# CSCI 334:
# Principles of Programming Languages

## Lecture 2: More F#

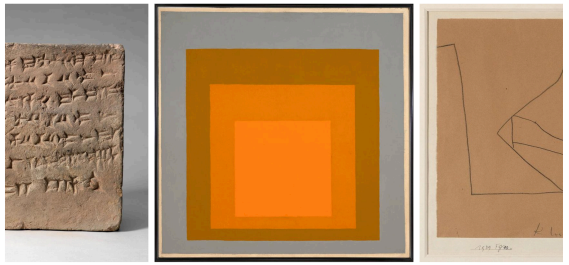
Instructor: Dan Barowy

Williams

# Announcements

Computer Science 334: Principles of Programming Languages
Professor Dan Barowy

A programming language is a distinct vocabulary and set of rules used to communicate with a computer and to achieve a desired result. Students learn to use precise, nuanced words to describe something and how to combine those words effectively. They practice this careful description by writing about compositional elements they observe here such as shape, color, size, and texture. For the final project, each student designs a programming language capable of generating artwork in the style of an artist selected from *Object Lab*.



Homage to the Square: Warming, 1959. Oil on Masonite. Josef Albers, b. 1888, Bottrop, Deutsches Reich (present-day Germany), d. 1976, New Haven, Connecticut. Gift of Jillion and Michael D. Weisberg, Class of 1993. M.2007.13

Field trip: meet at WCMA on Thursday

# Announcements


**Computer Science Movie Night: The Matrix**

**Thursday, Feb 13 at 6:30pm Wege Auditorium**

**(finishes before fireworks @ Poker Flats)**

**Benefits:**

- Fun!
- Snacks!
- You will finally be able to understand your professor's jokes!
- You will be able to converse fluently with other insufferable nerds!
- You might learn a little computer science!
- Did I mention snacks?!!

# Topics

Function definitions

`unit`

Lists

Recursion

Pattern matching

## Your to-dos

1. Read *What is a PL?, Intro to F#, More F#*.
2. Lab 1, **due Wed 2/12 by midnight** (solo lab).
3. Practice quiz on Thursday.

## Nobody ever makes the first jump

## Freeing your mind is difficult

## Give yourself time to try and fail

# Functions

ML-family languages avoid unnecessary syntax

```
> let foo a b c = a;;
val foo: a: 'a -> b: 'b -> c: 'c -> 'a
> foo 1 2 3;;
val it: int = 1
```

# Functions

But you can use parens if it makes you comfortable

```
> let foo(a: 'a)(b: 'b)(c: 'c) = a;;
val foo: a: 'a -> b: 'b -> c: 'c -> 'a
> foo(1)(2)(3);;
val it: int = 1
```

# Functions

Sidebar

```
> let foo(a, b, c) = a;;
val foo:'a * 'b * 'c -> 'a
> foo(1, 2, 3);;
val it: int = 1
```

# Tuples

```
> ("a", 1, 4.4);;
val it: string * int * float = ("a", 1, 4.4)
> let baz (a: string, b: int, c: float) = (a,b,c);;
val baz: a: string * b: int * c: float -> string * int * float
```

# Functions

Multi-line

```
let foo (x: int)(y: int) : int =
  x +
  y
```

# `unit` datatype

```
public static void main(String[] args) { … }
```

```
let main args = …
```

# `unit` datatype

```
public static void main(String[] args) { … }
```

```
let main(args: string[]) = …
```

Remember: every expression must **return a value**.

A function **can't** return nothing.

# `unit` datatype

```
public static void main(String[] args) { … }
```

```
let main(args: string[]) : unit = …
```

Therefore, "nothing" is a thing… called `unit`.

## `unit` datatype

```
$ dotnet fsi

Microsoft (R) F# Interactive version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> unit;;

  unit;;
  ^^^^

stdin(1,1): error FS0039: The value or constructor 'unit' is
not defined.

> ();;
val it : unit = ()

>
```

How does one obtain a value of `unit`? `()`

## By the way…

```
let main(args: string[]) : unit = …
```

## By the way…

```
let main(args: string[]) : int = …
```

## Records

```
> type Point = { x: int; y: int; z: int };;
type Point =
  {
    x: int
    y: int
    z: int
  }
> let p = { x = 1; y = 2; z = 3 };;
val p: Point = { x = 1
                 y = 2
                 z = 3 }
> let up pt = { x = pt.x; y = pt.y + 1; z = pt.z };;
val up: pt: Point -> Point
> up p;;
val it: Point = { x = 1
                  y = 3
                  z = 3 }
```

# Lists

## Linked List

A **linked list** is a recursive data structure.

A list is either:

- the **empty list**, or
- a **node**, containing an element and a reference to a list.

## Linked List

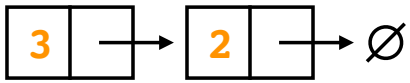
∅

The empty list is defined as **nil** (or `[]`)

## Linked List

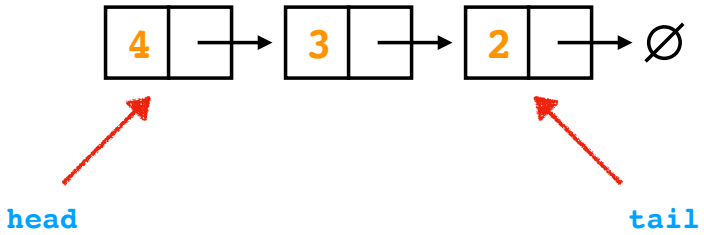
Every other list has at least one list node.

# Linked List




The last node in the list always points to **nil**.

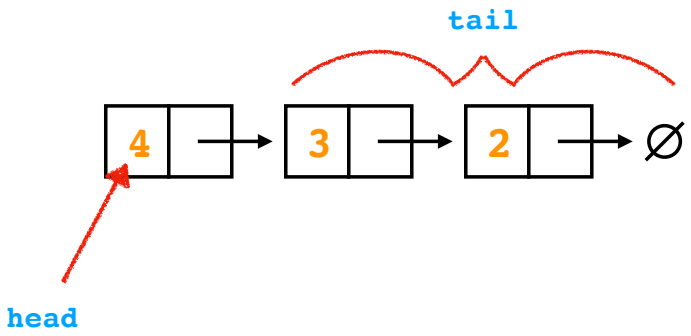
# Linked List



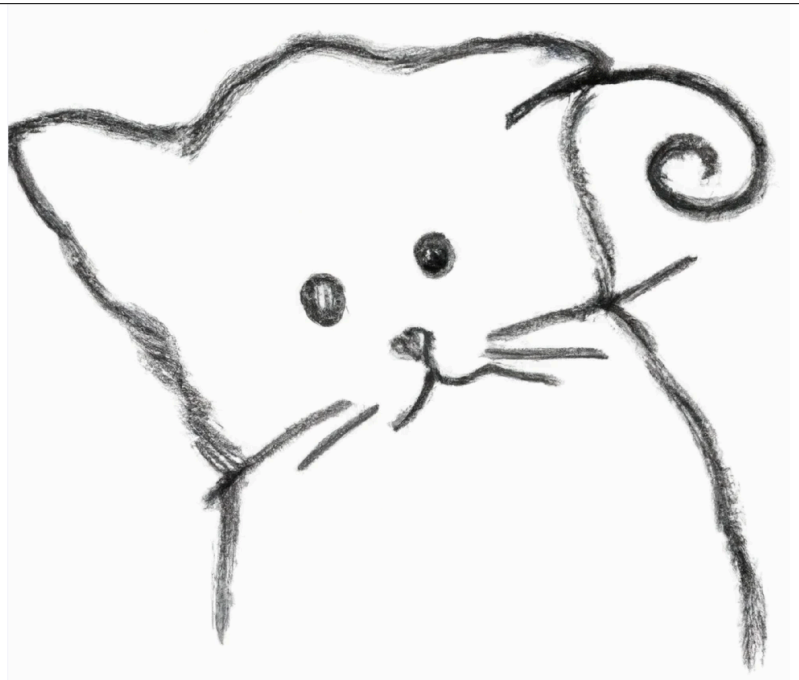

A list has parts.*

*we do not use this definition in 334

# Linked List




A list has parts.



# Linked List

A **linked list** is a recursive data structure.

A list is either:

- the **empty list**, or
- a **node**, containing a head and a tail.

# Lists

- Examples
  - `[]` is the empty list
  - `[1; 2; 3; 4]`, `["wombat"; "dingbat"]`
  - all elements of list **must be same type**
- Operations
  - length — `List.length [1;2;3]` ⇒ `3`
  - cons — `1::[2;3]` ⇒ `[1; 2; 3]`
  - head — `List.head [1;2;3]` ⇒ `1`
  - tail — `List.tail [1;2;3]` ⇒ `[2;3]`
  - append — `[1;2]@[3;4]` ⇒ `[1; 2; 3; 4]`
  - map — `List.map succ [1;2;3]` ⇒ `[2;3;4]`

# List types

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What **type** of list is `[]`?

```
- [];
val it : 'a list
```

- Polymorphic type
  - `'a` is a type variable that represents **any type**
  - `1::[] : int list`
  - `"a"::[] : string list`

# Recursive functions

- Note that **recursive** functions must use **`rec`** keyword.
- Not valid:

```
let fact n =
  if n <= 0 then
    1
  else
    n * fact (n - 1)
```

- Instead:

```
let rec fact n =
  if n <= 0 then
    1
  else
    n * fact (n - 1)
```

## Recursive functions

Let's write a list `length` function

```
let rec length (xs: 'a list) : int =
    if xs = [] then
        0
    else
        1 + length (List.tail xs)
```

Let's use the following functions:

- **`List.head`**
- **`List.tail`**

**`length`** should be polymorphic.

## Recursive functions

Let's write a list `length` function

```
let rec length (xs: 'a list) : int =
    if xs = [] then
        0
    else
        1 + length (List.tail xs)
```

Let's write a list `sum` function (not polymorphic)

## Recursive functions

Let's write a list `length` function

```
let rec length (xs: 'a list) : int =
    if xs = [] then
        0
    else
        1 + length (List.tail xs)
```

Let's write a list `sum` function (not polymorphic)

```
let rec sum (xs: int list) : int =
    if xs = [] then
        0
    else
        (List.head xs) + sum (List.tail xs)
```

## Recap & Next Class

Today:

Pattern matching

Function definitions

Lists

Recursion

unit

Next class:

WCMA