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**

**State**: the information maintained by a computation

**Mutable**: can be changed

\[ x = x + 1 \]
Functional vs. Imperative

Functional languages

- *Higher* level of abstraction: *What* to compute, not *how*
- *Immutable* state: easier to reason about (meaning)
- *Easier* to develop robust software

Imperative languages

- *Lower* level of abstraction: *How* to compute, not *what*
- *Mutable* state: harder to reason about (behavior)
- *Harder* to develop robust software
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...

The reality?
- Machines are good at complicated manipulation of state
- Humans are not good at understanding it!
Imperative Programming: Reality

Functions/methods may **mutate** state, a **side effect**

```cpp
int cnt = 0;

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

Mutation **breaks referential transparency**: ability to replace an expression with its value without affecting the result

\[ f(x) + f(x) + f(x) \neq 3 \ast f(x) \]
Imperative Programming: Reality

Worse: 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

So: Can’t look at one piece of code and reason about its behavior

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

\[
\begin{align*}
x &= x+1; \\
a[i] &= 42; \\
p.next &= p.next.next;
\end{align*}
\]
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
- Can perform local reasoning, assume referential transparency

Easier to build **correct** programs
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**
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++

• Exceptions and garbage collection
Why study functional programming?

Functional languages predict the future:

- **Garbage collection**
  - LISP [1958], Java [1995], Python 2 [2000], Go [2007]
- **Parametric polymorphism (generics)**
  - ML [1973], SML [1990], Java 5 [2004], Rust [2010]
- **Higher-order functions**
- **Type inference**
  - ML [1973], C++11 [2011], Java 7 [2011], Rust [2010]
- **Pattern matching**
  - SML [1990], Scala [2002], Rust [2010], Java X [201?]
    - [http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html](http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html)
Why study functional programming?

Functional languages in the real world

- Java 8 (Oracle)
- F#, C# 3.0, LINQ (Microsoft)
- Scala (twitter, foursquare, LinkedIn)
- Haskell (facebook, BARCLAYS, at&t)
- Erlang (facebook, Amazon, T-Mobile)
- OCaml (facebook, Bloomberg, CITRIX, Jane Street)

This slide is old---now there are even more!
Useful Information on OCaml

• 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
OCaml Coding Guidelines

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

• https://ocaml.org/learn/tutorials/guidelines.html
CMSC 330: Organization of Programming Languages

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:

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

```
% ocamlc hello.ml
% ./a.out
Hello world!
%```
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`)
  ```bash
  ocamlc util.ml main.ml
  ```
- Or compile separately
  ```bash
  ocamlc -c util.ml
  ocamlc util.cmo main.ml
  ```
- To execute
  ```bash
  ./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 via the `ocaml` command
  ```
  ocaml
  OCaml version 4.07.0
  # print_string "Hello world!\n";;
  Hello world!
  - : unit = ()
  # exit 0;;
  ```

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

Expressions can be typed and evaluated at the top-level

```ocaml
# 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 Code Files into the Top-level

File `hello.ml`:

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

- Load a file into top-level
  ```ml
  #use "filename.ml"
  ```

- Example:
  ```ml
  # #use "hello.ml";;
  Hello world!
  - : unit = ()
  #
  ```
  #use processes a file a line at a time
OPAM: OCaml Package Manager

- **opam** is the package manager for OCaml
  - Manages libraries and different compiler installations

- You should install 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
Project Builds with **dune**

- Use **dune** to compile projects---automatically finds dependencies, invokes compiler and linker
- Define a **dune** file, similar to a **Makefile**:

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

**dune**:

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

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-d4f74e291de8](https://medium.com/@bobbypriambodo/starting-an-ocaml-app-project-using-dune-d4f74e291de8)
Dune commands

- If defined, run a project’s test suite:
  
  \texttt{dune runtest}

- Load the modules defined in src/ into the \texttt{utop} top-level interface:
  
  \texttt{dune utop src}

- \texttt{utop} is a replacement for \texttt{ocaml} that includes dependent files, so they don’t have to be \texttt{#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
CMSC 330: Organization of Programming Languages

OCaml Expressions, 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
  – 34 is a value, true is a value
  – 34+17 is an expression, but not a value
• Evaluating an expression means running it until it’s a value
  – 34+17 evaluates to 51
• We use metavariable $v$ to designate an arbitrary value
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 $t$
  - 0, 1, and -1 are values of type int while true has type bool
  - $34+17$ 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

• Syntax
  
  \[(\text{if } e_1 \text{ then } e_2 \text{ else } e_3): t\]

  \[
  : \text{bool} \quad : t
  
  \text{(each has the same type } t)\]

• Type checking
  – Conclude if \( e_1 \text{ then } e_2 \text{ else } e_3 \) has type \( t \) if
    • \( e_1 \) has type \( \text{bool} \)
    • Both \( e_2 \) and \( e_3 \) have type \( t \) (for some \( t \))
If Expressions: Type Checking and Evaluation

# 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

- Evaluation (happens if type checking succeeds)
  - If \( e_1 \) evaluates to true, and if \( e_2 \) evaluates to \( v \),
    then if \( e_1 \) then \( e_2 \) else \( e_3 \) evaluates to \( v \)
  - If \( e_1 \) evaluates to false, and if \( e_3 \) evaluates to \( v \),
    then if \( e_1 \) then \( e_2 \) else \( e_3 \) evaluates to \( v \)
Quiz 1

To what value does this expression evaluate?

```
if 10 < 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?

```plaintext
if 10 < 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?

\[
\text{if } 22 < 0 \text{ then } 2021 \text{ else } \text{``home''}
\]

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

To what value does this expression evaluate?

```plaintext
if 22 < 0 then 2021 else "home"
```

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

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

- Use (* *) for comments (may nest)
- Parameter (type inferred)
- \texttt{rec} needed for recursion (else \texttt{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
Calling Functions, *aka* Function Application

- **Syntax** \( f e_1 \ldots e_n \)
  - Parentheses not required around argument(s)
  - No commas; use spaces instead

- **Evaluation**
  - Find the definition of \( f \)
    - i.e., \( \text{let rec } f \ x_1 \ldots \ x_n = e \)
  - Evaluate arguments \( e_1 \ldots e_n \) to values \( v_1 \ldots v_n \)
  - **Substitute** arguments \( v_1, \ldots v_n \) for params \( x_1, \ldots x_n \) in body \( e \)
    - Call the resulting expression \( e' \)
  - Evaluate \( e' \) to value \( v \), which is the final result
Calling Functions: Evaluation

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

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

Fun fact: Evaluation order for function call arguments in OCaml is right to left (not left to right)
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
  
  – \( \text{not} \) \hspace{1cm} (* type bool \rightarrow bool *)
  
  – \( \text{int_of_float} \) \hspace{1cm} (* type float \rightarrow int *)
  
  – \( + \) \hspace{1cm} (* type int \rightarrow int \rightarrow int *)
Type Checking: Calling Functions

• Syntax \( f \, e_1 \ldots \, e_n \)

• Type checking
  – If \( f : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \)
  – and \( e_1 : t_1 \),
  – \( \ldots , e_n : t_n \)
  – then \( f \, e_1 \ldots \, e_n : u \)

• Example:
  – not true : bool
  – since not : bool \rightarrow bool
  – and true : bool
Type Checking: Defining Functions

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

• 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)

\[
\text{let rec fact } n = \\
\quad \text{if } (n = 0) \text{ then } 1 \\
\quad \text{else } (n \times \text{fact}(n-1))
\]

:bool assuming \( n: \text{int} \)

\( \text{fact}(n-1): \text{int} \)

since

\( \text{and } (n-1): \text{int} \)

assuming \n
\( \text{fact}: \text{int} \rightarrow \text{int} \)
Function Type Checking: More Examples

- let next x = x + 1  (* type int -> int *)
- let fn x = (int_of_float x) * 3  (* type float -> int *)
- fact  (* type int -> int *)
- let sum x y = x + y  (* type int -> int -> int *)
Quiz 3: What is the type of `foo 3 1.5`?

```ocaml
let rec foo n m =
  if n >= 9 || n > 0 then
    m
  else
    m +. 10.3
```

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

Type: `float -> float -> float`
Quiz 3: What is the type of `foo 3 1.5`?

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

```ocaml
let rec foo n m =
  if n >= 9 || n > 0 then
    m
  else
    m +. 10.3
```

: float -> float -> float
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 4: What is the value of \texttt{bar 4}

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

a) Syntax Error
b) 4
c) 5
d) 8
Quiz 4: What is the value of $\text{bar} \ 4$

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