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

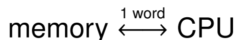
- Highest: “I began to see through the features in programming languages.”

  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:



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. (Not comprehensive—I do the hard parts and some live-coding; you learn the easy parts.)
- Some topics stemming from it.

Part II:

- Syntax: Moar context-free grammars; simple parsers.
- Semantics: By implementing toy languages in Haskell.
  Why in Haskell: As close to executable *mathematical* models as I can get in an undergrad course.

(Yes, IMO ideally this should be a *math* course!)

Next few slides elaborate a bit...

Example Topic: Evaluation Order

Define $f(x) = 4$. Now $f(1/0) =?$

Call by value (most languages): Evaluate $1/0$ first. Error.

Lazy evaluation (e.g., Haskell): Don't evaluate $1/0$ yet, just plug in as-is. Oh x is unused, $f(1/0) = 4$.

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 is call by value, but provides a macro system for user-definable control constructs and other constructs.

Example Topic: {Dynamic, Static} Typing

`0 if True else "hello"` — OK in Python.
⇐ Dynamic (run-time, and only on executed code paths) type checking.

`if True then 0 else "hello"` — static (compile-time or load-time) error in Haskell.
⇐ Static (compile-time, and all code considered) type checking.

Food for thought: What about `1 ? 0 : "hello"` in C?
Could you demo static type checking in C?

Food for thought: Where does Java stand?

Example Topic: Parametric Polymorphism

In Haskell define: `trio x = [x, x, x]`
[Inferred] Type: `t -> [t]`
Like Java's `<t> LinkedList<t> trio(<t> x)`

`trio 0` and `trio "hello"` are both legal.

"Parametric" = User chooses what type to use for the type variable `t`, *and* implementation not told what it is.

Consequence: Behaviour cannot vary by types. Corollary: Inflexible, but easy to test—test on one type and conclude for all types.

Teaser preview: Haskell allows type-determined behaviour, but the function type will look like `Foo a => a -> [a]`

Some Other Example Topics

(Mathematical model of) Mutable variables.

If there is time: (mathematical model of) objects.

If there is time: (mathematical model of) logic programming.

What Is “Powerful”?—The Tradeoff

“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.

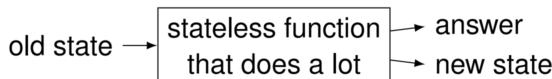
But we will have a guest talk by some functional programming practitioners (they're using Clojure).

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)`   )

`map f [x0, x1, ...]` computes `[f x0, f x1, ...]`
`map abs [3, -1, 4]` computes `[3, 1, 4]`
Scheme: `(map abs '(3 -1 4))`

`foldr (+) 0 [3, 1, 4]` computes `3+(1+(4+0))`
Scheme (using Racket): `(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 . f`   Racket: `(compose g f)`

`foldr (+) 0 . fmap 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.