CS3110 Prelim Review

10/10/2018

Static Semantics

- All about **types**
- Does the program compile?

```
if false then 1 else "2"
```

Dynamic Semantics

- All about **values**
- What happens when a compiled program runs?

```
let x = 1 in x + x
```

Higher Order Functions

- Idea: functions are just another kind of value!

```
3             (* value of type int *)
"3"           (* value of type string *)
fun () -> 3 (* value of type unit -> int *)
```

Higher Order Functions (cont.)

Nothing special about defining a function with `let`:

```
let f x y = x + y
```

Just syntactic sugar for

```
let f = fun x y -> x + y
```

Partial Application

You can "fill in" just part of a function. For example, if you know certain variable values, you can plug those in while leaving the rest unchanged.

```
# let concat x y = x ^ y;;

# let name x = concat "my name is ";;

# name "Inigo Montoya";;
- : string = "my name is Inigo Montoya"
```

Polymorphism

Sometimes a function's implementation doesn't depend on the specific type of its input:

```
(* val int_id: int -> int *)
let int_id i = i

(* val string_id: string -> string *)
let string_id s = s
```

Polymorphism (cont.)

Sometimes a type's implementation can itself be parametrized on different types:

```
type int_list =
| Nil
| Cons of int * int_list

type string_list =
| Nil
| Cons of string * string_list
```

Polymorphism (cont.)

Solution: polymorphic types allow us to abstract over types themselves!

```
(* val id: 'a -> 'a *)
let id x = x

type 'a list =
| Nil
| Cons of 'a * 'a list
```

Polymorphism (cont.)

Can write *extremely* general functions:

```
(* val map: ('a -> 'b) -> ('a list) -> ('b list) *)
let map f = function
| []     -> []
| h :: t -> (f h) :: (map f t)

(* val compose: ('b -> 'c) -> ('a -> 'b) -> ('a -> 'c) *)
let compose f g = fun x -> f (g x)

(* val twice: ('a -> 'a) -> 'a -> 'a *)
let twice f x = f (f x)
```

Basic Patterns

```ocaml
match x with
| 3             -> (* primitive constant *)
| 'a'..'c'      -> (* character range *)
| x             -> (* variable name *)
| Constructor y -> (* tagged union member *)
| { name; hp }  -> (* record *)
| (a, b)        -> (* tuple *)
| h :: t        -> (* list *)
```

Advanced Patterns

```
(* Patterns in function arguments *)
let f (a, b) (c, d) = ...

(* Patterns in let expressions *)
let (a, b) = (1, 2) in ...

match x with
| (a, h :: t)  -> (* Nested patterns *)
| a when a > 5 -> (* Pattern guards *)
| 5 | 7 | 8    -> (* Or patterns *)
```

Lists

- Standard library `List` module functions:
  - `List.map`
  - `List.fold_left`
  - `List.find_opt`
  - etc.
- Recursive and tail-recursive functions
- Basic performance characteristics of linked lists

Tail Recursion

- Iterative vs. recursive processes
- Computing "on the way down" vs. "on the way up"
- Canonical example: `fold_left` vs. `fold_right`

Tail Recursion (cont.)

```
(* val fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
let fold_left f acc = function
| []     -> acc
| h :: t -> fold_left f (f acc h) t

(* val fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b *)
let fold_right f init = function
| []     -> init
| h :: t -> f h (fold_right f init t)
```

When is the work being done?

When do we need to keep track of previous stack frames?

Records

```
type movie = {name:string; year:int; actor:string}

(* Defining a value: *)
let l = {name = "Labyrinth"; year = 1986; actor = "David Bowie"}

(* Accessing elements *)
let goblin_king = l.actor

(* Pattern matching *)
match l with {name = n; year = y; actor = a} -> n

(* Pattern matching with field punning *)
match l with {name; year; actor} -> name
```

Variants

```
type cities = NYC | Boston | Chicago | LA | Houston

let c:cities = NYC

(* Pattern matching: *)
let city_population = function
  | NYC -> 8.54
  | Boston -> 0.67
  | Chicago -> 2.71
  | LA -> 3.98
  | Seattle -> 0.70
```

JSON

JSON is just another tree! (we'll talk about trees more soon)

```
type json =
| Null
| Bool of bool
| Int of int
| Float of float
| String of string
| Assoc of (string * json) list
| List of json list
| Tuple of json list
```

Modules

Good for code modularity and abstraction.

```
module Module = struct
        (* definitions *)
end
```

Modules (cont.)

```
module ListStack = struct
  type 'a t = 'a list
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let peek = function
  | []     -> failwith "Empty"
  | x :: _ -> x
  let pop = function
  | []      -> failwith "Empty"
  | _ :: xs -> xs
end
```

Signatures

- Controls what the client sees.
- Describes what a module does, but not how it implements it.

```
module type Signature = sig
  (* declarations *)
end

module Module : Signature = struct
  (* definitions *)
end
```

Signatures (cont.)

```
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 peek     : 'a t -> 'a
  val pop      : 'a t -> 'a t
end
```

Abstract Types

- Useful for information hiding and encapsulation
- Allows us to enforce invariants inside our module
- Prevents client code from being too tightly coupled to our implementation

Functors

- Similar motivation to polymorphic functions and types
- Parametrize module on some other signature
- Express dependency on **signature only**
- "As long as your module can do this, I can use it"
- Example: A4 and A5, where an `Engine` is parametrized on a `Dictionary`
- Implementation independence!

Functors (cont.)

```
module type mySig = sig
  val x : int
end

module IncX (M: mySig) = struct
  let x = M.x + 1
end
```

```
# module A = struct let x = 0 end;;
# A.x
- : int = 0
# module B = IncX(A);;
# B.x;;
- : int = 1
```

Trees

- Almost always a recursive variant
- Binary trees, abstract syntax trees, tries, etc.
- At least one base case
- At least one recursive case

Trees (cont.)

```
type 'a binary =
| Leaf
| Node of ('a * 'a binary * 'a binary)

type 'a trinary =
| Leaf
| Node of ('a * 'a trinary * 'a trinary * 'a trinary)

type ('a, 'b) wild =
| LeafA of 'a
| LeafB of 'b
| Node of (('a, 'b) wild * 'a * ('a, 'b) wild * 'b * ('a, 'b) wild)
```

Tree Traversal

Exercises coming soon!

Questions

- Attempting to answer questions from the Google form!

Anonymous Functions

Functions are values! Like other values (3, "sup", etc) we may not need to give them variable names.

For example:

```
# fun x -> x * x;;
```

takes in some value x, and returns its square. We can put it directly into another function. This can be helpful if we only need to use this anonymous function in that particular place.

For example, we can partially apply a function:

```
# let mul x = fun y -> x * y;; (* multiplies x and y *)

# let mul5 = mul 5;; (* multiplies 5 and y *)
```

We didn't need to give a name to the internal function (`fun y -> x * y`) because we don't use it elsewhere.

Now we can use the mul5 function to multiply anything by 5:

```
# mul5 4;;
- : int = 20
```

Abstraction

In your code, you may notice that you are doing similar operations to different input types (i.e. adding ints or contatenating strings). Abstraction lets us pull out the structure of these operations so we're not effectively copy-pasting code. Instead, we modify this base structure for each operation.

We will show some examples of abstraction in the following slides with maps and folding.

Maps

Apply a function to every element of a list

```
let rec map f = function
  |  [] -> []
  | h::t -> (f h)::(map f t)

(* Examples: *)
let add1 lst = map (fun x -> x+1) lst
let concat3110 = map (fun x -> x ^ "3110") lst
```

This also shows another use for anonymous functions! (i.e. `fun x -> x + 1`)

More examples

There are TONS of examples in the textbook!
See chapters 5.8 - 5.12

http://www.cs.cornell.edu/courses/cs3110/2018fa/textbook/modules/ex_c

Include vs. Open

Use `include` and `open` to make code from one module accessable in another.

Include: essentially the same as copy-pasting your code!

Open: Allows access to your "opened" module only within a limited scope and not any farther.

Include vs. Open example:

```
module LifeUniverseAndEverything = struct
   the_answer = 42
end

module type Stack = sig
  ...
end

module ListStack : Stack = struct
  include LifeUniverseAndEverything
  ...
end
```

We can now access `the_answer = 42` in our ListStack, but what if we used open instead of include?

Difference Between `<>` and `!=`

A: Let's use the = and == operators to make this simpler.

The = operator checks structural equality, i.e. are the values on either side of the operator equivalent?

```
3 = 3 (* true *)
```

The <> operator checks structural inequality - it returns true when the values are not the same.

The == operator checks the physical equality, i.e. are the values on either side of the operator *the same value*. This checks if their memory locations are the same.

```
"hello" == "hello" (* false! *)
```

Similarly, the != operator checks physical inequality - it returns true when the two values are stored in different memory locations.

Streams

Streams are infinite lists - they can be useful for user input, lists of number sequences, etc.

We need to delay evaluation so that we can get only a few elements of the stream at a time. Otherwise, we will get a stack overflow!

```
(* without delayed evaluation -
        will cause stack overflow! *)
type 'a stream =
  | Cons of 'a * 'a stream
```

```
(* With delayed evaluation *)
type 'a stream =
  Cons of 'a * (unit -> 'a stream)
```

To access a value after the first, we will need apply a function with unit input. For example, nats is the stream of natural numbers starting at 0:

```
# let rec from n =
  Cons (n, fun () -> from (n+1));;

val from : int -> int stream = <fun>

# let nats = from 0;;
val nats : int stream = Cons (0, <fun>)
```

Exam Topics Professor Clarkson is Very Excited About!

static semantics
dynamic semantics
higher-order functions
polymorphic functions
pattern matching
lists
tail recursion
records
association lists
variants
JSON

modules
module types
abstract types
functors
trees
tree traversals
queues
specifications
abstraction functions
representation invariants