CS 11 Ocaml track: lecture 1

- Preliminaries
  - Need a CS cluster account
    - http://www.cs.caltech.edu/
      cgi-bin/sysadmin/account_request.cgi
  - Need to know UNIX
    - ITS tutorial linked from track home page
  - Track home page:
    - www.cs.caltech.edu/courses/cs11/material/ocaml

Assignments

- 1st assignment is posted now
- Due one week after class, midnight
- Late penalty: 1 mark/day
- Redos

Redos

- 1st redo = 1 mark off
- 2nd redo = 1 to 2 more marks off
- 3rd redo = 1 to 3 more marks off
- No 4th redo! Grade - 6 mark penalty

Passing

- Need average of 7/10 on labs
- 6 labs → 42/60 marks

Other administrative stuff

- See admin web page:

http://www.cs.caltech.edu/courses/cs11/material/ocaml/admin.html

- Covers how to submit labs, collaboration policy, grading, etc.

Textbook

- Introduction to Objective Caml

  by Jason Hickey

  - draft (don't redistribute)

Ocaml: pros

- Ocaml is a very nice language!
- Strong static type system
  - catches lots of errors at compile time
- Very expressive type system
  - first-class functions
  - polymorphic types
  - algebraic data types
    - makes it easy to build complex data types
  - references for mutable data

Ocaml: pros

- Garbage collection
- Byte-code and native-code compilers
- Very fast!
  - very competitive with C and C++
  - especially if data structures are very complex
- Interactive interpreter for experimenting
- Clean design
- Can interface with C fairly easily

Ocaml: pros

- Fully supports several different programming paradigms:
  - functional programming
  - imperative programming
  - object-oriented programming
- Most natural to use as a "mostly-functional" language
- Safe language: no core dumps!

But wait! There's more!

- Type inference to get benefits of static typing without having to write out tons of declarations
- Very powerful module system
  - including separate compilation of modules
- Parameterizable modules (functors)
- Simple and powerful exception handling system
- Plus more experimental features

Ocaml: cons

- Very few bad things about ocaml
- Native-code compiler doesn't support shared libraries
  - though 3rd-party tools can do this
- Type system sometimes too rigid
- Object system doesn't support "downcasting" *i.e.* "instanceof"

Ocaml: cons

- Messy, ambiguous syntax
- "Operator underloading"
  - + to add integers
  - +. to add floats
- For purists: not as purely functional as *e.g.* Haskell
- Some messy aspects of type system
  - "polymorphic references"

Ocaml: uses

- Great language for writing compilers!
- Also great for writing theorem provers
- Recently, Ocaml used for tasks in many other areas:
  - simulations
  - finance
  - operating systems
  - etc.

Ocaml: uses

- Can compete successfully with C/C++
- Especially when
  - safety is important
  - data structures are very complex
- In these cases, can often outperform C/C++
- Example: Ensemble system re-written from C → Ocaml; new version faster

Ocaml: uses

- Why should Ocaml give faster code in those cases?
- After all, C/C++ "closer to the machine"
- Answer:
  - easier to tweak very complex algorithms in ways that would overwhelm C/C++ programmers
  - and still have correct, working code

Ocaml: history

- Ocaml is a dialect of the "ML" language
  - ML originally the "meta-language" for a theorem-proving program called "LCF"
    - "Logic for Computable Functions"

Ocaml: history

- Adapted into a language called CAML by researchers in INRIA (France)
  - "Categorical Abstract Machine Language"
  - Newer versions have a very different internal structure, but kept name
- "Ocaml" is "Objective Caml"
  - CAML with object-oriented extensions
  - Prime candidate for worst computer language name of all time

Our emphasis

- In this track, we will focus on Ocaml's use as a functional programming language
- We will also cover imperative aspects
  - but not OO features
- Good preparation for *e.g.* CS 134b (compiler course)

Functional programming

- What is a functional programming language?
- It's a language that
  - treats functions as "first-class" data
- Meaning?
- Functions can be
  - passed as arguments
  - created on-the-fly
  - returned as a result from other functions

Functional programming

- Other aspects of FP:
- Data should be persistent
  - names, once bound, do not get rebound
  - (unless they are function arguments)
  - mutable data structures like arrays avoided
  - in favor of non-mutable data structures like singly-linked lists
- Assignment statements rarely used
- Explicit loops rarely used; use recursion instead
- Higher-order functions used a lot

Functional programming

- Learning the syntax of Ocaml is relatively easy
- Learning to program in a "functional style" is much harder
- Main goal of track is to force you to learn to think this way
- (If you've taken CS 1, you already know how to think this way)

Getting started

- The interactive interpreter is just called **`ocaml`**
- Get out of it by typing control-D (**`^D`** AKA end-of-file)
- When inside, can do essentially anything that could be done in a file
  - define functions
  - define types
  - run code

Getting started

- The "hello, world!" program (sort of):

```
% ocaml
        Objective Caml version 3.08.3

# Printf.printf "hello, world!\n";;
hello, world!
- : unit = ()
^D
%
```

Getting started

- The "hello, world!" program (sort of):

```
% ocaml
        Objective Caml version 3.08.3

# Printf.printf "hello, world!\n";;
hello, world!
- : unit = ()
^D
%
```

prompt

Getting started

- The "hello, world!" program (sort of):

```
% ocaml
        Objective Caml version 3.08.3

# Printf.printf "hello, world!\n";;
hello, world!
- : unit = ()
^D
%
```

statement

Getting started

- The "hello, world!" program (sort of):

```
% ocaml
        Objective Caml version 3.08.3

# Printf.printf "hello, world!\n";;
hello, world!
- : unit = ()
^D
%
```

side effect

Getting started

- The "hello, world!" program (sort of):

```
% ocaml
        Objective Caml version 3.08.3

# Printf.printf "hello, world!\n";;
hello, world!
- : unit = ()
^D
%
```

result name, type and value

Getting started

- In interactive interpreter, signal that you want interpreter to process your code by typing two semicolons (`;;`)
- This is not necessary for source code in files
  - So don't put it in! It's annoying to read.

Stand-alone executables (1)

- Consider this file called **`hello.ml`**:

```
let _ = Printf.printf "hello, world!\n"
```

- Compile to executable thusly:

```
% ocamlc hello.ml -o hello
% hello
hello, world!
```

Stand-alone executables (2)

- Can also do this:

```
% ocamlopt.opt hello.ml -o hello
% hello
hello, world!
```

- Generates native code; previous version generated byte code

All right, then...

- Now we'll start talking about the language itself
- Very sketchy; see textbook for more details
- Also see ocaml manual on website
  - http://www.ocaml.org

Basic data types (1)

- unit `()`
- bool `false true`
- int `1 2 3 4 -1 0 42`
- float `1.0 3.14 2.71828`
- char `'c' 'h' '\n' '\\' '\''`
- string `"this is a string"`

Basic data types (2)

- lists `["this"; "is"; "a"; "list"]`
  - all elements of a list must be the same type!
- arrays `[| 1.0; 2.0; 3.0; 4.0 |]`
- references `ref 0`
- tuples `(1, "two", 3.0)`
  - elements of tuple don't have to be of same type
  - but each particular tuple has the type which is the product of its constituent types!
  - here, type is `int * string * float`

Operators

- int `+   -   *   /`
- float `+.   -.   *.   /.`
- string `^` (string concatenation)
- lists `::` ("cons")
- lists `@` (list concatenation)
- reference `!` (dereference)
- reference `:=` (assignment to)

`let` expressions

```
# let x = 10 in x + x;;
- : int = 20
# let x = 10 in
  let y = 20 in
      x + y;;
- : int = 30
```

- Scope of name extends to end of `let` expression

Defining functions (1)

```
# let f x = 2 * x - 3;;
val f : int -> int = <fun>
# f 4;;
- : 5 = int
```

Defining functions (2)

```
# let rec sum_to x =
    if x = 0 then 0
    else x + sum_to (x - 1);;
val sum_to : int -> int = <fun>
# f 10;;
- : 55 = int
```

- Need **`let rec`** to define recursive functions, not just **`let`**

Pattern matching (1)

```
# let rec sum_to x =
    match x with
      0 -> 0
    | x' -> x' + sum_to (x' - 1)
;;
```

- Note: can use single quote (`'`) as a character in identifiers

Pattern matching (2)

```
# let rec list_length lst =
    match lst with
      [] -> 0
    | h :: t -> 1 + list_length t
;;
```

- Pattern matching usually simpler than explicit if statement
- Also can match deeply nested patterns
  - can make code much more readable

Higher-order functions (1)

```
# let rec filter f lst =
    match lst with
      [] -> []
    | h :: t ->
      if (f h) then (h :: (filter f t))
      else (filter f t)
;;
```

- Create new list from old list (all elements where **`f x`** is **`true`**)

Higher-order functions (2)

```
# filter (fun x -> x mod 2 = 0)
  [1;2;3;4;5] ;;
- : int list = [2; 4]
```

- **`fun`** is ocaml's equivalent of a lambda expression (anonymous function)

Pattern guards

```
# let rec filter f lst =
    match lst with
      [] -> []
    | h :: t when (f h) ->
      (h :: (filter f t))
    | h :: t -> (filter f t)
;;
```

- Same meaning as previous filter function

Tail recursion (1)

```
# let sum lst =
    let rec sum_iter rest sum =
      match rest with
        [] -> sum
      | h :: t -> sum_iter t (sum + h)
    in
      sum_iter lst 0
;;
```

Tail recursion (2)

- Two interesting things in sum code:
  - helper function `sum_iter` in the body of `sum`
  - `sum_iter` is tail recursive
  - meaning: recursive call has no pending operations to complete once it returns
  - significance?
  - executes in a constant amount of space
    - highly desirable!

That's all for now

- Lab 1 is up
- Several small functions to write
- Get practice in all these aspects of language
- Have fun!