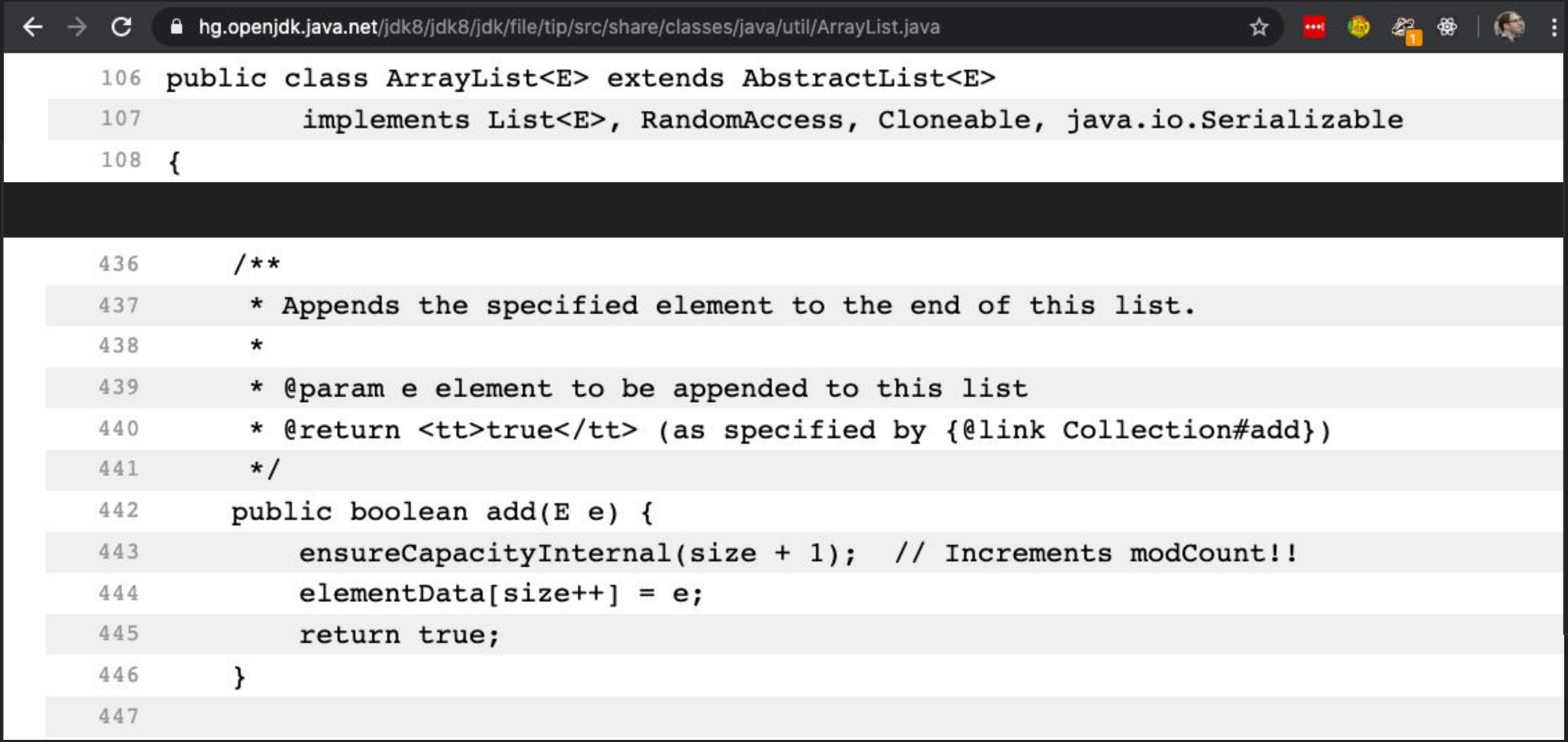
3. An Introduction to Type Theory

You have also seen how generics in Java allow **parameterizing a type** with a **type variable**.

Here, the compiler does not forbid us from inserting both ints and strings into the ArrayList.



```
Class Fibizer {

    int fib(int n) { ... }

    public static void main(String[] args) {
        Fibizer f = new Fibizer();

        Arraylist results = new ArrayList();
        for (int i = 0; i < 10; i++) {
            results.add(f.fib(i)); // inserts an int into the ArrayList
        }
        results.add("All done!") // inserts a string; compiler allows this!
    }
}

```

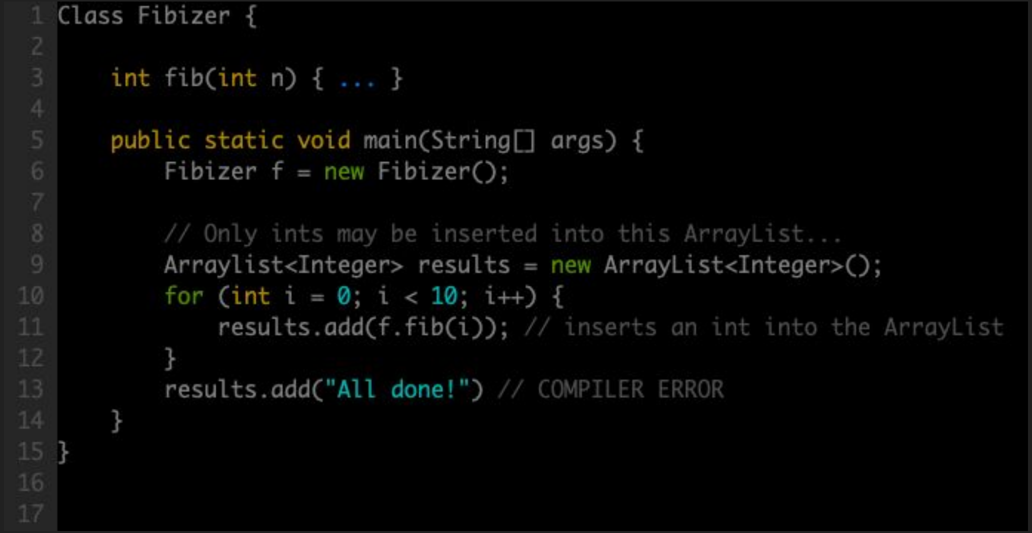

hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java

```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
```

```java
    /**
     * Appends the specified element to the end of this list.
     *
     * @param e element to be appended to this list
     * @return <tt>true</tt> (as specified by {@link Collection#add})
     */
    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }

```

3. An Introduction to Type Theory

With **parametric polymorphism**, a **type variable** can be used to **parameterize another type**.



```
Class Fibizer {

    int fib(int n) { ... }

    public static void main(String[] args) {
        Fibizer f = new Fibizer();

        // Only ints may be inserted into this ArrayList...
        Arraylist<Integer> results = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            results.add(f.fib(i)); // inserts an int into the ArrayList
        }
        results.add("All done!") // COMPILER ERROR
    }
}
```

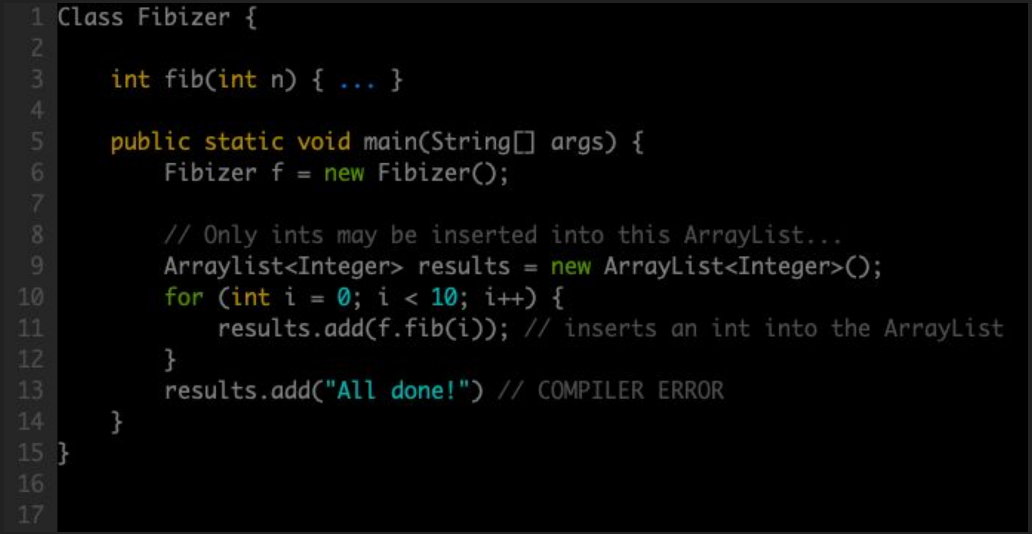
Note the similarity between:

- an object constructor that takes one argument
- a **type constructor** that takes one type

3. An Introduction to Type Theory

With **parametric polymorphism**, a **type variable** can be used to **parameterize another type**.

Thus far, you have seen how a more expressive type system further constrains the space of valid programs



```
Class Fibizer {

    int fib(int n) { ... }

    public static void main(String[] args) {
        Fibizer f = new Fibizer();

        // Only ints may be inserted into this ArrayList...
        Arraylist<Integer> results = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            results.add(f.fib(i)); // inserts an int into the ArrayList
        }
        results.add("All done!") // COMPILER ERROR
    }
}


```

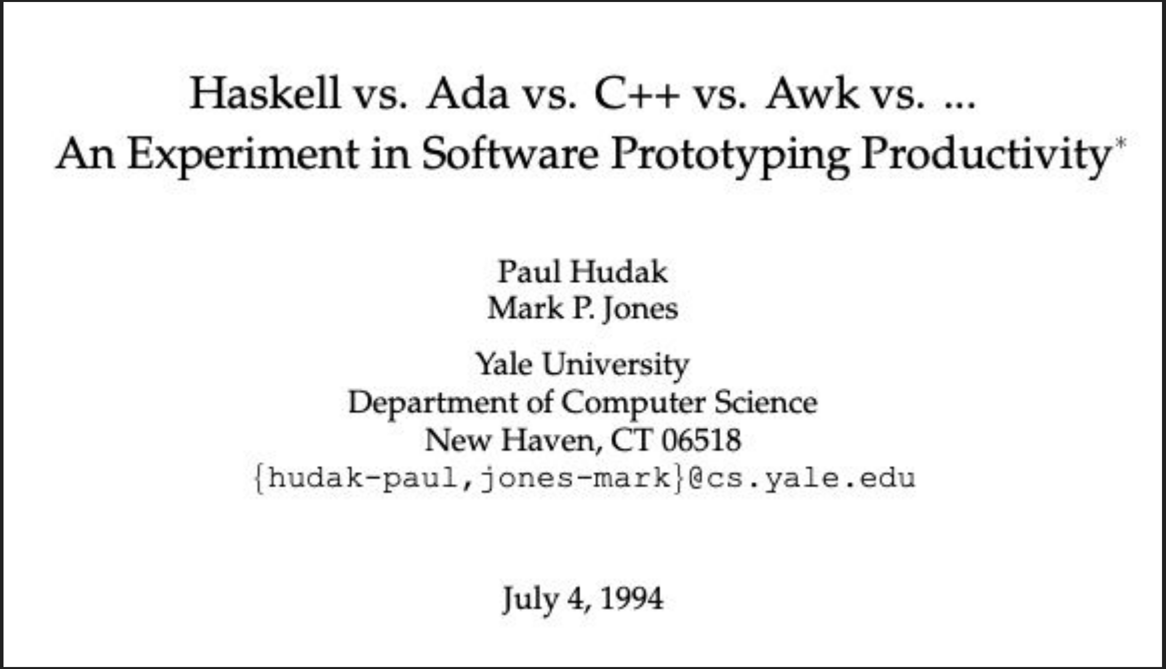
3. An Introduction to Type Theory

In this class, we will introduce more powerful and expressive type systems than what you have previously seen.

This will allow us to use type systems not just for catching programming errors but as a formal system to build abstractions within our programs' design, that let us write powerful programs without things like mutable state.

We will also think about the ways in which a language's type system is *itself a programming language.*

Case study in functional programming


Haskell vs. Ada vs. C++ vs. Awk vs. ...

An Experiment in Software Prototyping Productivity*

Paul Hudak
Mark P. Jones

Yale University
Department of Computer Science
New Haven, CT 06518
{hudak-paul,jones-mark}@cs.yale.edu

July 4, 1994

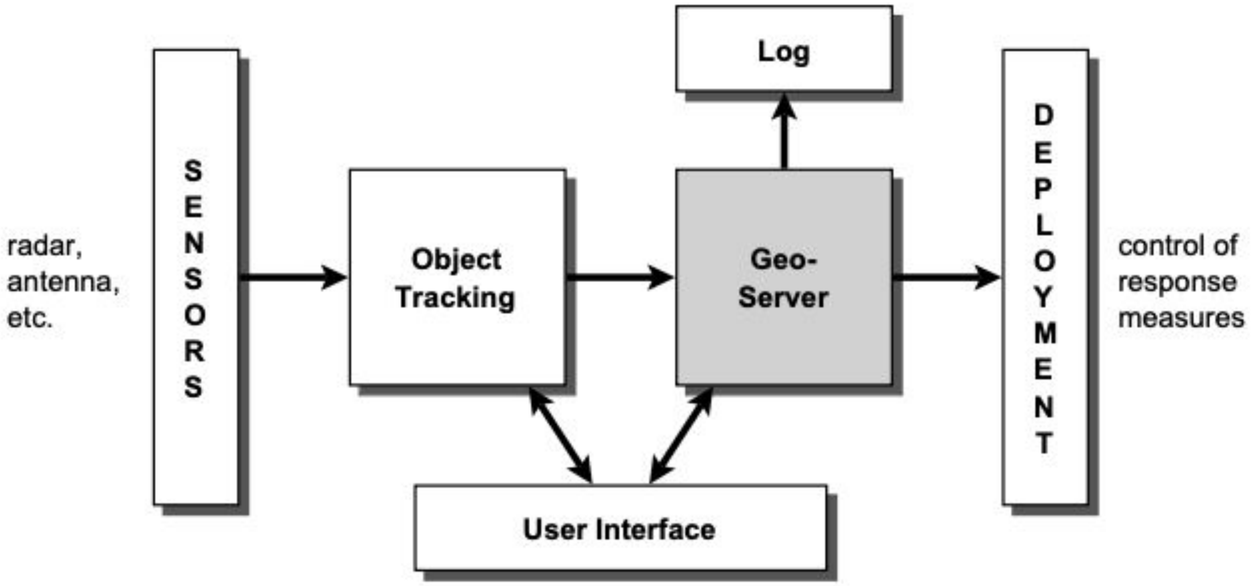


Figure 1: Simplified Aegis Weapons Systems Diagram

The input to the geo-server consists of data that conveys the positions of various ships, airplanes, and other objects on the globe; the output consists of relationships between these objects as computed by the geo-server. To gain some intuition about the functionality of the geo-server, it is helpful to observe a typical input/output pattern. The input data is best conveyed using a *map*, for example as shown in Figure 2. In this diagram:

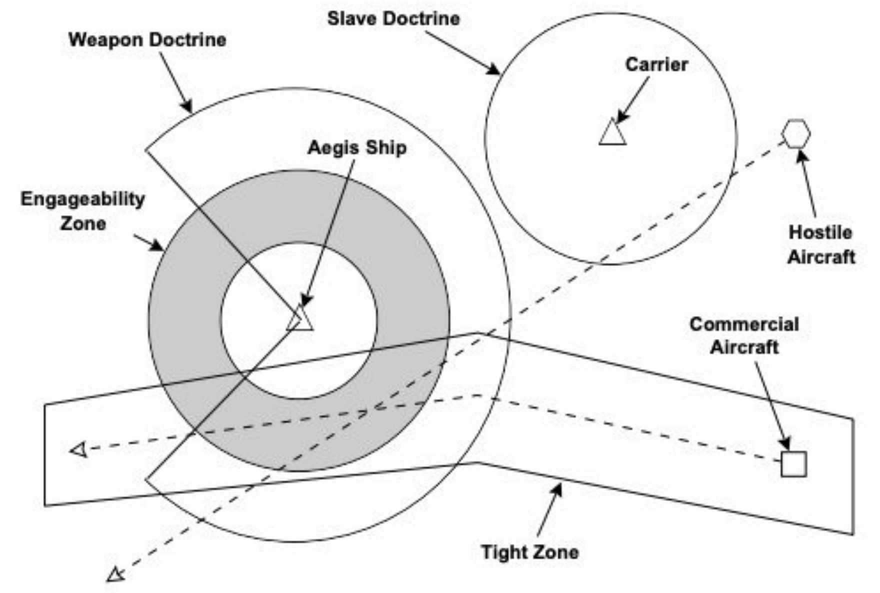
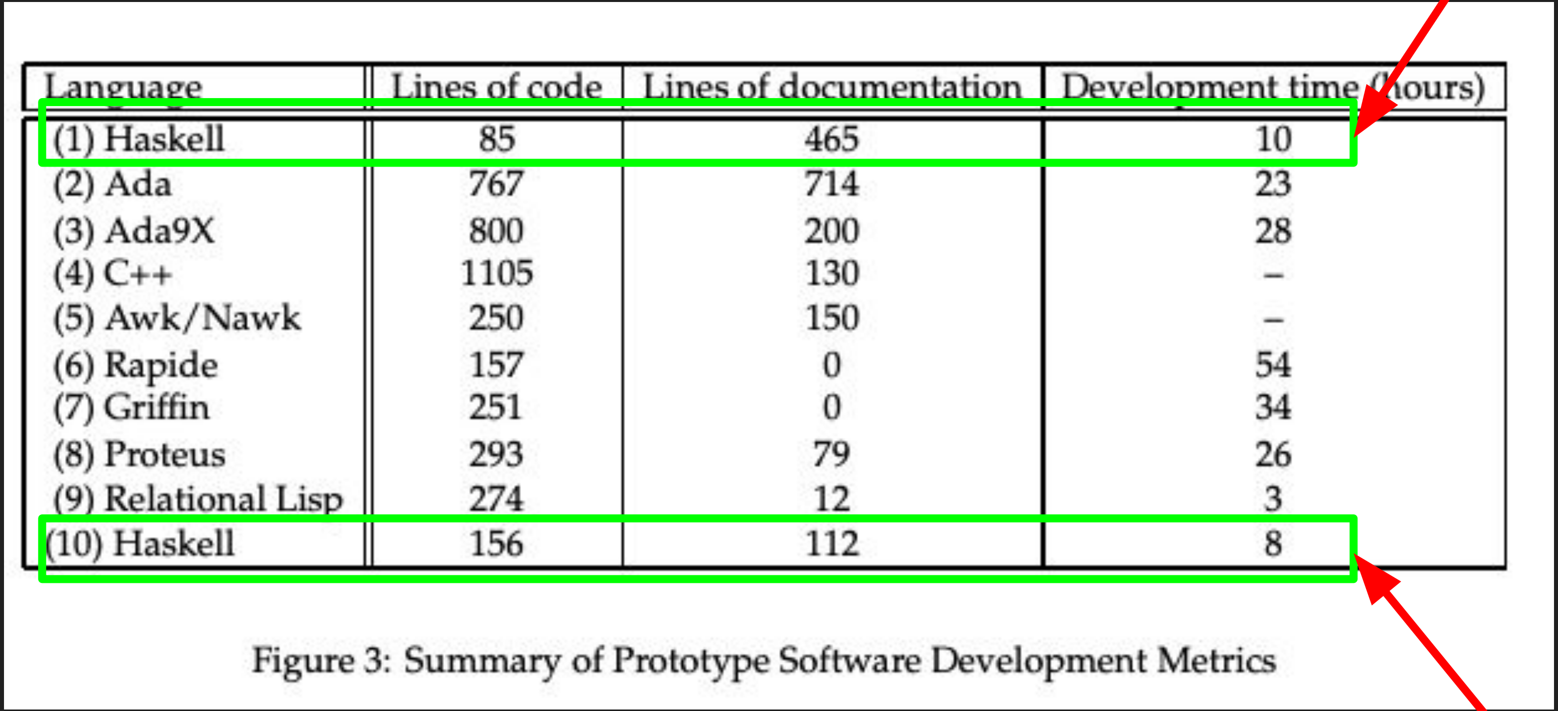


Figure 2: Geo-Server Input Data

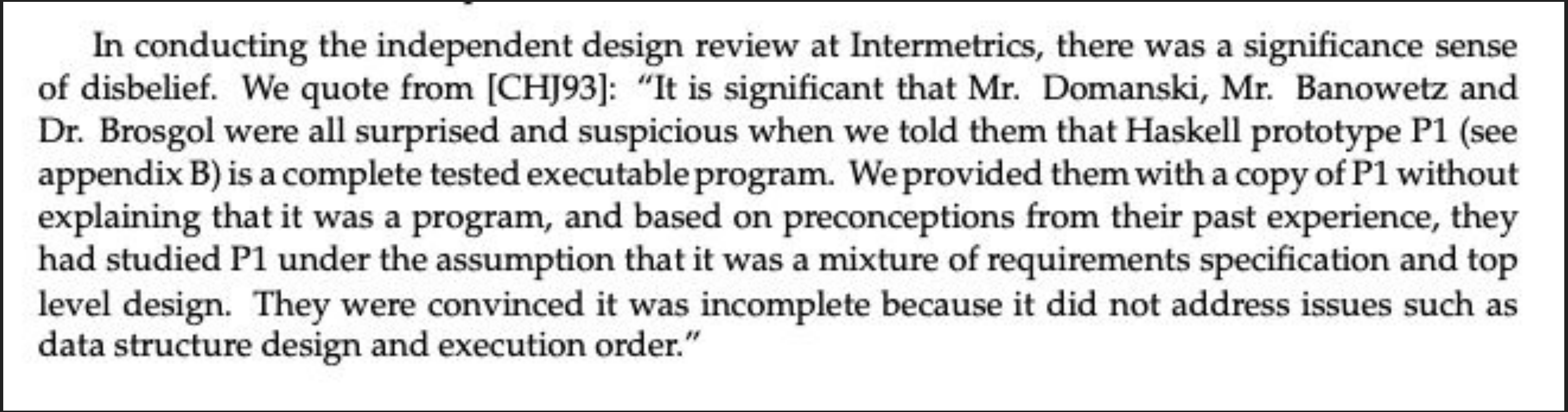
Version written by a professional Haskell programmer



Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Figure 3: Summary of Prototype Software Development Metrics

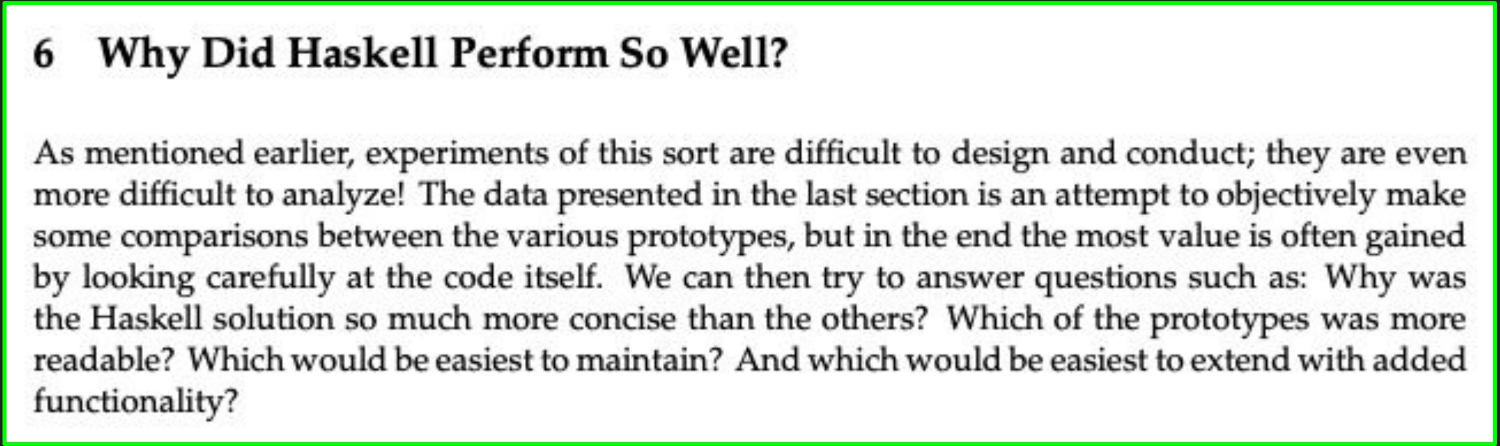
Version written by a college grad with a week of experience writing Haskell



In conducting the independent design review at Intermetrics, there was a significance sense of disbelief. We quote from [CHJ93]: "It is significant that Mr. Domanski, Mr. Banowetz and Dr. Brosgol were all surprised and suspicious when we told them that Haskell prototype P1 (see appendix B) is a complete tested executable program. We provided them with a copy of P1 without explaining that it was a program, and based on preconceptions from their past experience, they had studied P1 under the assumption that it was a mixture of requirements specification and top level design. They were convinced it was incomplete because it did not address issues such as data structure design and execution order."

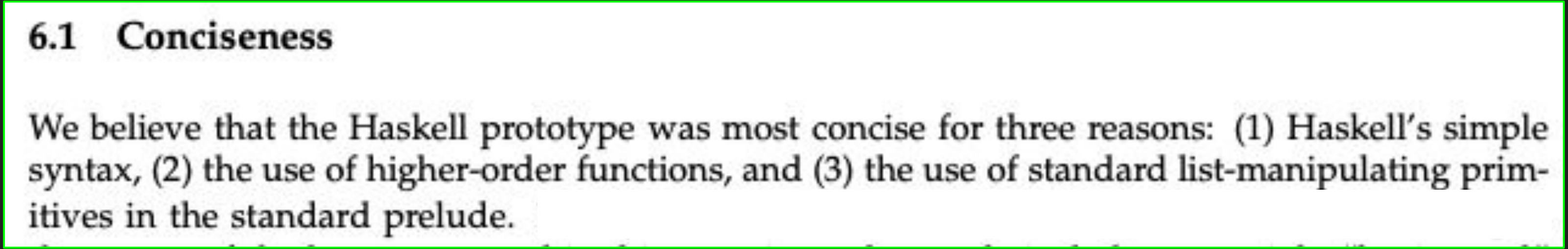
6 Why Did Haskell Perform So Well?

As mentioned earlier, experiments of this sort are difficult to design and conduct; they are even more difficult to analyze! The data presented in the last section is an attempt to objectively make some comparisons between the various prototypes, but in the end the most value is often gained by looking carefully at the code itself. We can then try to answer questions such as: Why was the Haskell solution so much more concise than the others? Which of the prototypes was more readable? Which would be easiest to maintain? And which would be easiest to extend with added functionality?



6 Why Did Haskell Perform So Well?

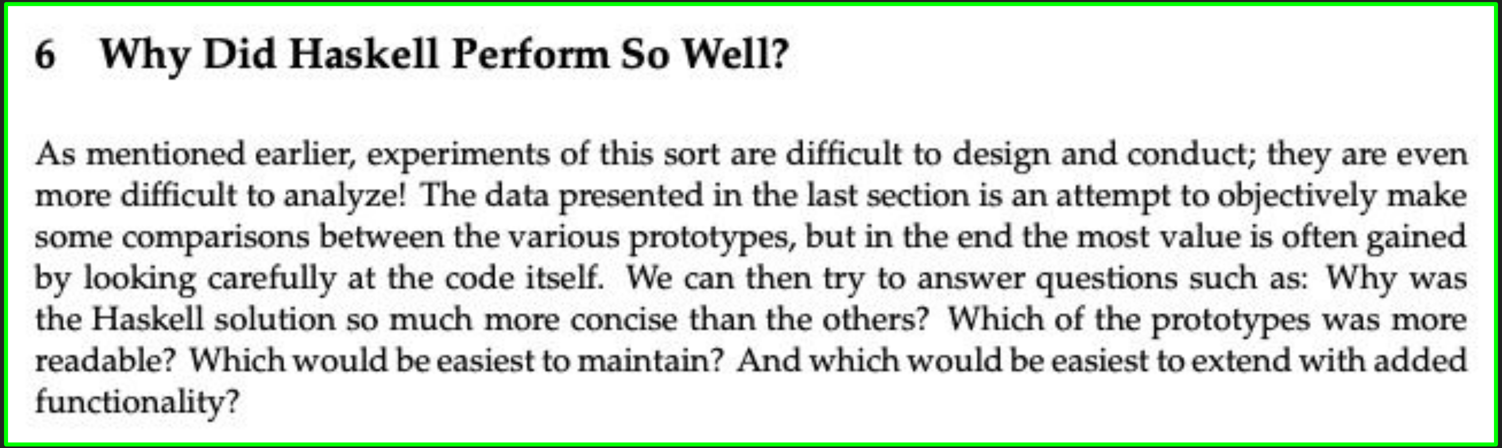
As mentioned earlier, experiments of this sort are difficult to design and conduct; they are even more difficult to analyze! The data presented in the last section is an attempt to objectively make some comparisons between the various prototypes, but in the end the most value is often gained by looking carefully at the code itself. We can then try to answer questions such as: Why was the Haskell solution so much more concise than the others? Which of the prototypes was more readable? Which would be easiest to maintain? And which would be easiest to extend with added functionality?



6.1 Conciseness

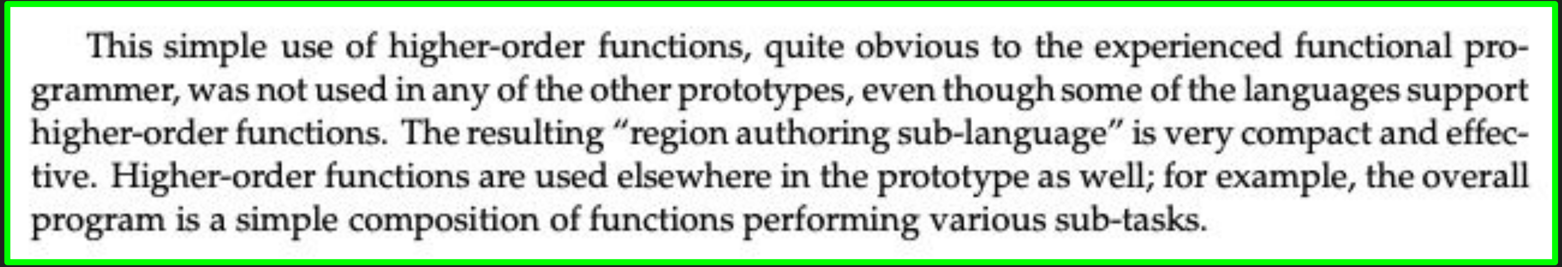
We believe that the Haskell prototype was most concise for three reasons: (1) Haskell's simple syntax, (2) the use of higher-order functions, and (3) the use of standard list-manipulating primitives in the standard prelude.

...part 1 of this course (the FP paradigm)...



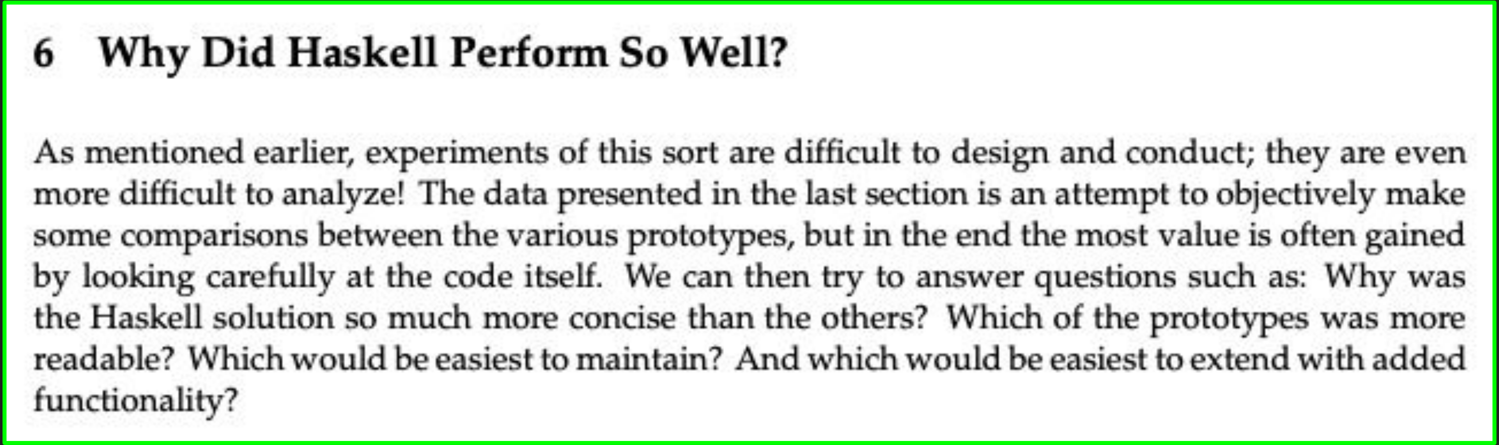
6 Why Did Haskell Perform So Well?

As mentioned earlier, experiments of this sort are difficult to design and conduct; they are even more difficult to analyze! The data presented in the last section is an attempt to objectively make some comparisons between the various prototypes, but in the end the most value is often gained by looking carefully at the code itself. We can then try to answer questions such as: Why was the Haskell solution so much more concise than the others? Which of the prototypes was more readable? Which would be easiest to maintain? And which would be easiest to extend with added functionality?



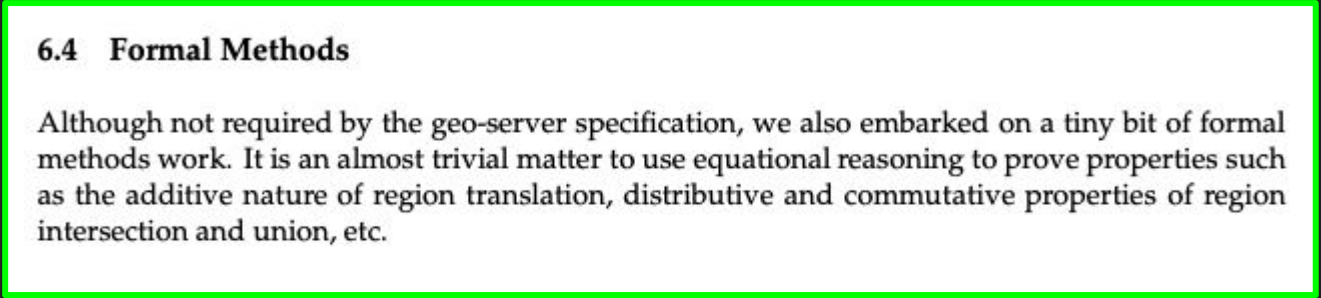
This simple use of higher-order functions, quite obvious to the experienced functional programmer, was not used in any of the other prototypes, even though some of the languages support higher-order functions. The resulting "region authoring sub-language" is very compact and effective. Higher-order functions are used elsewhere in the prototype as well; for example, the overall program is a simple composition of functions performing various sub-tasks.

...part 1 & 2 of this course (metaprogramming a sub-language)...



6 Why Did Haskell Perform So Well?

As mentioned earlier, experiments of this sort are difficult to design and conduct; they are even more difficult to analyze! The data presented in the last section is an attempt to objectively make some comparisons between the various prototypes, but in the end the most value is often gained by looking carefully at the code itself. We can then try to answer questions such as: Why was the Haskell solution so much more concise than the others? Which of the prototypes was more readable? Which would be easiest to maintain? And which would be easiest to extend with added functionality?



6.4 Formal Methods

Although not required by the geo-server specification, we also embarked on a tiny bit of formal methods work. It is an almost trivial matter to use equational reasoning to prove properties such as the additive nature of region translation, distributive and commutative properties of region intersection and union, etc.

...part 3 of this course (type theory and formal reasoning)...

Next time:

The syntax, semantics, and evaluation of programming languages