Modules and Stacks
Today's topics

• Some Ocaml surprises
• Why does check-error have that weird form?
• What type does "failwith" produce?
• Analysis of subsets (just to keep in practice)
• Signatures and Modules
  • The "stack" data structure as an example
Ocaml surprises 1: "redefinition"

```ocaml
# let a = 5;;
val a : int = 5
# let f = function x -> x + a;;
val f : int -> int = <fun>
# f 4;;
- : int = 9

# let a = -10;;
val a : int = -10
# f 4;;
- : int = 9
```
Ocaml surprises 2: "type redefinition"

```ocaml
# type shade = Gray of int;;
type shade = Gray of int
# let g = Gray 10;;
val g : shade = Gray 10
# type shade = Gray of int;;
type shade = Gray of int
# let h = Gray 10;;
val h : shade = Gray 10
# g = h;;
Error: This expression has type shade/1033
    but an expression was expected of type shade/1030
```
check_error

check_error (function () -> (transpose [])) "A matrix can't be 0-dimensional."

• Takes two args; the second is the expected contents of the "exception" produced by "failwith"
• The first is a procedure which, when called, should fail
• If we just had
  check_error (transpose []) "A matrix can't be 0-dimensional.";;
the "transpose" would fail, produce an error, and halt evaluation, so that check_error would never get
to compare the result to the expected result!
• Instead, check_error proc val invokes "proc () " inside a "try...catch" block, catches the resulting error,
  and compares its message to the expected value.
• "try...catch" blocks run code that may fail, and then try to fix things later
• "try...catch" blocks often used for dealing with uncertain things, like user input or network data or ...
What type does "failwith" have?

fun nth : int * 'a list -> 'a = function
  0,hd::tl -> hd
| 0,_ -> failwith "Can't get an element of an empty list"
| ...  

• Since "hd" is of type 'a, it must be that the result of "failwith" is of type 'a. And in fact failwith is of type string -> 'a

• But what does it actually produce?

• Nothing.
  • When failwith is called, ordinary processing halts.
Modules: a tool for organization
Barbara Liskov
(Turing Awardee)

https://www.youtube.com/watch?v=dtZ-o96bH9A
What happens as software gets big?

• It gets messy
• Taming the messiness is one of CS's big goals
• One solution: structures and interfaces
  • Example of a real-world interface: ATM
• Ocaml's version of this is modules
An example of a module: lists

- We have a collection of functions (like List.map, List.nth, ...) that all "play well together"
- All operate on similar kinds of data
Interfaces

• Sometimes we promise to provide a service, but it doesn't matter how it's done...just that it gets done.

• Example: we asked you all to transpose a matrix.
  • Some folks transposed the "rest" of the matrix and then consed something onto the front of each resulting row
  • Others mapped "rest" over the matrix, getting the last n-1 columns, transposed that, and then consed on one last row
  • Both got the same job done!

• The procedure specification and type signature constitute a tiny "interface": you don't need to know how things work, just that they DO work.

• If someone responsible for the "transpose" procedure decides to reimplement it, your program (which depends on transpose) still works!
Larger interfaces

• For modules, there's *also* a notion of an interface.
• A specification like that, in Ocaml, is called a module *signature*
• A bunch of code that does everything the signature requires
• A "signature" is like a type, but for modules rather than procedures.
An example module and signature: stacks
Stacks

• Represents something like a stack of playing cards
• Defined by allowed operations:
  • You can put something onto the top of the stack ("push")
  • You can remove something from the top of the stack ("pop")
  • You can look at the top item without removing it ("top") [sometimes "peek"]
  • You can create an empty stack
  • You can check whether a stack is empty
• Typically a stack contains items all of the same kind
  • ints, bools, ...
  • "processes" in your computer
  • ...
• This kind of description is called an "abstract data type" or ADT
An implementation of Stack

Name of module must start with capital letter

module ListStack =
struct
  type 'a stack = Stack of 'a list
  let empty = Stack []
  let is_empty: 'a stack -> bool = function s -> (s = empty)
  let push: 'a * 'a stack -> 'a stack = function datum, Stack lst -> Stack (datum :: lst)
  let pop = function
    | Stack [] -> failwith "Can't pop from empty stack."
    | Stack (hd::tl) -> Stack tl
  let top = function
    | Stack [] -> failwith "Empty stack has no top element."
    | Stack (hd::tl) -> hd
end
Ocaml's way of representing an ADT

- "Module type": says what a module must contain, but doesn't say how anything is done. MUST BE ALL CAPS.

```ocaml
module type STACK = sig
    type 'a stack
    val empty: 'a stack
    val is_empty: 'a stack -> bool
    val push : 'a * 'a stack -> 'a stack
    val pop : 'a stack -> 'a stack
    val top: 'a stack -> 'a
end
```
Type ascription and signature ascription

```ocaml
# let car = function a::rest -> a;;

Warning: pattern-matching not exhaustive.

val car : 'a list -> 'a = <fun>

# (car: int list -> int);;

- : int list -> int = <fun>

# let intcar = (car:int list -> int);;

val intcar : int list -> int = <fun>

# let intcar: int list -> int = car;;

val intcar : int list -> int = <fun>

Ascribing a signature to a module

module Stack : STACK = ListStack

module Stack = (ListStack: STACK)
```
Signature ascription and data hiding

- When we construct a module using signature ascription, some things visible without type ascription become hidden from us

- We can only see a ListStack through its interface
- Why would we want this?

- This lets us change our implementation of ListStack without breaking any program that uses it!
- Problems: testing is a pain
- Advantages: sometimes we find ourselves wanting to get at the underlying representation, and the "hiding" annoys us.
  - That's proof that it's doing its job!
Quiz: Write a signature for Queues.

A queue is a data structure. You can
- start with an empty queue using `empty`
- test if it is empty using `is_empty`
- add an element to the beginning using `enqueue`
- look at the last element using `peek`
- remove an element from the end using `dequeue`

For reference, here is the signature for stacks:

```plaintext
module type STACK =
  sig
    type 'a stack
    val empty: 'a stack
    val is_empty: 'a stack -> bool
    val push : 'a * 'a stack -> 'a stack
    val pop : 'a stack -> 'a stack
    val top: 'a stack -> 'a
  end
```
Practical matter

- To build a module that meets a signature...
- Start with the signature "module type FOO"
  - Change that to "module Foo" (or "module Foo:FOO" later)
  - Change "sig" to "struct" in the next line
  - Change each "val ....." to "let ..."
  - And then fill in each one