Today’s music: ToneMatrix demo
[https://www.youtube.com/watch?v=TaeelzfVmc]
Review

So far:
• lots of language features
• syntax, static semantics (type checking), and dynamic semantics (evaluation)
• how to build small programs

Today:
• new language feature: modules
• how to build big programs: abstraction and specification
Question #1

What’s the largest program you’ve ever worked on, by yourself or as part of a team?

A. 10-100 LoC
B. 100-1,000 LoC
C. 1,000-10,000 LoC
D. 10,000-100,000 LoC
E. 100,000 LoC or bigger
Scale

- My PS2 solution: 366
- cs3110 tool: 2,200
- OCaml: 200,000
- Unreal engine: 2,000,000
- Windows 7: 40,000,000

http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

...can’t be done by one person
...no individual programmer can understand all the details
...too complex to build with subset of OCaml we’ve seen so far
Modularity

**Modular programming:** code comprises independent *modules*

– developed separately
– understand behavior of module in isolation
– reason locally, not globally
Java features for modularity

• classes, packages
  – organize identifiers (classes, methods, fields, etc.) into namespaces

• interfaces
  – describe related classes

• public, protected, private
  – control what is visible outside a namespace
OCaml features for modularity

• modules
  – organize identifiers (functions, values, etc.) into namespaces

• signatures
  – describe related modules

• abstract types
  – control what is visible outside a namespace
OCaml modules

Syntax:
module ModuleName = struct
  \textit{definitions}
end

\begin{itemize}
  \item the name must be capitalized
  \item definitions can be any definition we've previously seen in top-level or in file
    \begin{itemize}
      \item let, type, exception, etc.
    \end{itemize}
  \item creates a new namespace, must prefix values inside with name to access:
    \begin{itemize}
      \item module M = struct let x = 42 end
      \item let fortytwo = M.x
    \end{itemize}
  \item modules can be nested inside other modules
    \begin{itemize}
      \item i.e., definitions can also be modules
    \end{itemize}
  \item every file \texttt{myfile.ml} with contents $D$ is essentially wrapped in a module definition: \texttt{module Myfile = struct $D$ end}
\end{itemize}

Semantics: going on hiatus for awhile
Stack module

(* implement stacks as lists *)

module Stack = struct
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let pop s = match s with
    [] -> failwith "Empty"
    | x::xs -> (x,xs)
end

fst (Stack.pop (Stack.push 1 Stack.empty)) --> 1
Might seem backwards...

- In Java, might write
  
  ```java
  s = new Stack();
  s.push(1);
  s.pop();
  ```

- The stack is to the left of the dot, the method name is to the right

- In OCaml, it’s seemingly backward:
  
  ```ocaml
  let s = Stack.empty in
  let s' = Stack.push 1 s in
  let (one,_) = Stack.pop s'
  ```

- The stack is an argument to every function (common idiom is last argument)

- Just a syntactic detail (boring)

- Actually, the Java syntax is syntactic sugar:
  - Compiler can rewrite `s.push(1)` to `push(s,1)`
  - Method implementation in Java: every method receives its “this” argument as implicit first argument
Opening modules

• Write open ModuleName at top of file to “import” all definitions from module
  – Can write `push` instead of `Stack.push`
• Considered poor **idiom** to open lots of modules
  – Pollutes namespace: which module did `foo` come from?
  – Stylistic tradeoff between terseness and explicitness
  – Can do local opens instead:
    ```
    let one =
    let open Stack in
    fst (pop (push 1 empty))
    – Or locally bind short module name:
      ```
      ```
      let one =
      let module S = Stack in
      fst (S.pop (S.push 1 S.empty))
      ```
Opening modules

- Write open ModuleName at top of file to “import” all definitions from module
  - Can write `push` instead of `Stack.push`
- Considered poor **idiom** to open lots of modules
  - Pollutes namespace: which module did `foo` come from?
  - Stylistic tradeoff between terseness and explicitness
  - Can do local opens instead:
    ```
    let one =
      let open Stack in
      fst (pop (push 1 empty))
    ```
  - Or locally bind short module name:
    ```
    let one =
      let module S = Stack in
      fst (S.pop (S.push 1 S.empty))
    ```
Opening modules

- Write open ModuleName at top of file to “import” all definitions from module.
  - Can write push instead of Stack.push
- Considered poor idiom to open lots of modules:
  - Pollutes namespace: which module did foo come from?
  - Stylistic tradeoff between terseness and explicitness
- Can do local opens instead:
  - let one = let open Stack in fst (pop (push 1 empty))
  - Or locally bind short module name:
    - let module S = Stack in fst (S.pop (S.push 1 S.empty))

ONE DOES NOT SIMPLY

OPEN LIST

fst (S.pop (S.push 1 S.empty))
Decomposition

Modularity is about much more than namespace management

*Divide et impera* ... Divide and rule (or divide and conquer)

Decompose big problem into small subproblems:
- Each subproblem at same level of detail
- Each subproblem can be solved independently
- Solutions to subproblems combine to solve original problem

e.g., sorting with merge sort
- subproblem: divide list into pieces until each piece trivially sorted
- subproblem: merge two sorted lists into single sorted list

e.g., dynamic semantics of a programming language
- subproblem: divide language into syntactic pieces
- subproblem: give evaluation rules for each piece in isolation
Decomposition

Perhaps the most common difficulty: the sub-solutions don’t combine correctly

e.g., distributed knock-knock joke writing
e.g., distributed play writing
• subproblems: list of characters, lines of each character, vs.
• subproblems: number of acts, plot events in each act

Design tip: agree on division early; hard to change later

those subproblems are different abstractions of the problem
Abstraction

• Forgetting information
• Treating different things as though they were the same

e.g., biological classification
Abstraction of the Camel

- Domain
  - Eukarya
    - Eubacteria
    - Archaea
  - Animalia
    - Plantae
    - Fungi
- Kingdom
  - Annelida
  - Arthropoda
  - Chordata
- Phylum
  - Aves
  - Mammalia
  - Reptilia
- Class
  - Artiodactyla
  - Carnivora
  - Primates
- Order
  - Camelidae
  - Giraffidae
  - Hippopotamidae
- Family
  - Camelus
  - Loma
  - Vicugna
- Genus
  - bactrianus
  - dromedarius

Abstraction

- Forgetting information
- Treating different things as though they were the same

  e.g., animal kingdom
  e.g., files vs. block devices, inodes
  e.g., high-level programming languages vs. machine instruction set
  e.g., floating point arithmetic vs. idealized math
Computational Thinking

• Computational thinking is using abstraction and decomposition when... designing a large, complex system.
• Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.

Jeanette Wing
Corporate VP, MSR

https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf
Abstraction

Programming languages **pre-define abstractions**
- Data structures like lists
- Iterators like map and fold

Programming languages **enable definition of new abstractions**
- Procedural abstraction
- Data abstraction
- (Iteration abstraction)
Procedural Abstraction

Abstract from the details of a particular task, e.g.,

- \( \text{sqrt} : \) \( \text{float} \to \text{float} \)
- \( \text{List.sort} : \) \( ('a \to 'a \to \text{int}) \to 'a \text{ list} \to 'a \text{ list} \)

Abstract from how input is transformed into output

- Identity of particular input or output isn’t important
- But its type and any assumptions about it are
Data abstraction

Abstract from details of organizing data
• stacks, symbol tables, environments, bank accounts, polynomials, matrices, dictionaries, ...

Abstract from implementation of organization
• Actual code used to add elements (e.g.) isn’t important
• But types of operations and assumptions about what they do and what they require are important
OCaml Signatures

Syntax:
module type SIGNAME = sig
  declarations
end

– the name by convention is all caps
– declaration can be type or exception or a value declaration
  • val name : type
– e.g.
  • module type S = sig val x : int end
– creates a new namespace, must prefix declarations inside with name to access
– signatures can be nested inside other signatures
  • i.e., declarations can also be signatures
OCaml Signatures

Signatures are the “types” of modules

- module ModuleName : SIGNAME = struct ...
  end
- everything declared in SIGNAME must be defined in ModuleName
  • module type S1 = sig val x:int;; val y:int end
  • module M1 : S1 = struct let x = 42 end (* type error *)
- nothing except what’s declared in SIGNAME can be accessed from outside ModuleName
  • module type S2 = sig val x:int end
  • module M2 : S2 = struct let x = 42;; let y=7 end
  • M2.y (* type error *)

Signatures provide a mechanism for abstraction
Compilation units

Compilation unit = `myfile.ml + myfile.mli`

If `myfile.ml` has contents `DM` and `myfile.mli` has contents `DS` then OCaml behaves essentially as though:

```ocaml
module type MYFILESIG = sig
  DS
end
module Myfile : MYFILESIG = struct
  DM
end
```
Stack signature

module type STACK = sig
  val empty : 'a list
  val is_empty : 'a list -> bool
  val push : 'a -> 'a list -> 'a list
  val pop : 'a list -> 'a * 'a list
end

module Stack : STACK = struct
  ...
  (* as before *)
end
Stack Abstraction

• Procedural abstraction? Yes.
• Data abstraction? Not so much.
  – Not abstracting from details of lists
  – New OCaml feature: abstract types
    • In signature, just write “type t”
    • In module, write “type t = int list” (e.g.)
    • Inside module, it is known that t is a synonym for int list
    • Outside module, nothing is known about t.
      – It’s abstract
Int Stack with abstract types

module type STACK = sig
  type t
  val empty : t
  val is_empty : t -> bool
  val push : int -> t -> t
  val pop : t -> int * t
end

module Stack : STACK = struct
  type t = int list
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let pop s = match s with
           [] -> failwith "Empty"
           | x::xs -> (x,xs)
end
Stack with abstract types

```ocaml
module type STACK = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a * 'a t
end

module Stack : STACK = struct
  type 'a t = 'a list
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let pop s = match s with
    [] -> failwith "Empty"
    | x::xs -> (x,xs)
end

Now we have procedural and data abstraction!
```
Please hold still for 1 more minute

WRAP-UP FOR TODAY
Upcoming events

• PS3 released today
• Clarkson’s office hours today cancelled because of talk by visiting researcher

This is abstract.

THIS IS 3110