Levels of Achievements

Levels of achievements in this course:

- Lowest: “I learned some programming languages.”

  Principles of **Programming Languages**

Levels of Achievements

Levels of achievements in this course:

- Lowest: “I learned some programming languages.”

  Principles of **Programming Languages**

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

  Principles of Programming Languages

  I hope most of you will achieve this.

Levels of Achievements

Levels of achievements in this course:

- Lowest: “I learned some programming languages.”

  Principles of **Programming Languages**

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

  Principles of Programming Languages

  I hope most of you will achieve this.

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

  Deconstruction/Reductionism of Programming Languages?

  This one is very hard. I’m not sure I can teach it either.

Course Overview

Part I:

- Haskell. (Not comprehensive—I show the hard parts, you pick up the easy parts, and we focus on the parts we need.)
- Basic topics.

Course Overview

Part I:

- Haskell. (Not comprehensive—I show the hard parts, you pick up the easy parts, and we focus on the parts we need.)
- Basic topics.

Part II:

- Syntax: Moar context-free grammars; simple parsers.
- Semantics: By toy language models in Haskell.
  Why Haskell: Almost like math definition, and executable.
  (In a grad course I would use actual pure math.)
- Advanced topics.

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, $f(1/0) = 4$.

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.

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: 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.

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.

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

Consequence: Uniform behaviour. Can't vary by types:

```
trio 0 = [0, 0, 0]
trio "hello" = []
```

Less flexible, but easier to test—test on one type and conclude for all types.

If we have time, I'll show you how to *prove* that.

Some Other Example Topics

Type inference.

Model of local variables and local functions.

If there is time: Model of mutable variables.

If there is time: Continuations.

Practicality

My presentation of languages will tend to be academic.

This is not because they are impractical. It is only because I am teaching selected topics.

Example: I use naïve singly-linked lists all the time, but data structures for grown-ups such as random-access arrays and efficient dictionaries are available.