# Programming Paradigms

Runxi Yu

*Colloquium Cognitionis Callidum* 2023

August 6, 2023

# Copyright License

# External Copyright

# Fun fact

Me attempting to decide on a topic:

- Epistemology and the limitations of science and formal logic
- Elliptic-curve cryptography: Curve25519, Curve448
- Memory management: Manual, garbage collection, reference counting, borrow checking, use-after-free, and memory leaks
- Liberal feminism and trans-exclusive radical feminism
- Cybersecurity: Why school networks are always vulnerable (no)
- The federal court system of the United States
- **Programming paradigms**

Also, `https://git.andrewyu.org/andrew/school/ccc.git/plain` has the presentation and related files.

# Expected Knowledge

This presentation expects a little bit of knowledge in programming. You should know basic operations in at least one general-purpose programming language (no, not HTML).

You don't have to understand copmilers and interpreter design, or understand a plethora of languages, though the latter may be helpful.

If you understand and use the Lisp family of languages, or if you have sufficient experience in Haskell using the `IO` type correctly, you probably don't need to listen to this.

# Contents

# Factorials

We'll be using the factorial function (classic) as examples.

Let $n \in \mathbb{N}$,

$$\mathsf{fac}(n) = n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

(Yes, math in sans-serif is cringe)

# Programming Paradigms?

Look at the navigation sidebar.

# Programming Paradigms?

Look at the navigation sidebar.

**Why haven't I ever heard of this?**

# Programming Paradigms?

Look at the navigation sidebar.

## Why haven't I ever heard of this?

You probably thought that imperative was the only programming paradigm, without actually knowing any others ...

# Programming Paradigms?

Look at the navigation sidebar.

## Why haven't I ever heard of this?

You probably thought that imperative was the only programming paradigm, without actually knowing any others ...

I mean it is quite common

# Some "Normal" C Code

```c
static int seed_with_urandom(void) {
    unsigned int seed;
    int fd;
    fd = open("/dev/urandom", O_RDONLY);
    if (fd >= 0) {
        if (read(fd, &seed, sizeof(seed)) ==
                    sizeof(seed)) {
            close(fd);
            srand(seed);
            return 1;
        }
        close(fd);
    }
    return 0;
}
```

# Some "Normal" C Code

```c
static int seed_with_urandom(void) {
    unsigned int seed;
    int fd;
    fd = open("/dev/urandom", O_RDONLY);
    if (fd >= 0) {
        if (read(fd, &seed, sizeof(seed)) ==
                    sizeof(seed)) {
            close(fd);
            srand(seed);
            return 1;
        }
        close(fd);
    }
    return 0;
}
```

**Each statement modifies the program's state.**

# A Mini Bootloader

The following X86 assmebly code, which is part of a master boot record, loads 16 kilobytes of data immediately following the MBR from disk and executes it. X86 assembly is a good example of very imperative programming.

```
mov ah, 0x02      ; BIOS read disk
mov al, 32        ; 32 sectors
mov ch, 0         ; Cylinder 0
mov cl, 2         ; Sector 2
mov dh, 0         ; Head 0
mov dl, 0x80      ; First hard disk
mov bx, 0x7E00    ; RAM Destination
int 0x13
jc derr
jmp 0x0000:0x7E00
```

# The Imperative Paradigm

- The computer executes statements one-by-one
- Each instruction changes the state
  (Which could be memory, CPU registers, etc.)
- We may `jmp`/`goto` to another instruction
- We arrive at a final result from the final state

# The Imperative Paradigm

- The computer executes statements one-by-one
- Each instruction changes the state
  (Which could be memory, CPU registers, etc.)
- We may `jmp`/`goto` to another instruction
- We arrive at a final result from the final state

But there's a problem with `goto` ...

# Assembly Control Flows

```
_start:
    cmp esi, 1
    je .yes
    jmp .no
.yes:
    ; do A
    jmp .done
.no:
    ; do B
.done:
    ; do C
```

```
if (var == 1) {
    // do A
}
else {
    // do B
}
// do C
```

# goto Considered Harmful?


Figure: XKCD 292 "goto" (CC BY-NC 2.5)

(`jmp` and `goto` are basically the same, and the dinosaur is Goto Dengo in the novel *Cryptonomicon*)

# goto Leads to Spaghetti Code

Which of these factorial functions is more readable?

```
if (n >= 0)
    goto good;
errno = EINVAL;
return -1;
good:
    int r = 1;
loop:
    if (n > 1)
        goto yes;
    goto end;
yes:
    r *= n;
    n--;
    goto loop;
end:
    return r;
```

```
if (n < 0) {
    errno = EINVAL;
    return -1;
}
int r = 1;
for (; n > 1; n--) {
    r *= i;
}
return r;
```

# **Structured** Imperatives

- Code can be {grouped} into blocks.
- There are loops!
- No more spaghetti gotos.

```
int factorial(int n) {
    if (n < 0) {
        errno = EINVAL;
        return -1;
    }
    int r = 1;
    for (; n > 1; n--) {
        r *= i;
    }
    return r;
}
```

# Procedures

We could abstract commonly-used procedures into ... procedures!

```
char s[] = "Hello";
write(1, s, strlen(s));
write(1, "\n", 1);

char s[] = "Hey there";
write(1, s, strlen(s));
write(1, "\n", 1);
```

```
void puts(char s[]) {
    write(1, s, strlen(s));
    write(1, "\n", 1);
}

puts("Hello");
puts("Hey there");
```

(Please don't actually use this code which doesn't properly handle errors, but this illustrates the point)

# Functions

We could abstract commonly-used procedures that give us an output into ... functions!

```c
int r, n;

r = 1;
for (n = 5; n > 1; n--) {
    r *= i;
}
printf("%d", r);

r = 1;
for (n = 10; n > 1; n--) {
    r *= i;
}
printf("%d", r);
```

```c
int factorial(int n) {
    if (n < 0) {
        errno = EINVAL;
        return -1;
    }
    int r = 1;
    for (; n > 1; n--) {
        r *= i;
    }
    return r;
}

printf("%d", factorial(5));
printf("%d", factorial(10));
```

# Why Procedural?

Because we don't want to maintain five separate complicated procedures in different parts of the program.

Also, it saves space, because your object code would only have one copy of the procedure. (But you need to jump between functions with call and return, which might be eversoslightly slower.)

# Object-oriented???

I'm not very qualified to talk about this ...

## Object-oriented???

I'm not very qualified to talk about this ...

# Anyone?

# Smalltalk

A very ~~small~~pure object-oriented language

```
Transcript show: 'Hello, world!'.
"Double quotes for comments... strange"
```

# Smalltalk

## A very ~~small~~pure object-oriented language

```
Transcript show: 'Hello, world!'.
"Double quotes for comments... strange"
```

Seriously?

# Smalltalk

## A very ~~small~~pure object-oriented language

```
Transcript show: 'Hello, world!'.
"Double quotes for comments... strange"
```

Seriously? Yes.

- Grabs the `Transcript` object ...
- Sends a message called `show` to it ...
- With the argument `'Hello, world!'`

# Classes and objects

```
"Create a 'Dog' class that accepts 'bark'"
Object subclass: Dog [
    Dog class >> new [
        ^ super new
    ]

    Dog >> bark [
        Transcript show: 'Woof!'; cr.
    ]
]

"Create a new object 'dog' of the class 'Dog'"
dog := Dog new.

"Send the message 'bark' to the object 'dog'"
dog bark.
```

# So, what is OOP?

Smalltalk as the iconic OOP language:

- Everything is an object, objects are "instances" of classes
- Classes are descriptions of classes of objects
  - Defines the set of messages that its objects respond to
  - Defines variables contained in classes (not accessible from outside)
- Classes can inherit stuff from other classes when being created

# Criticism

John Ousterhout Scripting, IEEE Computer, March 1998:

> *Implementation inheritance causes the same intertwining and brittleness that have been observed when goto statements are overused. As a result, OO systems often suffer from complexity and lack of reuse.*

Joe Armstrong:

> *The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.*

# My Problems with OO

- Private state that's hidden from the programmer
  (The direct opposite of functional programming, sad)
- Memory leaks due to references in the private state
  (Though I guess it's better than use-after-free)
- Why even mix functions and data?
- It's just hard to refactor code with, by experience

# Functional Programming

- Programs are representations by applying and composing functions
  Functions are just variables, you can pass them around
- Makes it feel closer to pure math
- Generally uses higher-order functions, recursion, etc
- Functions do not have side effects in a purely functional programming language

# Haskell Examples

```haskell
-- Factorial by recursion
fac 0 = 1
fac n = n * fac (n - 1)

-- Factorial by higher-order
fac n = foldl (*) 1 [1..n]

-- Factorial by cheating :)
fac n = product [1..n]
```

# Monads

A monad is just a monoid in the category of endofunctors???

# Monads

A monad is just a monoid in the category of endofunctors???
Monoids, okay, just semigroups with a identity. What's an endofunctor???

# Monads

A monad is just a monoid in the category of endofunctors???
Monoids, okay, just semigroups with a identity. What's an endofunctor???

If anyone here actually know how to explain monads, please do :)

# Monads

A monad is just a monoid in the category of endofunctors???
Monoids, okay, just semigroups with a identity. What's an endofunctor???

If anyone here actually know how to explain monads, please do :)

I only know why they're useful: Abstracting away imperative control flow in a referentially transparent-ish way. (Otherwise you won't be able to actually do anything in pure functional programming.)

# Why go through the hassle?

Because **mathematics**. (Yeah . . . )

# Why go through the hassle?

Because mathematics. (Yeah . . . )

- Easier to formally verify the program to be correct
  Not to be confused with testing
- More familiar representation of objects
- Better concurrency
- Easier to maintain code, encouraging modularity

# Lisp: Meta-programming

What's so great about Lisp? Why is it infrequently used (unless if you use Emacs)?

# Lisp: Meta-programming

What's so great about Lisp? Why is it infrequently used (unless if you use Emacs)?

- Lisp is good because it has a very minimal, simple, regular syntax.

# Lisp: Meta-programming

What's so great about Lisp? Why is it infrequently used (unless if you use Emacs)?

- Lisp is good because it has a very minimal, simple, regular syntax.
- Lisp is bad because it has a very minimal, simple, regular syntax.

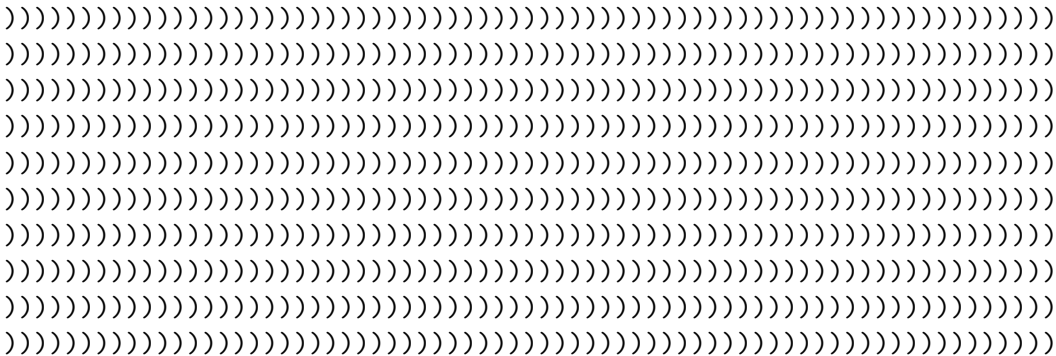
# Lisp's S-expressions

```
(defun main ()
  (print
    (let ((input-table (cl-csv:read-csv #P"tags.csv")))
      (mapcar
        (lambda (row) (let ((new-row (copy-tree row)))
                        (setf (fourth new-row)
                              (mapcar (lambda (s)
                                        (string-trim " " s))
                                      (comma-split (fourth new-row))))
                        (setf (third new-row)
                              (parse-float:parse-float
                                (third new-row)))
                        new-row))
        input-table))))
```

# Lisp Macros

- S-expressions are a simple, uniform syntax
- It's very easy to create macros that handle a syntax tree
  If you want to, you could create macros that adds control flow that doesn't exist in default Lisp. Maybe introduce a quick notations for Monads while you're at it.
- Much better than C preprocessor macros that are literally just string manipulation, much more robust

# The problem with S-expressions



```
(to (the (normal (programmer))) (is this (very (readable))))
(setq readability 1000)
```

# Example . . .

```
(defmacro a (&body b)
  `(progn ,@(mapcar (lambda (c) `(print ,c)) b) ,@(mapcar (lambda (c) `(setq ,c (random 100))) b)))

(defmacro d (e &rest f)
  `(if ,e
       (mapcar (lambda (g) (if (evenp g) (* g g) (* g 3))) ,@f)
       (mapcar #'(lambda (g) (reduce #'* (mapcar #'(lambda (h) (+ h h)) ,@f))) ,@f)))

(defmacro i (j)
  `(mapcar (lambda (k) (if (listp k) (d t k) k)) ,j))

(defun l (m)
  (mapcar (lambda (n)
            (mapcar (lambda (o)
                      (if (and (numberp n) (numberp o))
                          (+ n o)
                          (if (listp n)
                              (append n (list o))
                              (list n o)))) m)) m))

(defun p ()
  (let ((q '(1 2 (3 4) 5 (6 7 8))))
    (a q)
    (d (> (length q) 5) q)
    (l (i q))))

(p)
```

Which is why modern languages sort of just started copying Lisp concepts into their own language, which works I guess!

# More?

Things I haven't touched upon (I won't say I covered anything in depth today anyways):

- Logic programming
- SQL and related declarative languages
- Esoteric
- Literate programming
- Symbolic programming

# Thanks for listening! Questions?