Compiler Construction

Dan R. Ghica

“Philosophy”

- This is an **advanced** module
- Compiler and Languages ☞ learn **about** compilers
- Compiler Construction ☞ **build** a compiler
- (POPL recommended ☞ learn **about languages)**
- Good programming **skills** required
- **OCaml** required
- **Curiosity** and self-motivation are essential
- Practice some good **engineering**
    - **ongoing** project, **revisions** very likely
    - to team or not to team?
- This is a **fun** module

Teaching, learning & assessment

- each student will build a different compiler for a different(ish) language
- each week in the lecture I describe a new task
    - you need to **design**, **implement** and **test** it
- 2 hrs reading + 4 hrs independent work + 2 hrs lab
    - Fridays @ 14-16 // laptops?
- you must use **SVN** or **GIT** (public repositories)

Assessment

- **Final** exam 80% ——> 40% (hurdle)
- Class **tests** 20% ——> 10%
- "**Workshop**" 100% ——> 50% (hurdle)

- **inspect** your code (including test cases)
- run regression **tests**

- weekly marks starting with W2:
  - 4 : exceptional (1st, >70)
  - 3 : good (2.1, >60)
  - 2 : acceptable (2.2, >50)
  - 1 : pass (3rd, >40)

OCaml taster

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

let rec fold_tree f g = function
  | Node (l, r) → f (fold_tree f g l)
                    (fold_tree f g r)
  | Leaf x      → g x

let map_tree f =
 fold_tree (fun l r → Node (l, r))
           (fun x   → Leaf (f x))

let max_tree = fold_tree max (fun x → x)

let t' = map_tree (fun x → 10 * x) t
```

Some new (?) OCaml features

Polymorphic variant records

```
let rec fold_tree f g = function
 | `Node (l, r) → f (fold_tree f g l) (fold_tree f g r)
 | `Leaf x      → g x

let map_tree f = fold_tree
                 (fun l r → `Node (l, r))
                 (fun x   → `Leaf (f x))
```

```
let t1 = `Leaf 1
let t2 = `Leaf 2
let t3 = `Leaf 3
let t4 = `Leaf 4
let t5 = `Node (t1, t2)
let t6 = `Node (t3, t4)
let t7 = `Node (t5, t6)

let max_tree = fold_tree max (fun x → x)

let mt = max_tree t7
let t7' = map_tree (fun x → 10 * x) t7
```

Polymorphic variant records

```
val t1 : [> `Leaf of int ] = `Leaf 1

val t5 : [> `Node of [> `Leaf of int ] * [>
`Leaf of int ] ] = `Node (`Leaf 1, `Leaf 2)

val t7 : [> `Node of [> `Node of [> `Leaf of int
] * [> `Leaf of int ] ] * [> `Node of [> `Leaf
of int ] * [> `Leaf of int ] ] ] = `Node (`Node
(`Leaf 1, `Leaf 2), `Node (`Leaf 3, `Leaf 4))
```

Polymorphic variant records

```
val fold_tree :
  ('a → 'a → 'a) →
  ('b → 'a) →
  ([< `Leaf of 'b | `Node of 'c * 'c ] as 'c) →
  'a = <fun>
```

```
val map_tree :
  ('a → 'b) →
  ([< `Leaf of 'a | `Node of 'c * 'c ] as 'c) →
  ([> `Leaf of 'b | `Node of 'd * 'd ] as 'd)
  = <fun>
```

More maintainable but…

Powerful // dangerous // understand // less efficient

Arrays

```
# let a = [| 1; 2; 3; 4 |] ;;
val a : int array = [|1; 2; 3; 4|]
# a.(1) ← 0;;
- : unit = ()
# a;;
- : int array = [|1; 0; 3; 4|]
```

Mutables

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# x := !x + 1;;
- : unit = ()
# !x;;
- : int = 2
```

```
# let f = ref (fun x → 2 * x);;
val f : (int → int) ref = {contents =
<fun>}
# f := fun x → 3 * x;;
- : unit = ()
# !f 4;;
- : int = 12
```

Weak polymorphism

```
# let x = ref [];;
val x : '_a list ref = {contents = []}
# x := [1];;
- : unit = ()
# x;;
- : int list ref = {contents = [1]}
# x := ['a'];; TYPE ERROR!

# let f = ref (fun x → x);;
val f : ('_a → '_a) ref = {contents = <fun>}
# !f 1;;
- : int = 1
# f;;
- : (int → int) ref = {contents = <fun>}
```

Quick Quiz:

What is the type of x

```
# let x = [| |];;
val x : 'a array = [||]
```

Read more in RWOC

https://realworldocaml.org/v1/en/html/imperative-programming-1.html#side-effects-and-weak-polymorphism

Records & “punning”

```
# type point = {x : int ; y : int; } ;;
type point = { x : int; y : int; }
# let px = { x = 1; y = 3 } ;;
val px : point = {x = 1; y = 3}
# px.x;;
- : int = 1
  (* Punning *)
# let x = 1;;
val x : int = 1
# let y = 3;;
val y : int = 3
# let px' = {x; y; } ;;
val px' : point = {x = 1; y = 3}
# let px'' = {y; x; } ;;
val px'' : point = {x = 1; y = 3}
```

Labeled and default args

```
# let divide ~dvd ~dvs = dvd / dvs ;;
val divide : dvd:int → dvs:int → int = <fun>
# divide 9 3 ;;
- : int = 3
# divide ~dvs:3 ~dvd:9 ;;
- : int = 3
```

```
# let concat ?(sep="") fst snd
      = fst ^ sep ^ snd ;;
val concat : ?sep:string → string → string →
string = <fun>
# concat "Hello" "Ocaml";;
- : string = "HelloOcaml"
# concat ~sep:" " "Hello" "OCaml";;
- : string = "Hello OCaml”
```

TODO

- Reference: **Real World OCaml**
  - brush up on OCaml
- **self-test assignment (tomorrow)**
  - learn (basics) & set up GIT / SVN repository
  - set up OCaml (and OPAM)
  - compiling multi-source OCaml programs
    - ocamlfind, ocamlbuild, make
    - RWOC: I.4, III.22