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:

\[
\begin{align*}
x &= x + 1; \\
a[i] &= 42; \\
p.next &= p.next.next;
\end{align*}
\]

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

\[
\begin{align*}
x &= x+1; \\
a[i] &= 42; \\
p.next &= p.next.next;
\end{align*}
\]

Thread 2 on CPU 2

\[
\begin{align*}
x &= x+1; \\
a[i] &= 42; \\
p.next &= p.next.next;
\end{align*}
\]

- 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
Functions/methods have **side effects**:

```c
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.

\[ f(x) + f(x) + f(x) \neq 3 f(x) \]
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?]
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](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:

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

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

---

CMSC 330 - Summer 2020
OCaml Compiler: Multiple Files

**main.ml:**

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

let () = main ()
```

**util.ml:**

```ocaml
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
```
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 "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 build main.exe
% _build/default/main.exe
30
%
```

- Indicates that an executable (rather than a library) is to be built
- Name of main file (entry point)

Check out https://medium.com/@bobbypriambodo/starting-an-ocaml-app-project-using-dune-CMSC-330-Summer-2020
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 be `#loaded`
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 $t$
  – $\{\ldots, -1, 0, 1, \ldots \}$ 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

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]
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?

\[
\text{if } 22 > 0 \text{ then } 2 \text{ else } 1
\]

A. 2  
B. 1  
C. 0  
D. none of the above
Quiz 1

To what value does this expression evaluate?

```python
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?

\[
\text{if } 22 < 0 \text{ then } "parasite" \text{ else } 1917
\]

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)
```

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

\[
\begin{align*}
\text{(* requires } n\geq 0 \*) \\
\text{(* returns: } n! \*) \\
\text{let rec fact } n = \\
\quad \text{if } n = 0 \text{ then } 1 \\
\quad \text{else } n \times \text{fact } (n-1)
\end{align*}
\]

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

n’s type is int. Why?

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

- In OCaml, \( \rightarrow \) is the function type constructor
  - Type \( t_1 \rightarrow t \) is a function with argument or domain type \( t_1 \) and return or range type \( t \)
  - Type \( t_1 \rightarrow t_2 \rightarrow t \) is a function that takes two inputs, of types \( t_1 \) and \( t_2 \), and returns a value of type \( t \). Etc.

- Examples
  - let next x = x + 1 (* type int \rightarrow int *)
  - let fn x = (int_of_float x) * 3 (* type float \rightarrow int *)
  - fact (* type int \rightarrow int *)
Type Checking Functions

• Syntax \( \texttt{let rec } f \ x_1 \ldots \ x_n = e \)

• Type checking
  – Conclude that \( f : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \) if \( e : u \) under the following assumptions:
    • \( x_1 : t_1, \ldots, x_n : t_n \) (arguments with their types)
    • \( f : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \) (for recursion)

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

Calling Functions

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

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

• Define functions’ parameter and return types

  ```ocaml
  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 \texttt{foo 4 2}

```ocaml
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 \texttt{foo 4 2}

\begin{verbatim}
let rec foo n m =
    if n >= 9 || n<0 then
        m
    else
        m + 10
\end{verbatim}

a) Type Error  
b) int  
c) float  
d) int -> int -> int
Quiz 4: What is the value of $\text{bar} \ 4$

let rec $\text{bar}(n:\text{int}):\text{int} =$
  if $n = 0 \text{||} n = 1$ then 1
  else
    $\text{bar}(n-1) + \text{bar}(n-2)$

a) Syntax Error
b) 4
c) 5
d) 8
Quiz 4: What is the value of \texttt{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