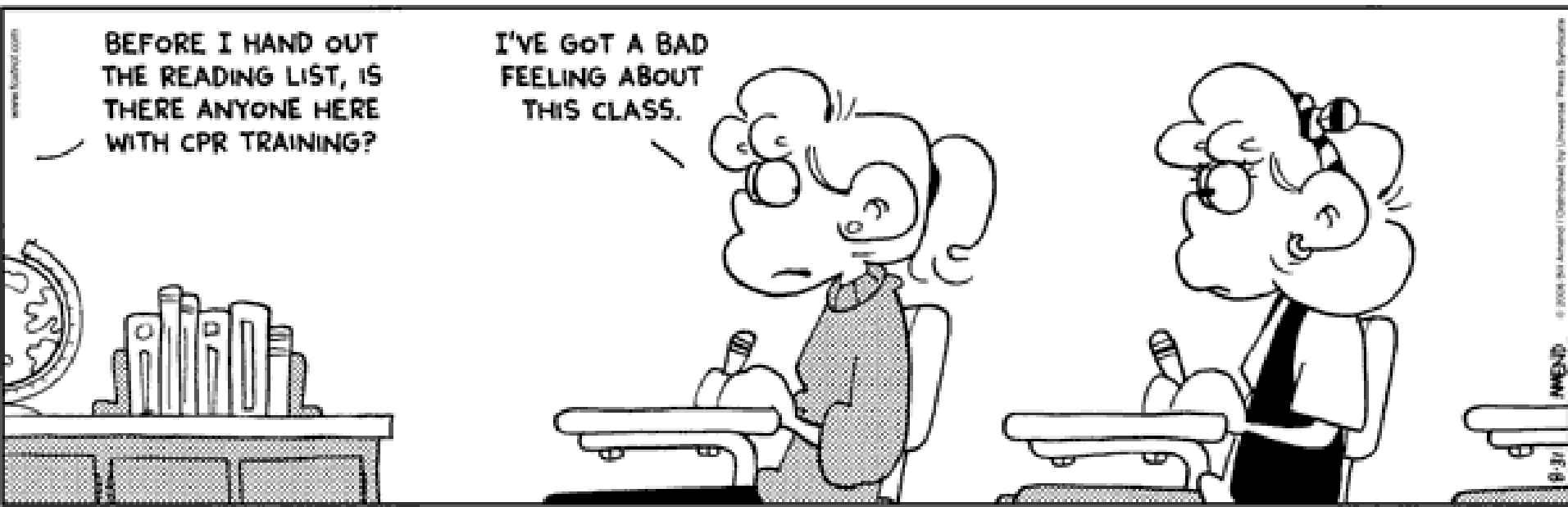


# History of Programming Languages

## Functional Programming



# Cunning Plan

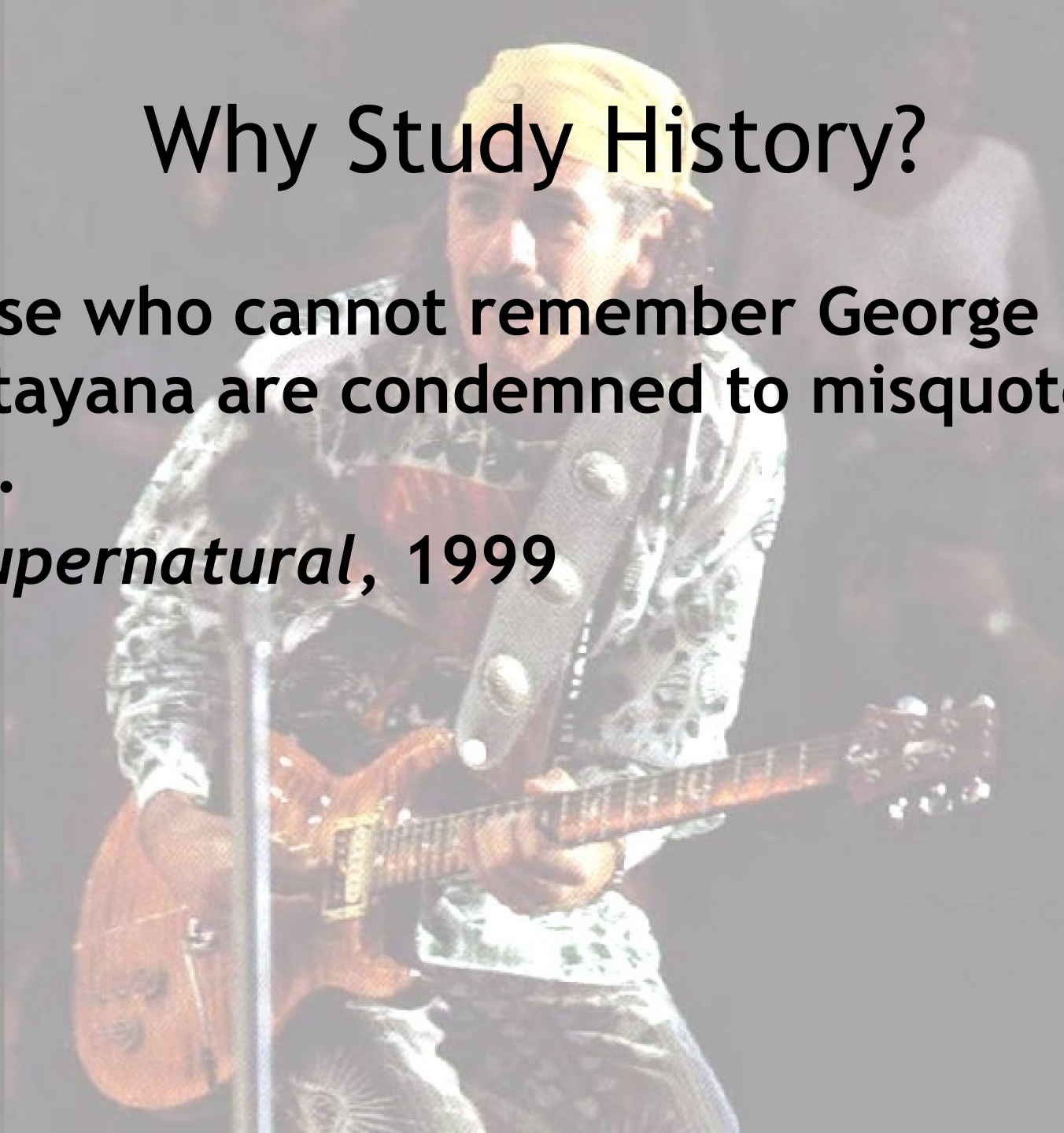
- History Lesson
- Functional Programming
  - OCaml
  - Types
  - Pattern Matching
  - Higher-Order Functions
- Basic Syntax
- Data Structures
- Higher-Order Functions
  - Fold

# Gone In Sixty Seconds

- **Imperative:** change state, assignments
- **Structured:** if/block/routine control flow
- **Object-Oriented:** message passing (= dynamic dispatch), inheritance
- **Functional:** functions are first-class citizens that can be passed around or called recursively. We can avoid changing state by passing copies.

# Why Study History?

- Those who cannot remember George Santayana are condemned to misquote him.
  - *Supernatural*, 1999



# Why Study History?

- Progress, far from consisting in change, depends on retentiveness. **Those who cannot remember the past are condemned to repeat it.**
  - George Santayana, *Life of Reason: Vol. I, Reason and Common Sense*, 1905-1906.
- Through meticulous analysis of history I will find a way to make the people worship me. By studying the conquerors of days gone by, **I'll discover the mistakes that made them go awry.**
  - The Brain, *A Meticulous Analysis of History*

## Politics this week

Jan 21st 2012 | from the print edition



In an effort to press **Iran** harder to make it halt its nuclear programme, the European Union agreed in principle to block imports of Iranian oil. Iran urged Arab countries not to increase their production in response to the sanctions. America is also seeking the support for the measure from other big importers of Iranian oil, notably China. [See article](#)



Strikes brought **Nigeria** to a halt after the government removed the country's fuel-subsidy regime, causing petrol prices to soar. Under pressure from trade unions, President Goodluck Jonathan agreed to lower prices again but said the subsidies would eventually be phased out entirely. [See article](#)

At least 51 people, mostly women and children, were killed in clashes in **South Sudan's** Jonglei state, which lies near the border with the north. The violence is part of a deadly cycle of cattle raids and revenge attacks between some of the country's ethnic groups.

---

### In this section

» Politics this week

Business this week

KAL's cartoon

Reprints

---

### Related topics

2012 Election



# Surprise Liberal Arts Trivia

- The **Ulster Cycle** (or **Red Branch Cycle**) is one of the four great sagas of *this country's* mythology. It includes prominent figures such as Cú Chulainn and queen Méabh, as well as the tragic Deirdre (source of Yeats and Synge plays). The earliest of the stories available is dated to the 8<sup>th</sup> century and refers to events and characters of the 7<sup>th</sup>.

# So What's It About?

- The longest and most important story of the cycle is the Táin Bó Cúailnge or "Cattle Raid of Cooley", in which Medb raises an enormous army to invade the Cooley peninsula and steal the Ulaid's prize bull [...] Warfare mainly takes the form of cattle raids [...] Cú Chulainn [...] staves off Medb's army for months, slaying every champion the queen sends to meet him. [...] Medb, of course, is not finished with Cú Chulainn, and seeks her revenge on him through more trickery.
  - Wikipedia and others, emphasis mine
- In other words: “A deadly cycle of cattle raids and revenge attacks between some of the country's groups.”



# One Reason Why

- Reason is a biological product -- a tool whose power is inherently and substantially restricted. It has improved how we do things; it has not changed why we do things. Reason has generated knowledge enabling us to fly around the world in less than two days. Yet we still travel for the same purposes that drove our ancient ancestors -- commerce, conquest, religion, romance, curiosity, or escape from overcrowding, poverty, and persecution. To deny that reason has a role in setting our goals seems, at first, rather odd. A personal decision to go on a diet or take more exercise appears to be based upon reason. The same might be said for a government decision to raise taxes or sign a trade treaty. But reason is only contributing to the 'how' portion of these decisions; the more fundamental 'why' element, for all of these examples, is driven by instinctive self-preservation, emotional needs, and cultural attitudes. We are usually reluctant to admit the extent to which these forces govern our behavior, and accordingly we often recruit reason to explain and justify our actions.

- Donald B. Calne, *Within Reason: Rationality and Human Behavior*

# Modern Era

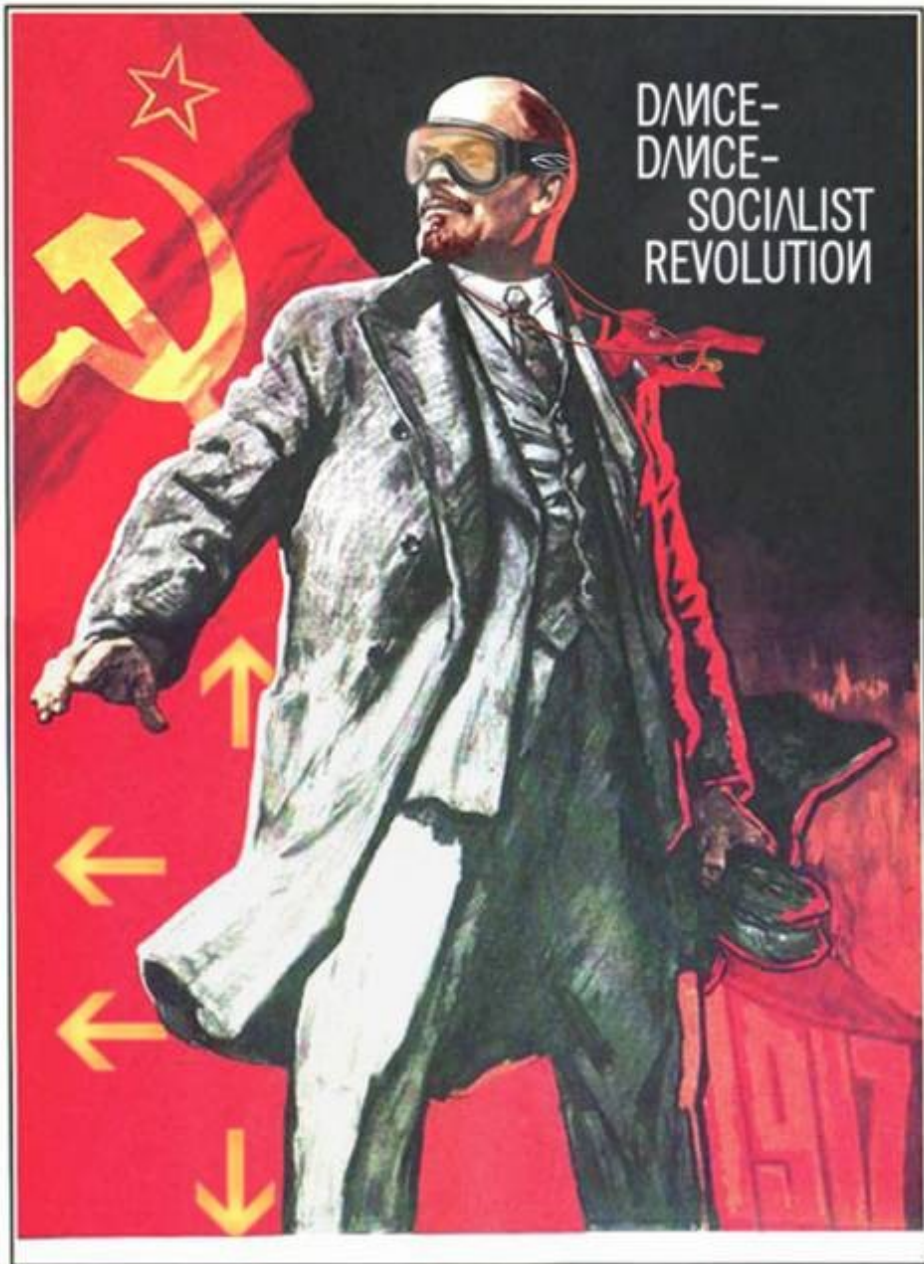
*I invented the term Object-Oriented, and I did not have C++ in mind.*

- Alan Kay

- 1972 - C Systems programming, ASM
- 1983 - Ada US DOD, static type safety
- 1983 - C++ classes, default args, STL
- 1987 - Perl dynamic scripting language
- 1990 - Python interp OO + readability
- 1991 - Java portable OO lang (for iTV)
- 1993 - Ruby Perl + Smalltalk
- 1996 - OCaml ML + C++
- 2000 - C# “simple” Java + delegates

# Time Travel

- Back to an earlier time when the US was worried about a Communist “perfect attack”
- In Soviet Russia, noun verbs you!  
(-1 Redundant)

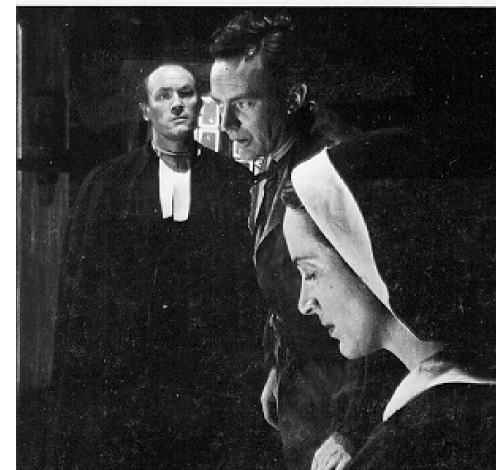


# The Land Before Time



- It was a time very different now ...
- **Senator Joseph McCarthy 1950**
  - "I have here in my hand a list of 205 — a list of names ..."
- **John McCarthy 1958**
  - **LISP** = List Processing Language
  - basic datatype is the List, programs themselves are lists, can self-modify, dynamic allocation, garbage collection (!), functional

ARTHUR  
**MILLER**  
THE CRUCIBLE

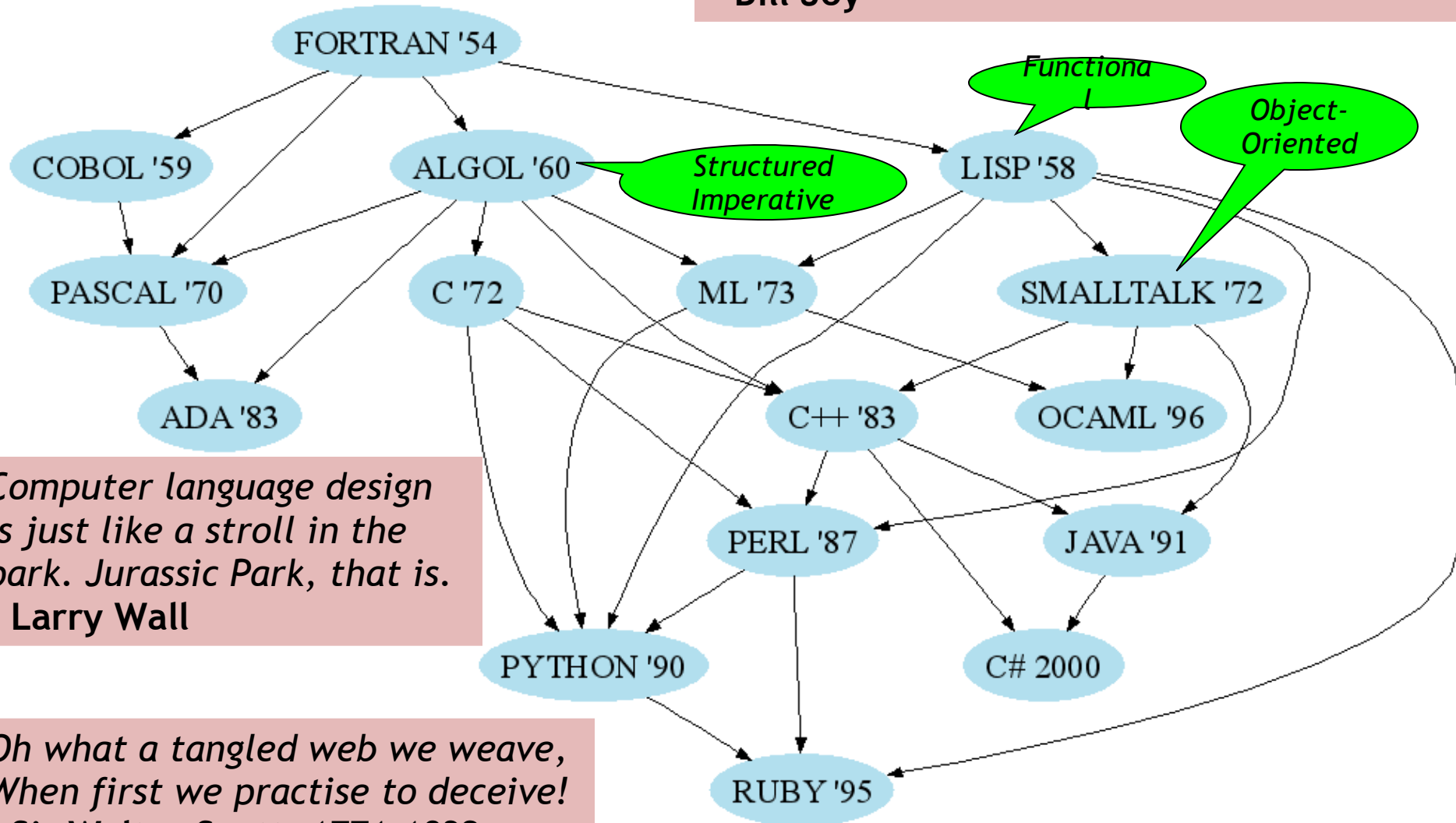


*There are only two kinds of programming languages: those people always [complain] about and those nobody uses.*

**- Bjarne Stroustrup**

*I fear the new OO systems may suffer the fate of LISP, in that they can do many things, but the complexity of the class hierarchies may cause them to collapse under their own weight.*

**- Bill Joy**



*Computer language design is just like a stroll in the park. Jurassic Park, that is.*

**- Larry Wall**

*Oh what a tangled web we weave,  
When first we practise to deceive!*

**- Sir Walter Scott, 1771-1832**

# Functional Programming

- You know OO and Structured Imperative
- Functional Programming
  - Computation = evaluating (math) functions
  - Avoid “global state” and “mutable data”
  - Get stuff done = apply (higher-order) functions
  - Avoid sequential commands
- Important Features
  - Higher-order, first-class functions
  - Closures and recursion
  - Lists and list processing

# State

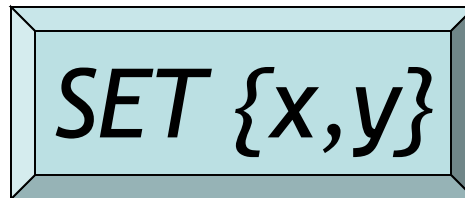
- The state of a program is all of the current variable and heap values
- Imperative programs destructively modify existing state



`add_elem(SET, y)`

# State

- The state of a program is all of the current variable and heap values
- Imperative programs destructively modify existing state

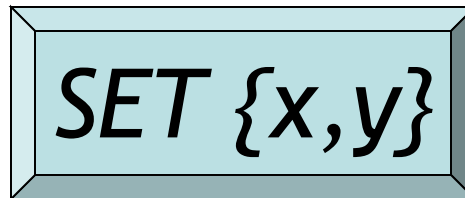


*SET {x,y}*

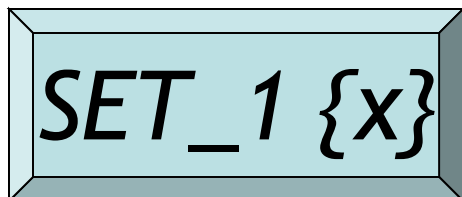


# State

- The state of a program is all of the current variable and heap values
- Imperative programs destructively modify existing state
- Functional programs yield new similar states over time

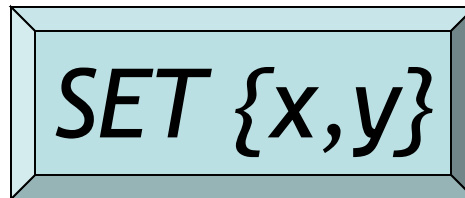


`SET_2 = add_elem(SET_1, y)`

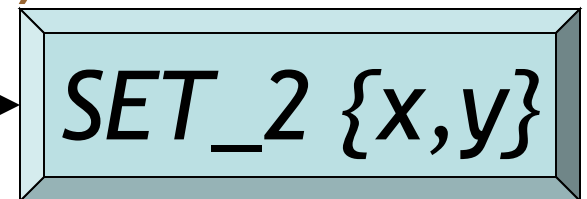
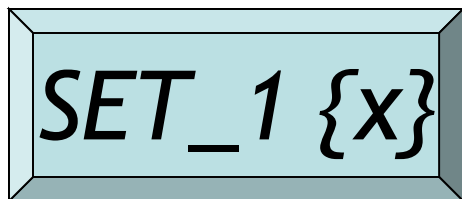


# State

- The state of a program is all of the current variable and heap values
- Imperative programs destructively modify existing state
- Functional programs yield new similar states over time



`SET_2 = add_elem(SET_1, y)`



# Basic OCaml

- Let's Start With C

```
double avg(int x, int y) {  
  double z = (double)(x + y);  
  z = z / 2;  
  printf("Answer is %g\n", z);  
  return z;  
}
```

# Basic OCaml

- Let's Start With C

```
double avg(int x, int y) {  
  double z = (double)(x + y);  
  z = z / 2;  
  printf("Answer is %g\n", z);  
  return z;  
}
```

```
let avg (x:int) (y:int) : float = begin
```

```
end
```

# Basic OCaml

- Let's Start With C

```
double avg(int x, int y) {  
    double z = (double)(x + y);  
    z = z / 2;  
    printf("Answer is %g\n", z);  
    return z;  
}
```

```
let avg (x:int) (y:int) : float = begin  
    let z = float_of_int (x + y) in  
  
end
```

# Basic OCaml

- Let's Start With C

```
double avg(int x, int y) {  
    double z = (double)(x + y);  
    z = z / 2;  
    printf("Answer is %g\n", z);  
    return z;  
}
```

```
let avg (x:int) (y:int) : float = begin  
    let z = float_of_int (x + y) in  
    let z = z /. 2.0 in  
  
end
```

# Basic OCaml

- Let's Start With C

```
double avg(int x, int y) {  
  double z = (double)(x + y);  
  z = z / 2;  
  printf("Answer is %g\n", z);  
  return z;  
}
```

```
let avg (x:int) (y:int) : float = begin  
  let z = float_of_int (x + y) in  
  let z = z /. 2.0 in  
  printf "Answer is %g\n" z ;  
  
end
```

# Basic OCaml

- Let's Start With C

```
double avg(int x, int y) {  
  double z = (double)(x + y);  
  z = z / 2;  
  printf("Answer is %g\n", z);  
  return z;  
}
```

```
let avg (x:int) (y:int) : float = begin  
  let z = float_of_int (x + y) in  
  let z = z /. 2.0 in  
  printf "Answer is %g\n" z ;  
  z  
end
```



# The Tuple (or Pair)

```
let x = (22, 58) in (* tuple creation *)
```

...

```
let y, z = x in (* tuple field extraction *)
```

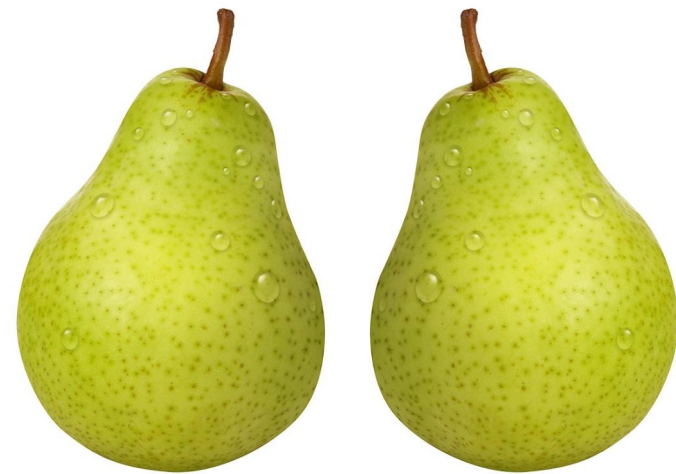
```
printf "first element is %d\n" y ; ...
```

```
let add_points p1 p2 =
```

```
  let x1, y1 = p1 in
```

```
  let x2, y2 = p2 in
```

```
  (x1 + x2, y1 + y2)
```



# List Syntax in OCaml

- Empty List `[ ]`
- Singleton `[ element ]`
- Longer List `[ e1 ; e2 ; e3 ]`
- Cons `x :: [y;z]` = `[x;y;z]`
- Append `[w;x]@[y;z]` = `[w;x;y;z]`
- `List.length`, `List.filter`, `List.fold`, `List.map` ...
- More on these later!
- Every element in list **must have same type**

# Functional Example

- Simple Functional Set (built out of lists)
  - let rec **add\_elem** (s, e) =
  - if **s = []** then **[e]**
  - else if **List.hd s = e** then **s**
  - else **List.hd s :: add\_elem(List.tl s, e)**
- Pattern-Matching Functional (same effect)
  - let rec **add\_elem** (s,e) = match **s** with
  - | **[]** -> **[e]**
  - | **hd :: tl** when **e = hd** -> **s**
  - | **hd :: tl** -> **hd :: add\_elem(tl, e)**

# Imperative Code

- More cases to handle

```
List* add_elem(List *s, item e) {  
    if (s == NULL)  
        return list(e, NULL);  
    else if (s->hd == e)  
        return s;  
    else if (s->tl == NULL) {  
        s->tl = list(e, NULL); return s;  
    } else  
        return add_elem(s->tl, e);  
}
```

*I have stopped reading Stephen King novels. Now I just read C code instead.*

- Richard O'Keefe

# Real-World Languages

- This Indo-European language spans 34 centuries of written records. It arose from Phoenician and in turn served as the basis for Latin and Cyrillic. It boasts a number of Western canon works, including the Odyssey, Iliad, Platonic dialogues, and Christian New Testament.

# Functional-Style Advantages

- Tractable program semantics
  - Procedures are functions
  - Formulate and prove assertions about code
  - More readable
- Referential transparency
  - Replace any expression by its value without changing the result
- No side-effects
  - Fewer errors

# Functional-Style Disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation
- Unfamiliar (to you!)
  - New programming style
- Not appropriate for every program
  - Operating systems, etc.

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

*17 small benchmarks*

# ML Innovative Features

- Type system
  - Strongly typed
  - Type inference
  - Abstraction
- Modules
- Patterns
- Polymorphism
- Higher-order functions
- Concise formal semantics

*There are many ways of trying to understand programs. People often rely too much on one way, which is called “debugging” and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.*

*- Robin Milner, 1997*



# Type System

- Type Inference
  - let rec **add\_elem** (s,e) = match s with
  - | [] -> [e]
  - | hd :: tl when e = hd -> s
  - | hd :: tl -> hd :: add\_elem(tl, e)
  - **val add\_elem :  $\alpha$  list \*  $\alpha$  ->  $\alpha$  list**
  - “ $\alpha$  list” means “List<T>” or “List< $\alpha$ >”
- ML infers types
  - Inconsistent or incomplete type is an error
- Optional type declarations (**exp : type**)
  - Clarify ambiguous cases, documentation

# Pattern Matching

- Simplifies Code (eliminates ifs, accessors)

```
type btree = (* binary tree of strings *)
```

```
  | Node of btree * string * btree
```

```
  | Leaf of string
```

```
let rec height tree = match tree with
```

```
  | Leaf _ -> 1
```

```
  | Node(x,_,y) -> 1 + max (height x) (height y)
```

```
let rec mem tree elt = match tree with
```

```
  | Leaf str -> str = elt
```

```
  | Node(x,str,y) -> str = elt ||
```

```
      mem x elt || mem y elt
```

# Pattern Matching Mistakes

- What if I forget a case?
  - `let rec is_odd x = match x with`
  - `| 0 -> false`
  - `| 2 -> false`
  - `| x when x > 2 -> is_odd (x-2)`
  - **Warning P: this pattern-matching is not exhaustive.**
  - **Here is an example of a value that is not matched: 1**

# Polymorphism

- Functions and type inference are polymorphic

- Operate on more than one type

- let rec length x = match x with

- | [] -> 0

- | hd :: tl -> 1 + length tl

- val length :  $\alpha$  list -> int

- length [1;2;3] = 3

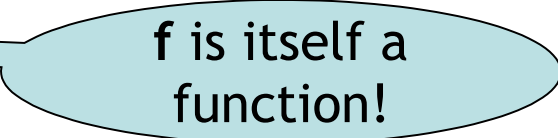
- length ["algol"; "smalltalk"; "ml"] = 3

- length [1 ; "algol" ] = ?

$\alpha$  means "any one type"

# Higher-Order Functions

- Function are first-class values
  - Can be used whenever a value is expected
  - Notably, can be passed around
  - Closure captures the environment
  - **let rec map f lst = match lst with**
  - **| [] -> []**
  - **| hd :: tl -> f hd :: map f tl**
  - **val map : ( $\alpha$  ->  $\beta$ ) ->  $\alpha$  list ->  $\beta$  list**
  - **let offset = 10 in**
  - **let myfun x = x + offset in**
  - **val myfun : int -> int**
  - **map myfun [1;8;22] = [11;18;32]**
- Extremely powerful programming technique
  - General iterators
  - Implement abstraction



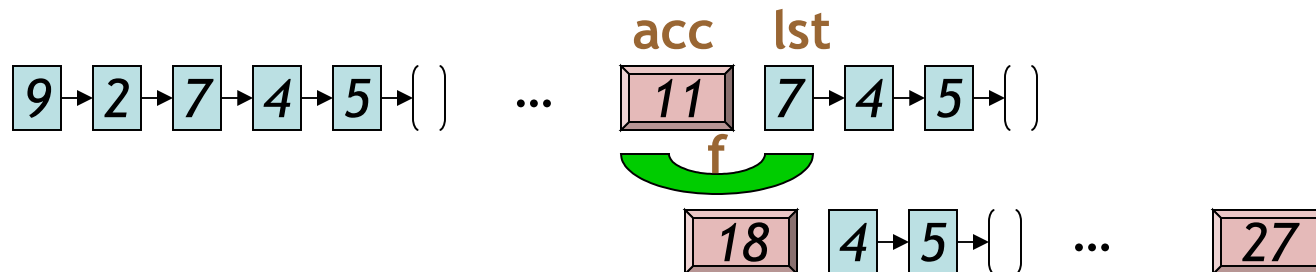
f is itself a function!

# The Story of Fold

- We've seen **length** and **map**
- We can also imagine ...
  - **sum** [1; 5; 8 ] = 14
  - **product** [1; 5; 8 ] = 40
  - **and** [true; true; false ] = false
  - **or** [true; true; false ] = true
  - **filter** (fun x -> x>4) [1; 5; 8] = [5; 8]
  - **reverse** [1; 5; 8] = [8; 5; 1]
  - **mem** 5 [1; 5; 8] = true
- Can we build all of these?

# The House That Fold Built

- The **fold** operator comes from Recursion Theory (Kleene, 1952)
  - let rec **fold** f acc lst = match lst with
  - | [] -> acc
  - | hd :: tl -> fold f (f acc hd) tl
  - val **fold** :  $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{ list} \rightarrow \alpha$
- Imagine we're summing a list (f = addition):



# It's Lego Time

- Let's build things out of Fold!
  - **length** lst = fold (fun acc elt -> acc + 1) 0 lst
  - **sum** lst = fold (fun acc elt -> acc + elt) 0 lst
  - **product** lst=fold (fun acc elt -> acc \* elt) 1 lst
  - **and** lst = fold (fun acc elt -> acc & elt) true lst
- How would we do **or**?
- How would we do **reverse**?





# Tougher Legos



- Examples:

- **reverse** lst = fold (fun acc e -> acc @ [e]) [] lst

- Note typing: (acc :  $\alpha$  list) (e :  $\alpha$ )

- **filter** keep\_it lst = fold (fun acc elt ->

- if keep\_it elt then elt :: acc else acc) [] lst

- **mem** wanted lst = fold (fun acc elt ->

- acc || wanted = elt) false lst

- Note typing: (acc : bool) (e :  $\alpha$ )

- How do we do **map**?

- Recall: map (fun x -> x +10) [1;2] = [11;12]

- Let's write it on the board ...

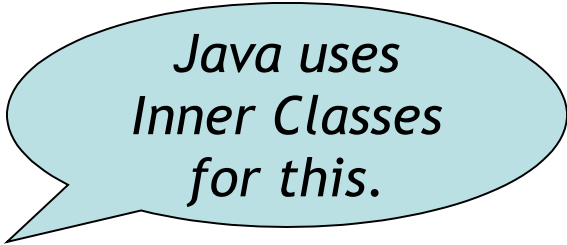
# Map From Fold

- let **map** myfun lst =
- fold (fun acc elt -> (myfun elt) :: acc) [] lst
  - Types: (myfun :  $\alpha \rightarrow \beta$ )
  - Types: (lst :  $\alpha$  list)
  - Types: (acc :  $\beta$  list)
  - Types: (elt :  $\alpha$ )
- How do we do **sort**?
  - (sort : ( $\alpha * \alpha \rightarrow \text{bool}$ ) ->  $\alpha$  list ->  $\alpha$  list)

*Do nothing which is of no use.*  
- Miyamoto Musashi, 1584-1645

# Sorting Examples

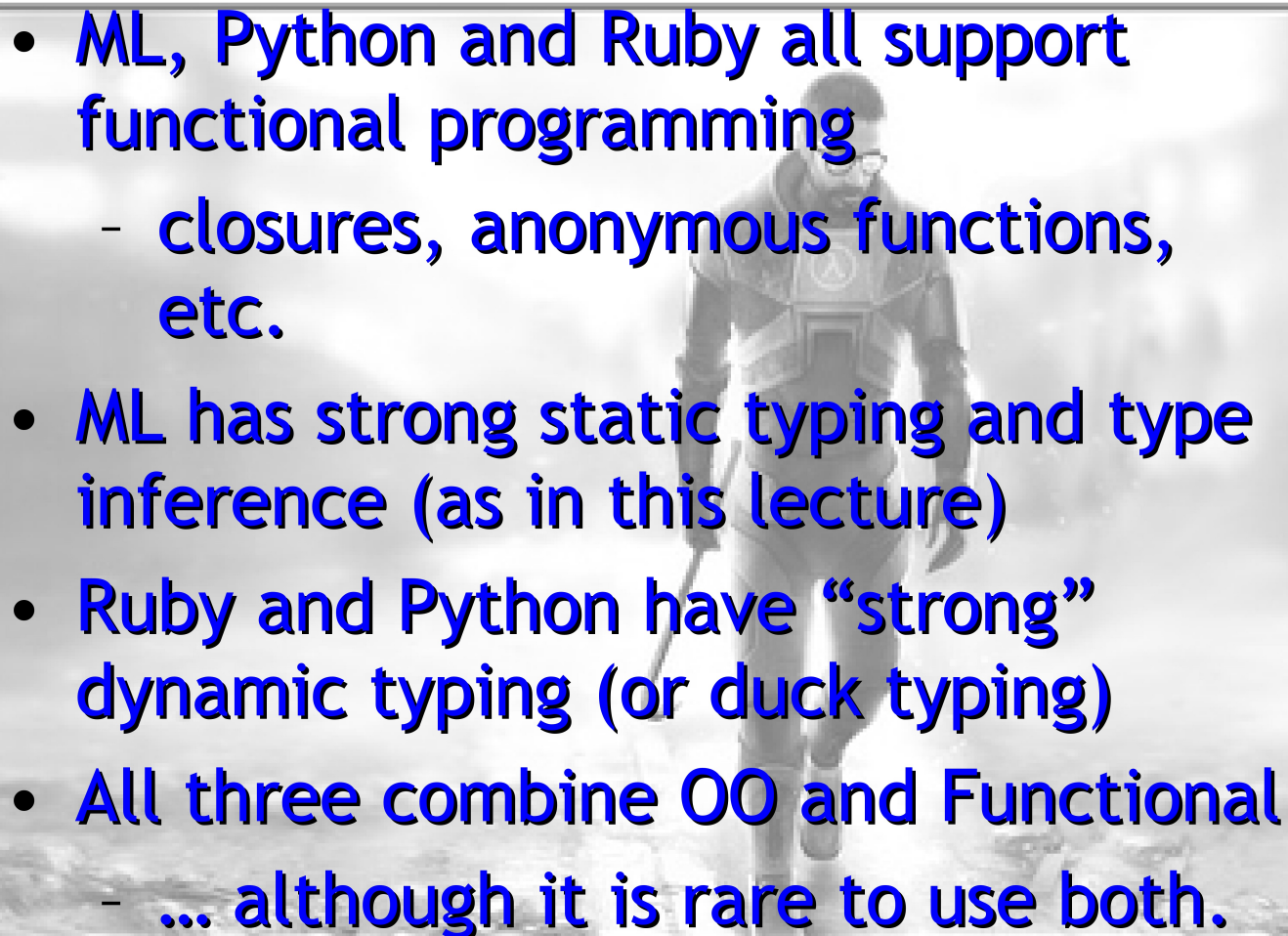
- `langs = [ "fortran"; "algol"; "c" ]`
- `courses = [ 216; 333; 415 ]`
- `sort (fun a b -> a < b) langs`
  - [ "algol"; "c"; "fortran" ]
- `sort (fun a b -> a > b) langs`
  - [ "fortran"; "c"; "algol" ]
- `sort (fun a b -> strlen a < strlen b) langs`
  - [ "c"; "algol"; "fortran" ]
- `sort (fun a b -> match is_odd a, is_odd b with`
  - | true, false -> true (\* odd numbers first \*)
  - | false, true -> false (\* even numbers last \*)
  - | \_, \_ -> a < b (\* otherwise ascending \*)) courses
  - [ 333 ; 415 ; 216 ]



*Java uses  
Inner Classes  
for this.*

# Partial Application and Currying

- let myadd x y = x + y
- **val myadd : int -> int -> int**
- myadd 3 5 = 8
- let addtwo = myadd 2
  - How do we know what this means? We use referential transparency! Basically, just substitute it in.
- **val addtwo : int -> int**
- addtwo 77 = 79
- Currying: “if you fix some arguments, you get a function of the remaining arguments”

- 
- **ML, Python and Ruby all support functional programming**
    - **closures, anonymous functions, etc.**
  - **ML has strong static typing and type inference (as in this lecture)**
  - **Ruby and Python have “strong” dynamic typing (or duck typing)**
  - **All three combine OO and Functional**
    - **... although it is rare to use both.**

# MULTIFUNCTIONALTY

One tool. One million uses.

# Homework

- Cool Reference Manual
- Backus Speedcoding
- PA1c