CMSC 330: Organization of Programming Languages

Functional Programming with OCaml
What is a functional language?

A functional language:

• defines computations as **mathematical functions**
• avoids mutable **state**

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

**Mutable:** can be changed
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
Imperative Programming

Commands specify how to compute by destructively changing state:

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

Functions/methods have side effects:

```c
int wheels(Vehicle v) {
    v.size++;
    return v.numWheels;
}
```
Mutability

The fantasy of mutability:
• It's easy to reason about: the machine does this, then this...

The reality of mutability:
• Machines are good at complicated manipulation of state
• Humans are not good at understanding it!
  • mutability breaks referential transparency: ability to replace an expression with its value without affecting the result

• In math, if $f(x)=y$, then you can substitute $y$ anywhere you see $f(x)$

• In imperative languages, you cannot: $f$ might have side effects, so computing $f(x)$ at one time might result in different value at another
Mutability

The **fantasy** of mutability:
- There is a single state
- The computer does one thing at a time

The **reality** of mutability:
- 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
- There is no single program
  - Most applications do many things at one time
Functional programming

Expressions specify what to compute
• Variables never change value
• 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
Why study functional programming?

Functional languages predict the future:

- Garbage collection
  - Java [1995], LISP [1958]
- 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]
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 on-line
  - realworldocaml.org
Features of ML

• First-class functions
  – Functions can be data, too: parameters and return values

• 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

• Garbage collection