CSCI 334:
Principles of Programming Languages

Lecture 3: ML

Instructor: Dan Barowy

Williams

Topics

ML family of languages

F#

Your to-dos

1. Lab 1, **due Sunday 9/10** (partner lab)

Announcements

- CS Colloquium **tomorrow @ 2:35pm in Wege Auditorium (TCL 123)**

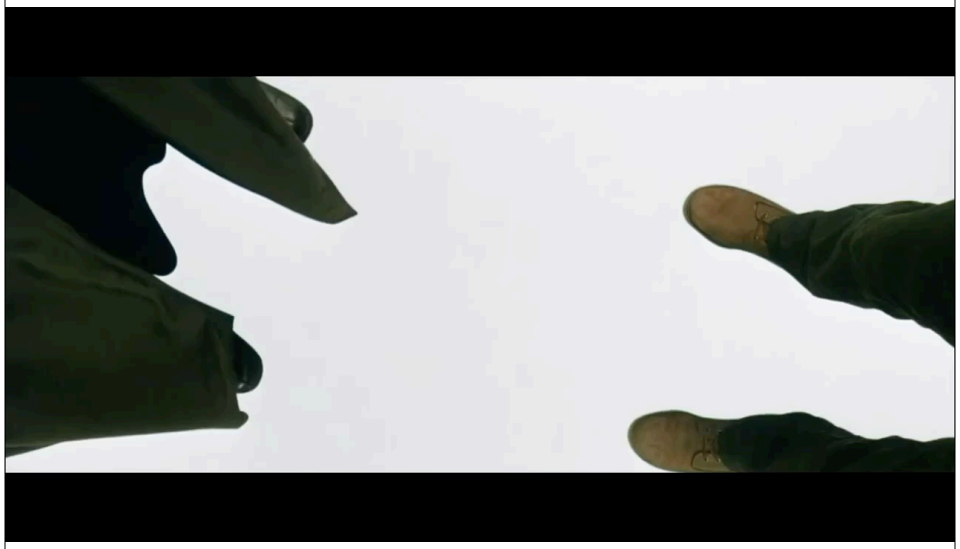
Ina Fiterau Brostean (UMass Amherst)

Machine Learning for Healthcare

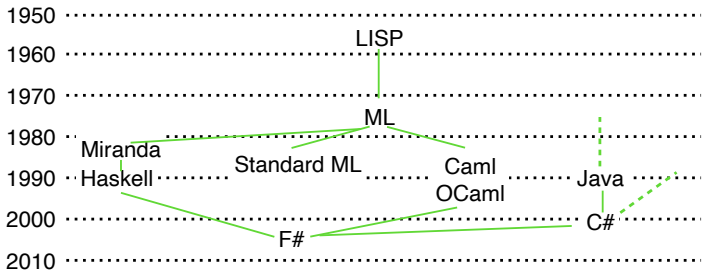
Fiterau's research lies at the intersection of machine learning and healthcare. Her Information Fusion Lab is currently working on a project combining features extracted from brain MRIs with patient demographics, test results, and contextual information, to detect Alzheimer's disease earlier than traditional diagnostics can.

ML

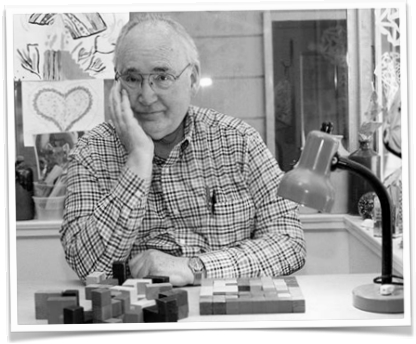
"Free your mind"



ML



ML



- Dana Scott
- Logic of Computable Functions
  - Can we automate proofs?
  - Yes. Theorem proving is essentially a "search problem"!
  - But proof search is "hard." Many problems are NP-Complete.
  - Works "in practice" with the right "tactics"

ML

- Robin Milner
- How to program tactics?
- A “meta-language” is needed
- ML is born (1973)
- First impression upon encountering a computer: "Programming was not a very beautiful thing. I resolved I would never go near a computer in my life."

F#

- Don Syme
- ML is “more fun” than Java or C#.
- Can we use ML instead?
- F# is born (2010).

Logical operators

Logical operators

operation	syntax
and	`&&`
not	`not`
equals	`=`
not equals	`<>`
inequalities	`<, >, <=, >=`

`unit`

`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`? `()`

You can also `ignore`…

```
> let foo() = 2;;
val foo : unit -> int

> foo();;
val it : int = 2

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

> foo() |> ignore;;
val it : unit = ()

>
```


"forward pipe" operator

```
<expr> |> <expr>

foo()  |> ignore
```

By the way…

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

By the way…

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

Primitives

Primitives

`bool`	`sbyte`
`byte`	`int16`
`int`	`uint16`
`single`	`uint`
`double`	`int64`
`char`	`uint64`
`unit`	`nativeint`
	`unativeint`
	`decimal`

† actually defined by the CLR

Combining forms

Functions

```
> let foo a b c = a;;
val foo: a: 'a -> b: 'b -> c: '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
```

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:

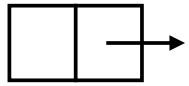
- the **empty list**, or
- a **node**, containing an element and a reference to a list.

Linked List

∅

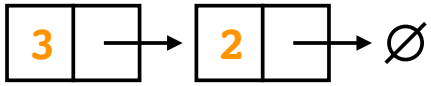
The empty list is defined as **nil** (or `[]`)

Linked List



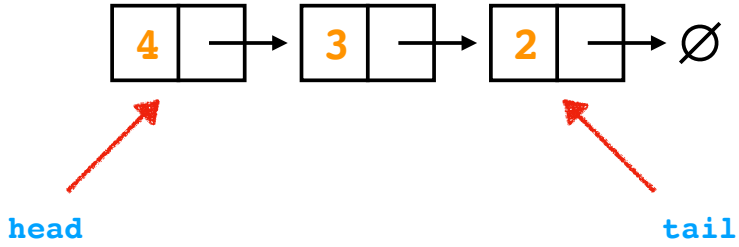
Every other list has at least one list node.

Linked List



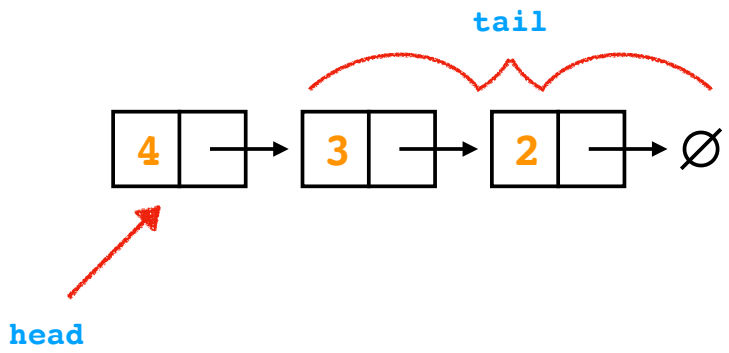
The last node in the list always points to **nil**.

Linked List



A list has parts.

Linked List



A list has parts.

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)
```

Functions on Lists

Let's define product…

```
> let rec product nums =
    if (nums = []) then
      1
    else
      (List.head nums)
       * product (List.tail nums);;

val product : int list -> int

> product [5; 2; 3];;
val it : int = 30
```

Pattern matching

```
let rec product nums =
  if (nums = []) then
    1
  else
    (List.head nums)
    * product (List.tail nums)
```

Using **patterns**…

```
let rec product nums =
  match nums with
  | []    -> 1
  | x::xs -> x * product xs
```

Recap & Next Class

Today:

History of ML

F#

Next class:

More F#