Functions

Prof. Clarkson

Fall 2017

Today's music: Function by E-40 (Clean remix)

A0: Warmup

- Worth only 1% of final grade; other assignments will be 5%
  - much easier coding problems
  - intended to give you low-stakes experience with 3110 workflow

- Please review the **late policy** in the course syllabus
  - sliding scale of penalty based on days late
  - *deadline* is the time by which you must successfully upload your solution files to CMS and confirm that CMS has recorded the correct versions of those files

- Please review the **academic integrity** policy in the course syllabus
  - we use MOSS to detect copying of code; *it works*
  - cite your sources (people, URLs)
  - don't claim other people's ideas/code as your own – that is a violation of AI and will lead to prosecution

- Please don't try to submit by email, regardless of reason

Recitation swap

- Opens today at about noon
- Closes Saturday at about 7 am
- https://goo.gl/forms/Njb5mk0AIUxcDnJf2
- PINs issued (hopefully) Monday
- If you already contacted me in any way about this, you still need to fill out form
- If you filled it out last weekend, your info is still there: please update as needed

Review

Previously in 3110:

- What is a functional language?
- Why learn to program in a functional language?

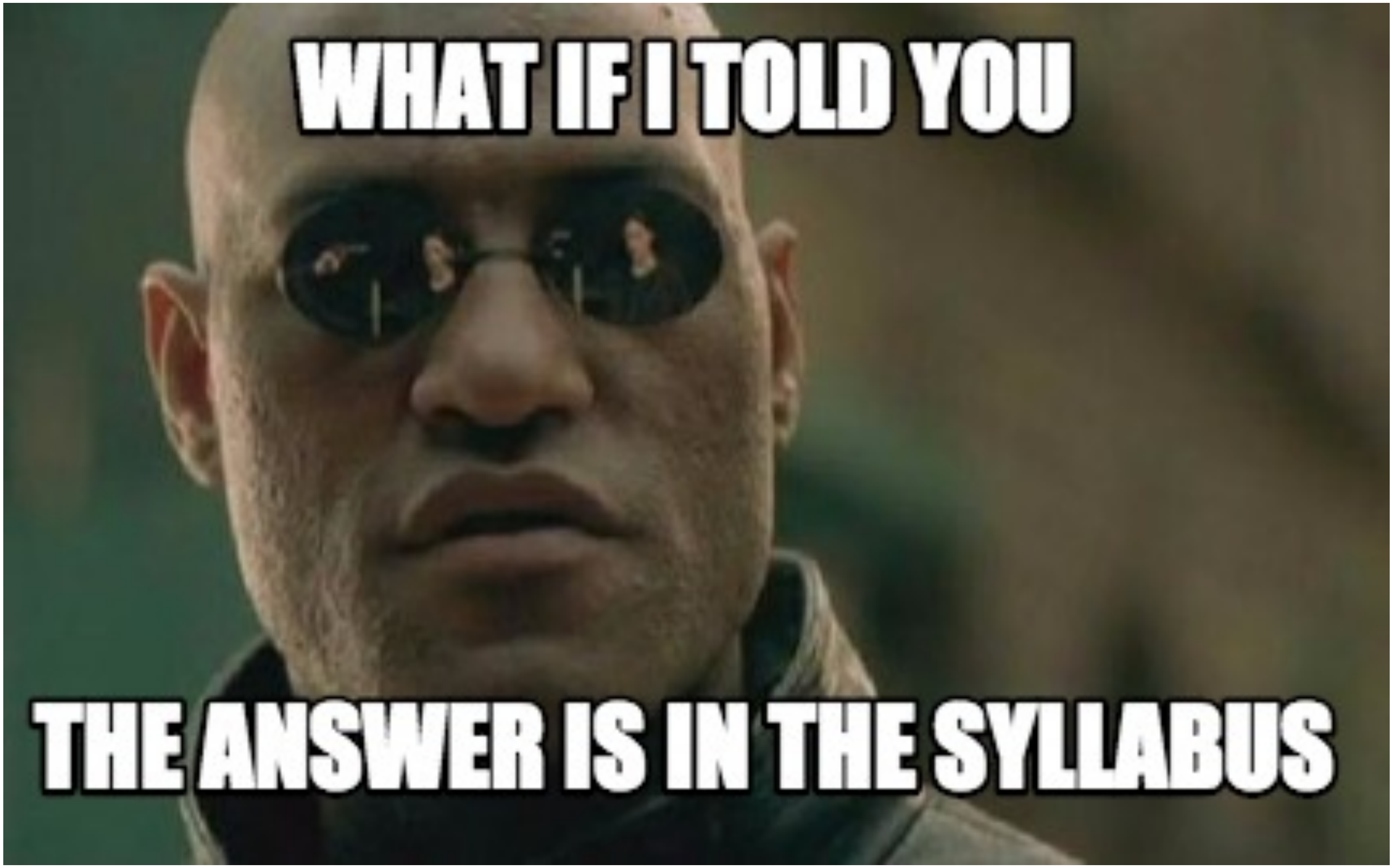
Today:

- **Functions:** the most important part of functional programming!

Question

Did you read the syllabus?

A. Yes
B. No
C. I plead the 5th


WHAT IF I TOLD YOU
THE ANSWER IS IN THE SYLLABUS

Five aspects of learning a PL

1. Syntax: How do you write language constructs?
2. Semantics: What do programs mean? (Type checking, evaluation rules)
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a third-party project) provide as "standard"? (E.g., file access, data structures)
5. Tools: What do language implementations provide to make your job easier? (E.g., top-level, debugger, GUI editor, …)

- All are essential for good programmers to understand
- Breaking a new PL down into these pieces makes it easier to learn

Our Focus

We focus on semantics and idioms for OCaml

- Semantics is like a meta-tool: it will help you learn languages
- Idioms will make you a better programmer in those languages

Libraries and tools are a secondary focus: throughout your career you'll learn new ones on the job every year

Syntax is almost always boring

- A fact to learn, like "Cornell was founded in 1865"
- People obsess over subjective preferences {yawn}
- Class rule: We don't complain about syntax

Expressions

Expressions (aka *terms*):

- primary building block of OCaml programs
- akin to *statements* or *commands* in imperative languages
- can get arbitrarily large since any expression can contain subexpressions, etc.

Every kind of expression has:

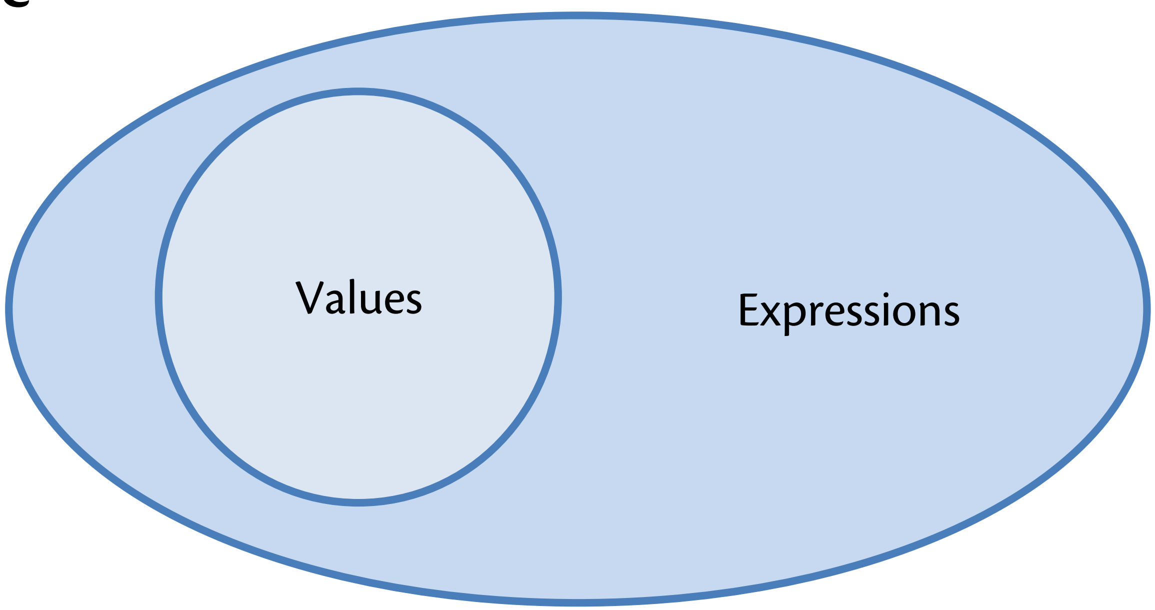
- **Syntax**
- **Semantics:**
  - **Type-checking rules (*static semantics*)**: produce a type or fail with an error message
  - **Evaluation rules (*dynamic semantics*)**: produce a *value*
    - (or exception or infinite loop)
    - Used only on expressions that type-check

Values

A **value** is an expression that does not need any further evaluation

– **`34`** is a value of type **`int`**

– **`34+17`** is an expression of type **`int`** but is not a value



IF EXPRESSIONS

`if` expressions

Syntax:

```
if e1 then e2 else e3
```

Evaluation:

- if `e1` evaluates to `true`, and if `e2` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`
- if `e1` evaluates to `false`, and if `e3` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`

Type checking:

if `e1` has type `bool` and `e2` has type `t` and `e3` has type `t` then `if e1 then e2 else e3` has type `t`

Types

Write colon to indicate type of expression

As does the top-level:

```
# let x = 22;;
val x : int = 22
```

Pronounce colon as "has type"

`if` expressions

Syntax:

`if e1 then e2 else e3`

Evaluation:

- if **`e1`** evaluates to **`true`**, and if **`e2`** evaluates to **`v`**, then **`if e1 then e2 else e3`** evaluates to **`v`**
- if **`e1`** evaluates to **`false`**, and if **`e3`** evaluates to **`v`**, then **`if e1 then e2 else e3`** evaluates to **`v`**

Type checking:

if **`e1: bool`** and **`e2:t`** and **`e3:t`**
then **`if e1 then e2 else e3 : t`**

`if` expressions

Syntax:

`if e1 then e2 else e3`

Evaluation:

- if `e1` evaluates to `true`, and if `e2` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`
- if `e1` evaluates to `false`, and if `e3` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`

Type checking:

if `e1: bool` and `e2:t` and `e3:t`
then `(if e1 then e2 else e3) : t`

Question

To what value does this expression evaluate?

```
if 22=0 then 1 else 2
```

A. 0
B. 1
C. 2
D. none of the above
E. I don't know

Question

To what value does this expression evaluate?

```
if 22=0 then 1 else 2
```

A. 0
B. 1
C. 2
D. none of the above
E. I don't know

Question

To what value does this expression evaluate?

```
if 22=0 then "bear" else 2
```

A. 0
B. 1
C. 2
D. none of the above
E. I don't know

Question

To what value does this expression evaluate?

```
if 22=0 then "bear" else 2
```

A. 0
B. 1
C. 2
D. **none of the above:** doesn't type check so never gets a chance to be evaluated; note how this is (overly) conservative
E. I don't know

FUNCTIONS

Function definition

Functions:

- Like Java methods, have arguments and result
- Unlike Java, no classes, **`this`**, **`return`**

Example *function definition*:

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow x y =
  if y=0 then 1
  else x * pow x (y-1)
```

Note: **`rec`** is required because the body includes a recursive function call

Note: no types written down! compiler does *type inference*

Writing argument types

Though types can be inferred, you can write them too.

Parens are then mandatory.

```
let rec pow (x : int) (y : int) : int =
  if y=0 then 1
  else x * pow x (y-1)

let rec pow x y =
  if y=0 then 1
  else x * pow x (y-1)

let cube x = pow x 3
let cube (x : int) : int = pow x 3
```

Function definition

Syntax:

```
let rec f x1 x2 ... xn = e
```

note: **`rec`** can be omitted if function is not recursive

Evaluation:

Not an expression! Just defining the function; will be evaluated later, when applied

Function types

Type `t -> u` is the type of a function that takes input of type **`t`** and returns output of type **`u`**

Type `t1 -> t2 -> u` is the type of a function that takes input of type **`t1`** and another input of type **`t2`** and returns output of type **`u`**

etc.

Function definition

Syntax:

```
let rec f x1 x2 ... xn = e
```

Type-checking:

Conclude that `f : t1 -> ... -> tn -> u` if `e:u` under these assumptions:

- `x1:t1, ..., xn:tn` (arguments with their types)
- `f: t1 -> ... -> tn -> u` (for recursion)

Function application v1

Syntax: **`f e1 ... en`**

- Parentheses not required around argument(s)
- Possible for syntax to look like C function call:
  - **`f(e1)`**
  - if there is exactly one argument
  - and if you do use parentheses
  - and if you leave out the white space

Function application v1

Type-checking

if `f : t1 -> ... -> tn -> u`

and `e1 : t1, ..., en : tn`

then `f e1 ... en : u`

e.g.

`pow 2 3 : int`

because `pow : int -> int -> int`

and `2:int` and `3:int`

Function application v1

Evaluation of `f e1 ... en`:

1. Evaluate arguments `e1...en` to values `v1...vn`
2. Find the definition of `f`
   `let f x1 ... xn = e`
3. Substitute `vi` for `xi` in `e` yielding new expression `e'`
4. Evaluate `e'` to a value `v`, which is result

Example

```
let area_rect w h = w *. h
let foo = area_rect (1.0 *. 2.0) 11.0
```

To evaluate function application:

1. Evaluate arguments `(1.0 *. 2.0)` and `11.0` to values `2.0` and `11.0`
2. Find the definition of `area_rect`
   `let area_rect w h = w *. h`
3. Substitute in `w *. h` yielding new expression `2.0 *. 11.0`
4. Evaluate `2.0 *. 11.0` to a value `22.0`, which is result

Anonymous functions

Something that is *anonymous* has no name

- **`42`** is an anonymous **`int`**
- and we can bind it to a name:
  `let x = 42`

- **`fun x -> x+1`** is an anonymous function
- and we can bind it to a name:
  `let inc = fun x -> x+1`

note: dual purpose for **`->`** syntax: function types, function values
note: **`fun`** is a keyword :)

Anonymous functions

Syntax: `fun x1 ... xn -> e`

Evaluation:

- Is an expression, so can be evaluated
- A function *is* a value: no further computation to do
- In particular, body `e` is not evaluated until function is applied

Type checking:

`(fun x1 ... xn -> e) : t1->...->tn->t`

if `e:t` under assumptions `x1:t1, ..., xn:tn`

Anonymous functions

These definitions are **syntactically different** but semantically equivalent:

```
let inc = fun x -> x+1
let inc x = x+1
```

For now, regard this as two ways of saying the same thing

Later, we'll see great uses for anonymous functions!

Lambda

- Anonymous functions a.k.a. *lambda expressions*
- Math notation: `λx . e`
- The lambda means "what follows is an anonymous function"
  - x is its argument
  - e is its body
  - Just like `fun x -> e`, but different "syntax"

- You'll see "lambda" show up in many places in PL, e.g.:
  - PHP: http://www.php.net/manual/en/function.create-function.php
  - Python: https://docs.python.org/3.5/tutorial/controlflow.html#lambda-expressions
  - Java 8: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
  - A popular PL blog: http://lambda-the-ultimate.org/
  - Lambda style: https://www.youtube.com/watch?v=Ci48kqp11F8

Function application operator

- Infix operator for reverse function application
- Instead of `f e` can write `e |> f`
- Run a value through several functions
  `5 |> inc |> square  (* 36 *)`
- "pipeline" operator

Functions are values

- Can use them **anywhere** we use values
- Functions can **take** functions as arguments
- Functions can **return** functions as results

  ...so functions are *higher-order*

- This is not a new language feature; just a consequence of "a functions *is* a value"
- But it is a feature with massive consequences!

Upcoming events

- [Mon] attendance starts for real; you must attend your registered registration section to get credit

This is **`fun`***!*

THIS IS 3110

FOR RECITATION

Function application v2

Syntax: `e0 e1 ... en`

- Function to be applied can be an expression
  - Maybe just a defined function's name
  - Or maybe an anonymous function
  - Or maybe something even more complicated
- Example:
  - `(fun x -> x + 1) 2`

Function application v2

Type-checking (not much of a change)

if `e0 : t1 -> ... -> tn -> u`

and `e1 : t1, ..., en : tn`

then `e0 e1 ... en : u`

Function application v2

Evaluation of `e0 e1 ... en`:

1. Evaluate arguments `e1...en` to values `v1...vn`
   Also evaluate `e0` to a function
   `fun x1 ... xn -> e`
2. Substitute `vi` for `xi` in `e` yielding new expression `e'`
3. Evaluate `e'` to a value `v`, which is result

Function application v2

Evaluation of **`e0 e1 ... en`**:

Evaluate **`e0`** to a function

```
fun x1 ... xn -> e
```

Examples:

- **`e0`** could be an anonymous function expression
  `fun x -> x+1`
  in which case evaluation is immediately done
- **`e0`** could be the name of a defined function
  `inc`
  in which case look up the definition
  `let inc x = x + 1`
  and we now know that's equivalent to
  `let inc = fun x -> x+1`
  so evaluates to
  `fun x -> x+1`