Lecture 10

Thursday, September 23, 2021 1:45 PM

"Tying the Knot"

```
let rec even x =
    if x=0 then true
    else not (even (x-1))
```

Challenge: Can we define even without rec?

```
let (* NOTE: No rec! *) even1 x =
let i = ref 0 in
let iEven = ref true in
let _ = while !i < x do
i := !i + 1;
iEven := not !iEven
done in
!iEven
```

i is a mutable value

It starts at 0

Increases over time

i dereferenced whenever used

iEven is also mutable

Starts at true

Flips over time

Invariant: Always: iEven = true $\iff$ i is even

Also: Assuming $X \geq 0$, always $i \leq X$

So when the loop stops, $i = X$.

Lesson to M: Think before posing challenge

Lesson No 1.1 : Think before posing challenge problems.

Tying the knot

1. `let dummy = ref (fun x -> true)`

   Create reference to function

2. `let even2 x = if x = 0 then true else not (!dummy (x - 1))`

   even2 works at least some of the time. (i.e. at x=0)

   Progress! :)

3. `dummy := fun x -> even2 x`

   Update dummy reference to point back at main function.

Trick is to use loops in space to achieve loops in time.

Useful to change function behavior online.

Equality in Ocaml

Two kinds: structural | physical

Checks if values have same "shape"

Checks if values point to same memory

Poly. `(=) : 'a -> 'a -> bool`

`(<>)`

`(==) : 'a -> 'a -> bool`

`(!=)`

When Base is open.

Rational numbers $\frac{1}{2} = \frac{2}{4}$

Intentional eq.

Representation

Extensional eq. $(1,2) \neq (2,4)$

Representational eq $\Rightarrow$ Intentional equality

$\not\Leftarrow$

Function equality

Do two Turing machines M_1 & M_2 compute the same function?

- Say there is a machine $M_=$ which computes this.
- How to check if an arbitrary T.M M halts?

  Build a machine M_∞ which never halts.

  Run $M_=$ on (M, M_∞).

. Checking Turing machines for equivalence is undecidable.

- Checking if two Ocaml functions "do the same thing" is undecidable.
- Be careful while checking functions for equality. Undefined.

"Programming in the Large"

Question: How to **organize** code?

Functions, classes, components,

front-end, back-end, ...

Organization, Sense 1: Physical, syntactic organization.

Break up code into smaller files

"Compilation units"

#include, #use, ...

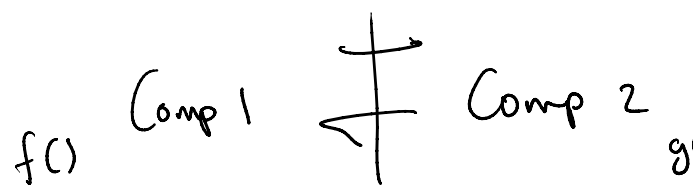
Sense 2: Organizing names

C++: namespaces scopes

Java: packages

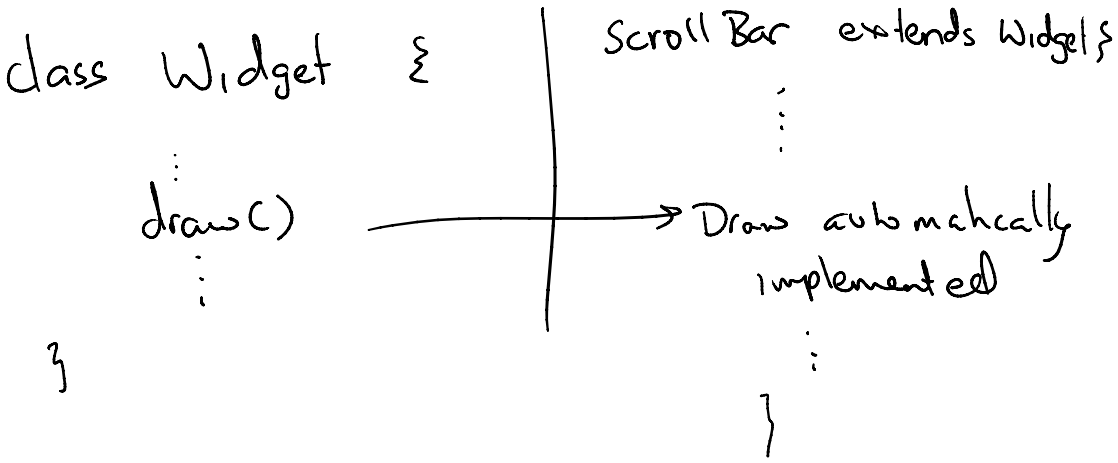
Sense 3: Abstraction

Components interact through well-defined boundaries



Classes, interfaces, traits, ...

Sense 4: Reuse of code



Inheritance ("ad-hoc polymorphism")

Parametric polymorphism (Ocaml, templates in C++, generics in Java/C#/...)

Ocaml: Module System

Two sublanguages:

① Vernacular (expressions, values, types, functions)

(2) Module language

structure — "implementation"

signature — "interface"

functor

C: Core / Preprocessor

C++: Core Templates Preprocessor

Lisp: Core Macros

Ocaml Module System

Signature

```
module type Stack =
sig
type 'a stack
val empty : 'a stack
val push : 'a -> 'a stack -> 'a stack
val peek : 'a stack -> 'a option
val pop : 'a stack -> 'a stack option
end
```

Types are mandatory

Interface exposed by all stacks

Structure

```
module ListStack : Stack =
struct
type 'a stack = 'a list
let empty = []
let push a s = a :: s
let peek s = List.hd s
let pop s = List.tl s
end
```

One specific implementation

Others possible.

For e.g.

```
module ADTStack : Stack =
```

→ Constructors

- Guarantees that implementation contains at least some things
- Guarantees that client uses the library in only permissible ways.

For e.g.

```
module ADTStack : Stack =
struct
type 'a stack = Empty | Cons of 'a * 'a
stack
let empty = Empty
let push a s = Cons(a, s)
let peek s = match s with
| Empty -> None
| Cons(hd, _) -> Some(hd)
let pop s = match s with
| Empty -> None
| Cons(_, tl) -> Some(tl)
end
```

Constructors

Type name

Functions supported

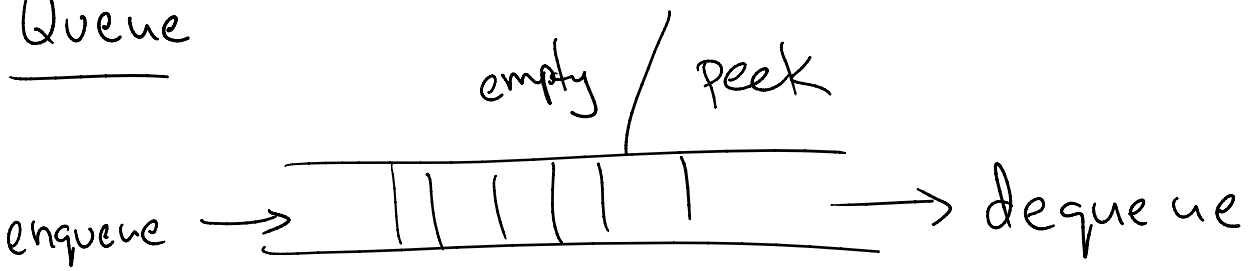
Also conforms to signature

But different implementations are incompatible.

If the signature is left unspecified, then client is unconstrained.

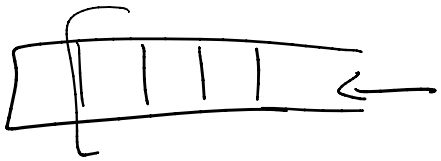
No encapsulation.

Queue



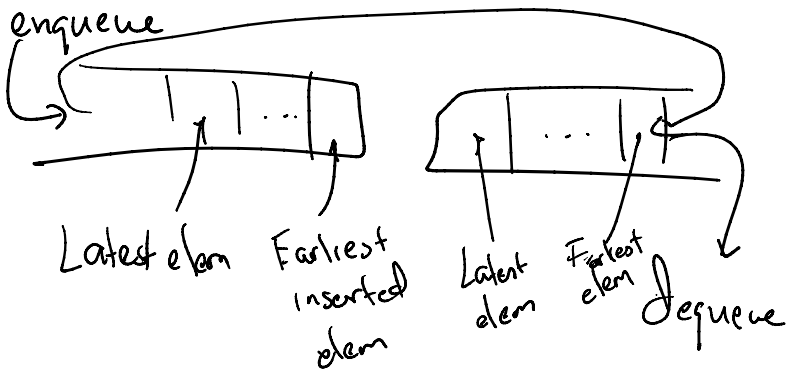
Implementation 1

- As a linked list



- enqueue : O(1) | O(n)
- dequeue : O(n) | O(1)
- peek : O(n) | O(1)

Implementation 2



- Enqueue : O(1)
- Dequeue: Worst case Peek O(n) time

Amortized O(1) time