CMSC 330: Organization of Programming Languages

Functional Programming with OCaml

What is a functional language?

A functional language:

- defines computations as **mathematical functions**
- discourages use of mutable **state**

`x = x + 1` ?

Functional vs. Imperative Programming

- Imperative programming
  - focuses on how to execute, defines control flow as statements that change a program state.

- Functional programming
  - treats programs as evaluating mathematical functions and avoids state and mutable data

Imperative Programming

Commands specify **how** to compute, by destructively changing state:

```
x = x+1;
a[i] = 42;
p.next = p.next.next;
```

The fantasy of changing state(mutability):

- It's easy to reason about: the machine does this, then this...
- Machines are good at complicated manipulation of state

Imperative Programming: Reality

Thread 1 on CPU 1

```
x = x+1;
a[i] = 42;
p.next = p.next.next;
```

Thread 2 on CPU 2

```
x = x+1;
a[i] = 42;
p.next = p.next.next;
```

- There is no single state
  - Programs have many threads, spread across many cores, spread across many processors, spread across many computers...
  - each with its own view of memory

Imperative Programming

Functions/methods have **side effects**:

```
int cnt = 0;//global

int f(Node *r) {
      r->data = cnt;
      cnt++;
      return cnt;
}
```

- mutability breaks referential transparency: ability to replace an expression with its value without affecting the result.

$$\texttt{f(x) + f(x) + f(x)} \neq \texttt{3 f (x)}$$

Functional vs. Imperative

Functional languages:

- *Higher* level of abstraction
- *Easier* to develop robust software
- *Immutable* state: easier to reason about software

Imperative languages:

- *Lower* level of abstraction
- *Harder* to develop robust software
- *Mutable* state: harder to reason about software

Functional programming

Expressions specify **what** to compute

- Variables never change value
  - Like mathematical variables
- Functions (almost) never have side effects

The reality of immutability:

- No need to think about state
- Easier (and more powerful) ways to build correct programs and concurrent programs

Key Features of ML

- First-class functions
  - Functions can be parameters to other functions (“higher order”) and return values, and stored as data
- Favor immutability (“assign once”)
- Data types and pattern matching
  - Convenient for certain kinds of data structures
- Type inference
  - No need to write types in the source language
    - But the language is statically typed
  - Supports parametric polymorphism
    - *Generics* in Java, *templates* in C++
- Like Ruby, Java, ...: exceptions and garbage collection

Why study functional programming?

Functional languages predict the future:

- Garbage collection
  - Java [1995], LISP [1958]
- Parametric polymorphism (generics)
  - Java 5 [2004], ML [1990]
- Higher-order functions
  - C#3.0 [2007], Java 8 [2014], LISP [1958]
- Type inference
  - C++11 [2011], Java 7 [2011] and 8, ML [1990]
- Pattern matching
  - ML [1990], Scala [2002], Java *X* [201?]
    - http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html

Why study functional programming?

Functional languages in the real world

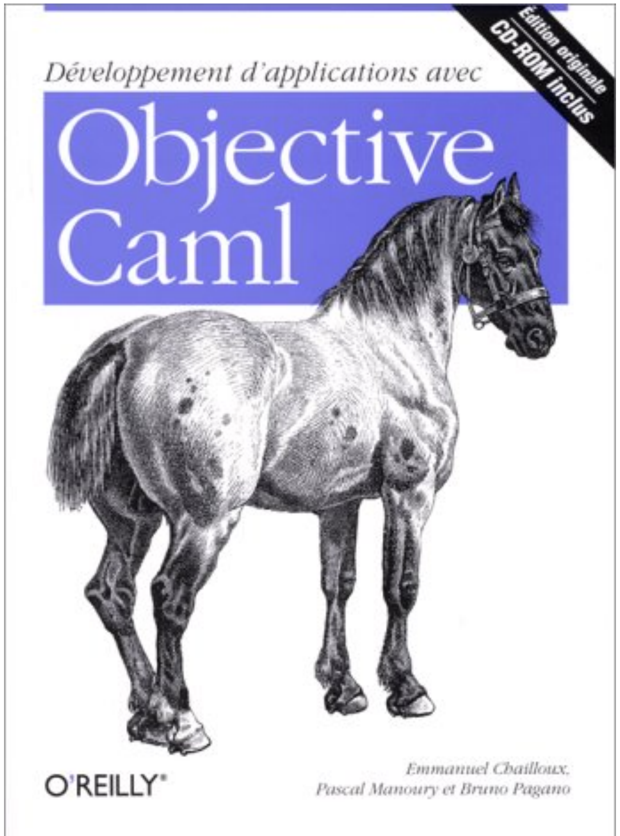
- Java 8
- F#, C# 3.0, LINQ
- Scala
- Haskell
- Erlang
- OCaml

https://ocaml.org/learn/companies.html

ML-style (Functional) Languages

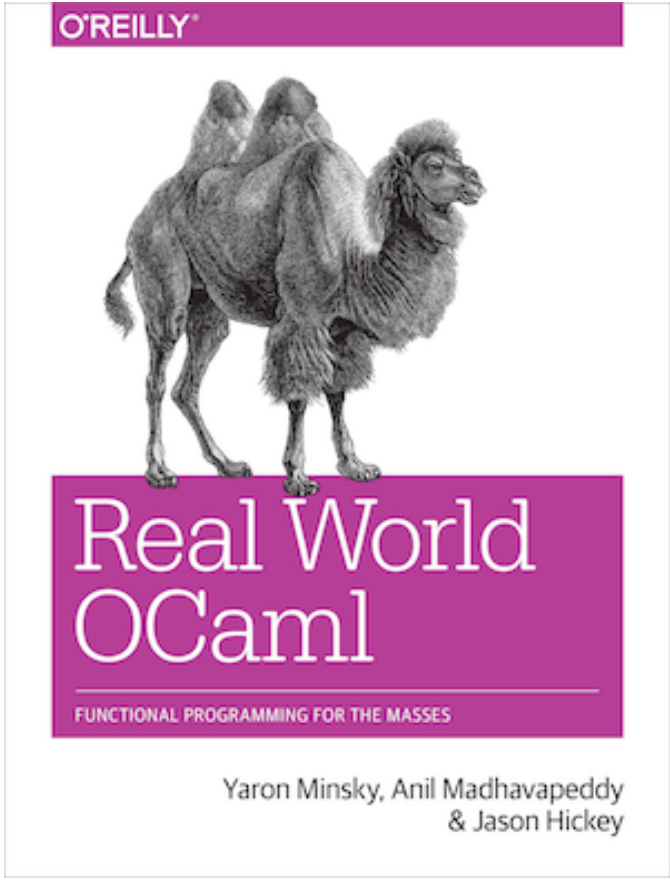
- ML (Meta Language)
  - Univ. of Edinburgh, 1973
  - Part of a theorem proving system LCF
- Standard ML
  - Bell Labs and Princeton, 1990; Yale, AT&T, U. Chicago
- OCaml (Objective CAML)
  - INRIA, 1996
    - French Nat'l Institute for Research in Computer Science
  - O is for "objective", meaning objects (which we'll ignore)
- Haskell (1998): *lazy* functional programming
- Scala (2004): functional and OO programming

Useful Information on OCaml language



- Translation available on the class webpage
  - *Developing Applications with Objective Caml*
- Webpage also has link to another book
  - *Introduction to the Objective Caml Programming Language*

More Information on OCaml



- Book designed to introduce and advance understanding of OCaml
  - Authors use OCaml in the real world
  - Introduces new libraries, tools
- Free HTML online
  - realworldocaml.org

Coding Guidelines

- We will not grade on style, but style is important
- Recommended coding guidelines:

- https://ocaml.org/learn/tutorials/guidelines.html

Working with OCaml

OCaml Compiler

- OCaml programs can be compiled using ocamlc
  - Produces .cmo (“compiled object”) and .cmi (“compiled interface”) files
    - We’ll talk about interface files later
  - By default, also links to produce executable a.out
    - Use -o to set output file name
    - Use -c to compile only to .cmo/.cmi and not to link
- Can also compile with ocamlopt
  - Produces .cmx files, which contain native code
  - Faster, but not platform-independent (or as easily debugged)

OCaml Compiler

- Compiling and running the following small program:

hello.ml:

```
(* A small OCaml program *)
print_string "Hello world!\n";;
```

```
% ocamlc hello.ml
% ./a.out
Hello world!
%
```

OCaml Compiler: Multiple Files

main.ml:

```
let main () =
  print_int (Util.add 10 20);
  print_string "\n"

let () = main ()
```

util.ml:

```
let add x y = x+y
```

- Compile both together (produces `a.out`)

```
ocamlc util.ml main.ml
```

- Or compile separately

```
ocamlc –c util.ml
ocamlc util.cmo main.ml
```

- To execute

```
./a.out
```

OCaml Top-level

- The *top-level* is a read-eval-print loop (REPL) for OCaml
  - Like Ruby's **`irb`**
- Start the top-level with the **`ocaml`** command:

```
ocaml
    OCaml version 4.07.0
# print_string "Hello world!\n";;
Hello world!
- : unit = ()
#
```

- To exit the top-level, type ^D (Control D) or call the exit 0

```
# exit 0;;
```

OCaml Top-level (cont'd)

Expressions can also be typed and evaluated at the top-level:

```
# 3 + 4;;
- : int = 7

# let x = 37;;
val x : int = 37

# x;;
- : int = 37

# let y = 5;;
val y : int = 5

# let z = 5 + x;;
val z : int = 42

# print_int z;;
42- : unit = ()

# print_string "Colorless green ideas sleep furiously";;
Colorless green ideas sleep furiously- : unit = ()

# print_int  "Colorless green ideas sleep furiously";;
This expression has type string but is here used with type int
```

gives type and value of each expr

"-" = "the expression you just typed"

unit = "no interesting value" (like void)

Loading Files in the Top-level

File **hello.ml** :

```
print_string "Hello world!\n";;
```

- Load a file into top-level

  #use "*filename.ml*"

- Example:

  #use loads in a file one line at a time

```
# #use "hello.ml";;
Hello world!
- : unit = ()
#
```

OPAM: OCaml Package Manager

- opam is the package manager for OCaml
  - Manages libraries and different compiler installations
- We recommend installing the following packages with opam
  - OUnit, a testing framework similar to minitest
  - Utop, a top-level interface similar to irb
  - Dune, a build system for larger projects

Ocamlbuild: Smart Project Building

- Use **`ocamlbuild`** to compile larger projects and automatically find dependencies
- Build a bytecode executable out of **`main.ml`** and its local dependencies

```
ocamlbuild main.byte
```

- The executable **`main.byte`** is in **`_build`** folder. To execute:

```
./main.byte
```

Dune: Smarter Project Building

- Use **dune** to compile larger projects and automatically find dependencies
- Define a dune file, similar to a Makefile:

dune:

```
(executable
  (name main))
```

Indicates that an executable (rather than a library) is to be built

Name of main file (entry point)

```
% dune build main.exe
% _build/default/main.exe
30
%
```

Check out https://medium.com/@bobbypriambodo/starting-

Dune commands

- If defined, run a project's test suite:

```
dune runtest
```

- Load the modules defined in src/ into the **`utop`** top-level interface:

```
dune utop src
```

- **`utop`** is a replacement for **`ocaml`** that includes dependent files, so they don't have be be **`#load`**ed

A Note on ;;

- ;; ends an expression in the top-level of OCaml
  - Use it to say: "Give me the value of this expression"
  - Not used in the body of a function
  - Not needed after each function definition
    - Though for now it won't hurt if used there
- There is also a single semi-colon ; in OCaml
  - But we won't need it for now
  - It's only useful when programming imperatively, i.e., with side effects
    - Which we won't do for a while

OCaml Expressions and Functions

Lecture Presentation Style

- Our focus: semantics and idioms for OCaml
  - *Semantics* is what the language does
  - *Idioms* are ways to use the language well

- We will also cover some useful libraries

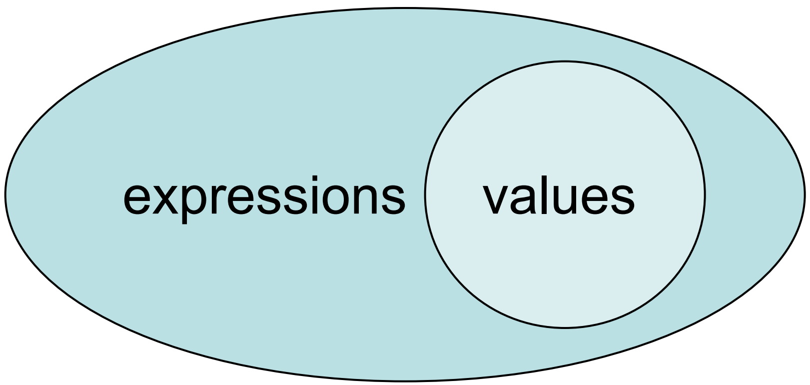
- Syntax is what you type, not what you mean
  - In one lang: Different syntax for similar concepts
  - Across langs: Same syntax for different concepts
  - Syntax can be a source of fierce disagreement among language designers!

Expressions

- Expressions are our primary building block
  - Akin to *statements* in imperative languages
- Every kind of expression has
  - Syntax
    - We use metavariable *e* to designate an arbitrary expression
  - Semantics
    - Type checking rules (static semantics): produce a type or fail with an error message
    - Evaluation rules (dynamic semantics): produce a value
      - (or an exception or infinite loop)
      - Used *only* on expressions that type-check

Values

- A value is an expression that is final
  - *Evaluating* an expression means running it until it becomes a value
  - We use metavariable *v* to designate an arbitrary value
- **`34`** is a value, **`true`** is a value
- **`34+17`** is an *expression*, but *not* a value
  - It *evaluates* to **`51`**



Types

- Types classify expressions
  - The set of values an expression could evaluate to
  - We use metavariable ***t*** to designate an arbitrary type
    - Examples include **`int`**, **`bool`**, **`string`**, and more.
- Expression ***e*** has type ***t*** if ***e*** will (always) evaluate to a value of type ***t***
  - { …, **`-1`**, **`0`**, **`1`**, … } are values of type **`int`**
  - **`34+17`** is an expression of type **`int`**, since it evaluates to **`51`**, which has type **`int`**
  - Write ***e : t*** to say ***e*** has type ***t***
  - Determining that ***e*** has type ***t*** is called type checking (or simply, typing)

If Expressions

```
if e1 then e2 else e3
```

bool t

If Expressions: Examples

```
# if 7 > 42 then "hello" else "goodbye";;
- : string = "goodbye"

# if true then 3 else 4;;
- : int = 3

# if false then 3 else 3.0;;
Error: This expression has type float but
an expression was expected of type int
```

Quiz 1

To what value does this expression evaluate?

```
if 22>0 then 2 else 1
```

A. 2

B. 1

C. 0

D. none of the above

Quiz 1

To what value does this expression evaluate?

```
if 22<0 then 2 else 1
```

A. 2

B. 1

C. 0

D. none of the above

Quiz 2

To what value does this expression evaluate?

```
if 22<0 then "parasite" else 1917
```

A. 2

B. 1

C. 0

D. none of the above

Quiz 2

To what value does this expression evaluate?

```
if 22<0 then "parasite" else 1917
```

A. 2

B. 1

C. 0

D. none of the above: doesn't type check so never gets a chance to be evaluated

Function Definitions

- OCaml functions are like mathematical functions
  - Compute a result from provided arguments

```
(* requires n>=0 *)
(* returns: n! *)
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

function body

Use (* *) for comments (may nest)

Parameter (type inferred)

rec needed for recursion (else **fact** not in scope)

Structural equality

Line breaks, spacing ignored (like C, C++, Java, not like Ruby)

Type Inference

- As we just saw, a declared variable need not be annotated with its type
  - The type can be inferred

```
(* requires n>=0 *)
(* returns: n! *)
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

n's type is **int**. Why?

= is an infix function that takes two **int**s and returns a **bool**; so **n** must be an **int** for **n = 0** to type check

  - Type inference happens *as a part of type checking*
    - Determines a type that satisfies code's constraints

Function Types

- In OCaml, `->` is the function type constructor
  - Type `t1 -> t` is a function with argument or *domain* type `t1` and return or *range* type `t`
  - Type `t1 -> t2 -> t` is a function that takes *two* inputs, of types `t1` and `t2`, and returns a value of type `t`. Etc.
- Examples
  - `let next x = x + 1` `(* type int -> int *)`
  - `let fn x = (int_of_float x) * 3` `(* type float -> int *)`
  - `fact` `(* type int -> int *)`

Type Checking Functions

- Syntax `let rec f x1 … xn = e`
- Type checking
  - Conclude that *f* : *t1* -> … -> *tn* -> *u* if *e* : *u* under the following assumptions:
    - *x1* : *t1*, …, *xn* : *tn* (arguments with their types)
    - *f* : *t1* -> … -> *tn* -> *u* (for recursion)

```
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-
1)
```

Calling Functions

```
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

Example evaluation

- `fact 2`
  - `if 2=0 then 1 else 2*fact(2-1)`
  - `2 * fact 1`
  - `2 * (if 1=0 then 1 else 1*fact(1-1))`
  - `2 * 1 * fact 0`
  - `2 * 1 * (if 0=0 then 1 else 0*fact(0-1))`
  - `2 * 1 * 1`
  - `2`

Type Annotations

- The syntax **(*e* : *t*)** asserts that "*e* has type *t*"
  - This can be added (almost) anywhere you like

```
let (x : int) = 3
let z = (x : int) + 5
```

- Define functions' parameter and return types

```
let fn (x:int):float =
       (float_of_int x) *. 3.14
```

- Checked by compiler: Very useful for debugging

Quiz 3: What is the type of `foo 4 2`

```
let rec foo n m =
  if n >= 9 || n<0 then
    m
  else
    m + 10
```

a) Type Error
b) `int`
c) `float`
d) `int -> int -> int`

Quiz 3: What is the type of `foo 4 2`

```
let rec foo n m =
  if n >= 9 || n<0 then
    m
  else
    m + 10
```

a) Type Error

b) `int`

c) `float`

d) `int -> int -> int`

Quiz 4: What is the value of `bar 4`

```
let rec bar(n:int):int =
  if n = 0 || n = 1 then 1
  else
    bar (n-1) + bar (n-2)
```

a) Syntax Error
b) 4
c) 5
d) 8

Quiz 4: What is the value of `bar 4`

```
let rec bar(n:int):int =
  if n = 0 || n = 1 then 1
  else
    bar (n-1) + bar (n-2)
```

a) Syntax Error

b) 4

c) 5

d) 8