Achievements

Levels of achievements in this course:

➤ Lowest: “I learned a few more programming languages.”

Principles of Programming Languages

➤ Medium: “I learned some topics in programming languages.”

Principles of Programming Languages

➤ Highest: “I began to see through the features in programming languages.”

Deconstruction and/or reductionism of Programming Languages?
Why This Course Exists

“You can pick up another language easily. It’s just another syntax.”

Was true 50 years ago because most languages had been just superficial enhancements to the von Neumann model:

\[
\text{memory} \xleftarrow{\text{1 word}} \text{CPU}
\]

i.e., still limited by \( s := s + a[i] \).

This is why they failed at abstraction, modularity, reusable components. (Even many OO languages today.)

(From John Backus’s *Can programming be liberated from the von Neumann style?* (1977).)

So people (including Backus) made new languages to explore new models. Therefore unfortunately for you. . .
Why This Course Exists

“You can pick up another language easily. It’s just another syntax.”

Now obsolete and false, even for practical programming languages.

Example: Java has both class-based OOP and automatic garbage collection. C has neither, but it has union types which Java doesn’t.

Switching from one to the other is non-trivial and requires re-thinking, dropping old habits, even overhaul. The difference is semantic.

And that’s pretty tame compared to this course.
Course Overview

Part I:

► Haskell (as implemented by GHC).
► I won’t teach its constructs systematically—you will read them up. But I will live-code to show how I think.
► Extra topics stemming from them.

Part II:

► Syntax: Moar context-free grammars.
► Some semantic topics, concretely through implementing interpreters for toy languages.
► If there is time, we examine object orientation and logic programming by implementing interpreters for them, too.
Example Topic: Evaluation Order

Define my own logical-and in Scheme (or Python, Java, C,...):

```
(define (my-and b c) (if b c #f))
```

`(my-and #f (> (/ 1 0) 1)) fails.

But in Haskell:

```
myAnd b c = if b then c else False
```

myAnd False (1 \div\ 0 > 1) succeeds.

Explanation (basic): In most languages, parameters are evaluated before passed to function bodies. In Haskell, parameters are passed as-is (and then the function body may evaluate some and short-circuit others).

Consequence: In Haskell many short-circuiting operators and control constructs are user-definable; in other languages you’re stuck with what’s hardwired.
Aside: Scheme Macros

To be fair, Scheme offers a macro system for user-defined constructs:

```scheme
(define-syntax-rule (my-and b c) (if b c #f))
```

(my-and #f (> (/ 1 0) 1)) succeeds this time.

Explanation: It’s macro expansion, parameters are copy-pasted.

Downside: To illustrate, define in Scheme

```scheme
(define-syntax-rule (double x) (+ x x))
```

(double (* 3 4)) spawns two copies of (* 3 4) and performs redundant work. (Haskell’s function application doesn’t.)

Upside: Scheme’s macro system offers other flexibilities.
Example Topic: {Dynamic, Static} Typing

In Python:
0 if False else 0+"hello" fails.
0 if True else 0+"hello" succeeds.

Explanation: Types are checked *dynamically*—when running the program, and only on the code path actually taken.

In Haskell:
if True then 0 else 0+"hello" fails.

Explanation: Types are checked *statically*—without running, and over all of the code. (If compiler, at compile time. If interpreter, at load time. Etc.)

Food for thought: Where does Java fit in this picture?
Example Topic: Parametric Polymorphism

In Haskell define: \( \text{trio} \ x = [x, x, x] \)
The [inferred] type is: \( a \rightarrow [a] \)
Analogous to Java’s \(<T> \text{LinkedList}<T> \ \text{trio}(<T> \ x) \)
\( \text{trio} \ 0 \) and \( \text{trio} \ "hello" \) are both legal.

“Parametric” means: Suppose you have defined \( d \) of type \( a \rightarrow [a] \)
Then I just need one test to know what it does.
Say I test \( d \ \text{True} \) and the answer has length 2.
Then I know \( d \ x \) gives \( [x, x] \), for all \( x \).

Explanation (basic): \( d \) cannot ask “what’s the type of the parameter?”, it cannot vary behaviour by types.

Teaser Preview: Haskell allows type-determined behaviour, but the function type will look like \( \text{Foo} \ a \Rightarrow a \rightarrow [a] \)
“Macro systems, dynamic typing, . . . are powerful.”
They mean flexibility when you write new code.

“Static typing, parametric polymorphism, . . . are powerful.”
They mean predictability when you use or understand existing code.

These two powers oppose each other.

Programming is a dialectic class struggle between the user and the implementer. Or between the maintainer and the original author.

My freedom is your slavery.

Your ignorance is my strength.
Practicality

My presentation of languages will tend to be academic.

This is not because they are impractical. It is only because I am teaching, not training, and I am teaching ideas.

Example: I use singly-linked lists all the time, but random-access arrays and efficient dictionaries are available in the standard libraries.

But we will have a guest talk by some functional programming practitioners (they’re using Clojure).
Appendix: Backus’s Proposal

(In case I have time left.)

Backus’s proposal:

- Higher order functions that work on aggregates (a whole list or array or dictionary or…)
- Combining forms e.g., function composition \((g \circ f)\).
- Reasoning by algebra.
- If you need state, have coarse-grained state transitions rather than changing only one word at a time.

These became the tenets of functional programming.
Higher-order Functions on Aggregates

(Notation: To apply a function to several parameters:
Haskell: \( f \ x \ y \ z \)  Scheme: \((f \ x \ y \ z)\)  

\( \text{map } f \ [x0, \ x1, \ldots] \) computes \([f \ x0, \ f \ x1, \ldots]\)
\( \text{map abs [3, -1, 4]} \) computes \([3, \ 1, \ 4]\)
Scheme: \((\text{map abs } '(3 \ -1 \ 4))\)

\( \text{foldr (+) 0 [3, 1, 4]} \) computes \(3+(1+(4+0))\)
Scheme (using Racket): \((\text{foldr } + \ 0 \ '(3 \ 1 \ 4))\)

“On aggregates”: Work on a whole list at once (or array, or some “container”…)

“Higher-order function”: Some parameters are functions—customizable.
Combining Forms

Obvious example: Function composition \((g \circ f)\).

Haskell: \(g \cdot f\)  
Racket: `(compose g f)

\(\text{foldr} (+) \ 0 \ . \ \text{fmap} \ \text{abs}\) computes the 1-norm of your vector.

(Unfortunately the equivalent Scheme code is not as nice.)

There are other combining forms. Here is an example in Haskell:

\(f \ &&& \ g\) satisfies: \((f \ &&& \ g) \ x = (f \ x, \ g \ x)\)

The point:

- You’ve got functions that perform basic tasks on aggregates. Now hook them up to perform compound tasks.

This is not about shorter code (although it has that side effect). This is about working with building blocks.