History of Programming Languages

Functional Programming

Cunning Plan

- Review and Administrivia
  - Office Hours
- History Lesson
  - Babbage to C#
- Functional Programming
  - OCaml
  - Types
  - Pattern Matching
  - Higher-Order Functions

Gone In Sixty Seconds

- Imperative: change state, assignments
- Structured: if/block/routine control flow
- Object-Oriented: message passing, inheritance
- Functional: functions are first-class citizens that can be passed around or called recursively. We can avoid changing state by passing copies.
Discussion Sections

- Structured Office Hours
  - Wednesdays 4pm - 5pm in MEC 341
  - Mondays 10am - 11am in OLS 011
- Pieter Office Hours
  - Thursdays 3:30pm - 4:30pm in OLS 235
- Wes Office Hour
  - Wednesday 2pm - 3pm in OLS 219

Why Study History?

- Those who cannot remember George Santayana are condemned to misquote him.
  - *Supernatural*, 1999

Why Study History?

- Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to repeat it.
- Through meticulous analysis of history I will find a way to make the people worship me. By studying the conquerors of days gone by, I’ll discover the mistakes that made them go awry.
  - The Brain, *A Meticulous Analysis of History*, P031
Theory and Math

- **1822** - Babbage: Difference Engine
  - PL = change the gears

- **1936** - Church & Kleene: Lambda Calculus
  - PL = function is primary unit of computation
  - Legacy = all of functional programming

- **1942** - ENIAC (first large electronic programmable)
  - PL = preset switches, rewire system

- **1945** - John Von Neumann
  - PL = subroutines you can jump to in any order
  - Idea = branch based program IF/THEN, FOR
  - Also = “libraries” = block of reused code

---

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements; they tend to sound pretty silly in five years.

- John Von Neumann, 1949

---

Firsts

- **1949** - Short Code (first PL for electronic devices)
  - PL = you changes stmts to 0’s and 1’s by hand

- **1952** - Grace Hopper: A-0 compiler
  - PL = 3 address assembly code for math probs

- **1954** - Backus: FORTRAN
  - PL = declare variables, types (bool, int, real, array), assignment statements, if-then-else, if-based error checking, goto, computed goto, for-loops, formatted I/O, optimization hints to compiler, no procedures until 1958!

---

Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic.

- Grace Hopper, 1952
Branching Out

- **1958 – John McCarthy: LISP**
  PL = basic datatype is the List, programs themselves are lists, can self-modify, dynamic allocation, garbage collection (!)

- **1959 – Grace Hopper: COBOL**
  PL = designed for businesses, types (strings, text, arrays, records), PICTURE clause (field specification). No local vars, recursion, dynamic allocation or structured programming.

Structure

- **1960 – ALGOL (de facto standard for 30 years)**
  - Formal grammar in Backus-Naur Form
  - PL = bracketed begin/end, parameter passing, recursive function calls
  - Legacy = Pascal, C, C++, Java, C#, ...

- **1970 – Niklaus Wirth: Pascal**
  - Takes best of Cobol, Fortran and Algol
  - PL = pointers, switch/case, dynamic allocation (new/dispose), enum, no dynamic arrays
  - Easy Adoption = PCODE stack virtual machine

Paradigms

- **1969-73 – Dennis Ritchie: C**
  - Systems programming, minimalist, low-level memory access, "portable", preprocessor

- **1972 – Alan Kay: Smalltalk**
  - PL = object-oriented, dynamic reflective, message passing

- **1973 – Robin Milner: ML**
  - PL = functional, strong static typing, type inference, algebraic datatypes, pattern matching, exception handling

---

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.
- Edsger Dijkstra, 1975

(ALGOL 60) is a language that is so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all of its successors.
- C. A. R. Hoare, 1973

C is often described, with a mixture of fondness and disdain varying according to the speaker, as 'a language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language.'
- MIT Jargon Dictionary
Modern Era

- 1983 - Ada  
  US DOD, static type safe

- 1983 - C++  
  classes, default args, STL

- 1987 - Perl  
  dynamic scripting lang

- 1990 - Python  
  interp OO, +readability

- 1991 - Java  
  portable OO lang (for iTV)

- 1993 - Ruby  
  Perl + Smalltalk

- 1996 - OCaml  
  ML + C++

- 2000 - C#  
  “simple” Java + delegates

I invented the term Object-Oriented, and I did not have C++ in mind.  
- Alan Kay

Oh what a tangled web we weave,  
When first we practise to deceive!  
- Sir Walter Scott, 1771-1832

There are only two kinds of programming languages: those people always [complain] about and those nobody uses.  
- Bjarne Stroustrup

Computer language design is just like a stroll in the park. Jurassic Park, that is.  
- Larry Wall

I fear the new OO systems may suffer the fate of LISP, in that they can do many things, but the complexity of the class hierarchies may cause them to collapse under their own weight.  
- Bill Joy

Let’s Get A Feel For It

- We’ll now see the same program in many of these languages
- We’ll watch it evolve over time
- The program reads lines of integers from standard input and prints the sum
- Think about how you would do this ...
FORTRAN -- 1954

```
program sum
  implicit none
  integer :: datum, s
  s = 0
  do
    read(5,*),end=10) datum
    s = s + datum
  end do
10 continue
  write(*, '(i10)') s
end program sum
```

Smalltalk -- 1972

```
| sum inStream |
sum := 0.
[inStream atEnd] whileFalse:
  [sum := sum + inStream nextLine asInteger].
Transcript show: sum displayString; nl !
```

ML -- 1973

```
val sum = lines sum =
  case TextIO.inputLine TextIO.stdin of
    MORE => print (concat Int.toString sum, "\n")
    DONE => sum: (sum + (Option.value Int.fromString str))
  end
val _ = sum lines 0
```
program sumcol;                PASCAL -- 1970
{mode objfpc}
var num, tot: longint;
begin
  while not Eof(input) do begin
    ReadLn(input, num);
    tot := tot + num;
  end;
  WriteLn(tot);
end.

C -- 1972

#include <stdio.h>
#include <stdlib.h>
#define MAXLINELEN 128
int main() {
  int sum = 0;
  char line[MAXLINELEN];
  while (fgets(line, MAXLINELEN, stdin)) {
    sum += atol(line);
    printf("%d", sum);
    return 0;
}

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <stdio.h>

using namespace std;
#define MAXLINELEN 128
int main(int argc, char * argv) {
  ios_base::sync_with_stdio(false);
  char line[MAXLINELEN];
  int sum = 0;
  char buff[4096];
  cin.tie(NULL) ->pubsetbuf(buff, 4096); // enable buffer
  while (cin.getline(line, MAXLINELEN)) {
    sum += atol(line);
  }
  cout << sum << '\'n';
Java -- 1991

```java
public class sumcol {
    public static void main(String[] args) {  
        int sum = 0;
        String line;
        try {  
            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
            while ((line = in.readLine()) != null) {
                sum = sum + Integer.parseInt(line);
            }
        } catch (IOException e) {  
            System.err.println(e);  
            return;
        }
        System.out.println(Integer.toString(sum));
    }
}
```

Ruby -- 1995

```
#!/usr/bin/ruby
#
# -*- mode: ruby -*-
# $Id: sumcol-ruby.code,v 1.12 2006/09/20 05:52
# http://www.bagley.org/~doug/shootout/
# from: Mathieu Bouchard, revised by Dave Ander

count = 0

STDIN.each{ |l|  
    count += l.to_i
}

puts count
```

Q: Advertising (785 / 842)

- Identify the company associated with two of the following four advertising slogans or symbols.
  - "Fill it to the rim."
  - "I bet you can't eat just one."
  - "Snap, Crackle, Pop"
  - "The San Francisco Treat"
Functional Programming

- You know OO and Structured Imperative
- **Functional Programming**
  - Computation = evaluating (math) functions
  - Avoid “global state” and “mutable data”
  - Get stuff done = apply (higher-order) functions
  - Avoid sequential commands
- Important Features
  - Higher-order, first-class functions
  - Closures and recursion
  - Lists and list processing

State

- The **state** of a program is all of the current variable and heap values
- **Imperative** programs destructively modify existing state
  
  \[ \text{add}_{\text{elem}}(\text{SET}, \text{item}) \]

- **Functional** programs yield new similar states over time
  
  \[ \text{add}_{\text{elem}}(\text{SET}_1, \text{item}) \rightarrow \text{SET}_2 \]

List Syntax in OCaml

- Empty List \[ [ ] \]
- Singleton \[ [ \text{element} ] \]
- Longer List \[ [ \text{e}_1; \text{e}_2; \text{e}_3 ] \]
- Cons \[ \text{x} :: [\text{y};\text{z}] = [\text{x};\text{y};\text{z}] \]
- Append \[ [\text{w};\text{x}];[\text{y};\text{z}] = [\text{w};\text{x};\text{y};\text{z}] \]
- List.length, List.filter, List.fold, List.map ...
- More on these later!
- Every element in list must have same type
Functional Example

- Simple Functional Set (built out of lists)
  - let rec add_elem (s, e) =
    - if s = [] then [e]
    - else if List.hd s = e then s
    - else List.hd s :: add_elem(List.tl s, e)

- Pattern-Matching Functional
  - let rec add_elem (s, e) = match s with
    - | [] -> [e]
    - | hd :: tl when e = hd -> s
    - | hd :: tl -> hd :: add_elem(tl, e)

Imperative Code

- More cases to handle
  - List* add_elem(List *s, item e) {
    - if (s == NULL)
      - return list(e, NULL);
    - else if (s->hd == e)
      - return s;
    - else if (s->tl == NULL) {
      - s->tl = list(e, NULL); return s;
    } else
      - return add_elem(s->tl, e);  

Functional-Style Advantages

- Tractable program semantics
  - Procedures are functions
  - Formulate and prove assertions about code
  - More readable

- Referential transparency
  - Replace any expression by its value without changing the result

- No side-effects
  - Fewer errors
Functional-Style Disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation
- Unfamiliar (to you!)
  - New programming style
- Not appropriate for every program
  - Operating systems, etc.

<table>
<thead>
<tr>
<th>Language</th>
<th>Speed</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc)</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>C++ (g++)</td>
<td>1.0</td>
<td>1.4</td>
</tr>
<tr>
<td>OCaml</td>
<td>1.5</td>
<td>2.8</td>
</tr>
<tr>
<td>Java (JDK server)</td>
<td>5.7</td>
<td>9.1</td>
</tr>
<tr>
<td>Lisp</td>
<td>1.7</td>
<td>11</td>
</tr>
<tr>
<td>C# (mono)</td>
<td>2.4</td>
<td>5.4</td>
</tr>
<tr>
<td>Python</td>
<td>6.5</td>
<td>3.9</td>
</tr>
<tr>
<td>Ruby</td>
<td>16</td>
<td>5.0</td>
</tr>
</tbody>
</table>

There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.

- Robin Milner, 1997

ML Innovative Features

- Type system
  - Strongly typed
  - Type inference
  - Abstraction
- Modules
- Patterns
- Polymorphism
- Higher-order functions
- Concise formal semantics

Type System

- Type Inference
  - `let rec add_elem (s,e) = match s with`  
  - `| [] -> [e]`  
  - `| hd :: tl when e = hd -> s`  
  - `| hd :: tl -> hd :: add_elem(tl, e)`  
  - `val add_elem : α list * α * α list = <fun>`
- ML infers types
  - Inconsistent or incomplete type is an error
- Optional type declarations `exp : type`
  - Clarify ambiguous cases
  - Documentation
Pattern Matching

- Simplifies Code (eliminates ifs, accessors)
  - type btree = (* binary tree of strings *)
  - | Node of btree * string * btree
  - | Leaf of string
  - let rec height tree = match tree with
    - | Leaf _ -> 1
    - | Node(x,_,y) -> 1 + max (height x) (height y)
  - let rec mem tree elt = match tree with
    - | Leaf str | Node(_,str,_) -> str = elt
    - | Node(x,_,y) -> mem x elt || mem y elt

Pattern Matching Mistakes

- What if I forget a case?
  - let rec is_odd x = match x with
    - | 0 -> false
    - | 2 -> false
    - | x when x > 2 -> is_odd (x-2)
  - Warning P: this pattern-matching is not exhaustive.
  - Here is an example of a value that is not matched: 1

Polymorphism

- Functions and type inference are polymorphic
  - Operate on more than one type
  - let rec length x = match x with
    - | [] -> 0
    - | hd :: tl -> 1 + length tl
  - val length : α list -> int = <fun>
  - length [1;2;3] = 3
  - length ["algol"; "smalltalk"; "ml"] = 3
  - length [1 ; "algol"] = ?
Higher-Order Functions

- Function are first-class values
  - Can be used whenever a value is expected
  - Notably, can be passed around
  - Closure captures the environment
- let rec map f lst = match lst with
  - | [] -> []
  - | hd :: tl -> f hd :: map f tl
- val map : (α -> β) -> α list -> β list = <fun>
- let offset = 10 in
- let myfun x = x + offset in
- val myfun : int -> int = <fun>
- map myfun [1;8;22] = [11;18;32]

- Extremely powerful programming technique
  - General iterators
  - Implement abstraction

The Story of Fold

- We’ve seen length and map
- We can also imagine ...
  - sum [1; 5; 8] = 14
  - product [1; 5; 8] = 40
  - and [true; true; false] = false
  - or [true; true; false] = true
  - filter (fun x -> x>4) [1; 5; 8] = [5; 8]
  - reverse [1; 5; 8] = [8; 5; 1]
  - mem 5 [1; 5; 8] = true
- Can we build all of these?

The House That Fold Built

- The fold operator comes from Recursion Theory (Kleene, 1952)
  - let rec fold f acc lst = match lst with
    - | [] -> acc
    - | hd :: tl -> fold f (f acc hd) tl
  - val fold : (α -> β -> α) -> α list -> β list -> α = <fun>
- Imagine we’re summing a list:
It’s Lego Time

- Let’s build things out of Fold
  - length lst = fold (fun acc elt -> acc + 1) 0 lst
  - sum lst = fold (fun acc elt -> acc + elt) 0 lst
  - product lst = fold (fun acc elt -> acc * elt) 1 lst
  - and lst = fold (fun acc elt -> acc && elt) true lst
- How would we do or?
- How would we do reverse?

Tougher Legos

- Examples:
  - reverse lst = fold (fun e acc -> acc @ [e]) [] lst
  - Note typing: (acc : α list) (e : α)
  - filter keep_it lst = fold (fun elt acc ->
    - if keep_it elt then elt :: acc else acc) [] lst
  - mem wanted lst = fold (fun elt acc ->
    - acc || wanted = elt false lst
  - Note typing: (acc : bool) (e : α)
- How do we do map?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]
  - Let’s write it on the board ...

Map From Fold

- let map myfun lst =
- fold (fun elt acc -> (myfun elt) :: acc) [] lst
  - Types: (myfun : α -> β)
  - Types: (lst : α list)
  - Types: (acc : β list)
  - Types: (elt : α)
- How do we do sort?
  - (sort : (α * α -> bool) -> α list -> α list)
Sorting Examples

- langs = ["fortran"; "algol"; "c"]
- courses = [216; 333; 415]
- sort (fun a b -> a < b) langs
  - ["algol"; "c"; "fortran"]
- sort (fun a b -> a > b) langs
  - ["fortran"; "c"; "algol"]
- sort (fun a b -> strlen a < strlen b) langs
  - ["c"; "algol"; "fortran"]
- sort (fun a b -> match is_odd a, is_odd b with
  - | true, false -> true (* odd numbers first *)
  - | false, true -> false (* even numbers last *)
  - | _, _ -> a < b (* otherwise ascending *)
) courses
  - [333; 415; 216]

Partial Application and Currying

- let myadd x y = x + y
- val myadd : int -> int -> int = <fun>
- myadd 3 5 = 8
- let addtwo = myadd 2
  - How do we know what this means? We use referential transparency! Basically, just substitute it in.
- val addtwo : int -> int = <fun>
- addtwo 77 = 79
- Currying: “if you fix some arguments, you get a function of the remaining arguments”

Applicability

- ML, Python and Ruby all support functional programming
  - closures, anonymous functions, etc.
- ML has strong static typing and type inference (as in this lecture)
- Ruby and Python have “strong” dynamic typing (or duck typing)
- All three combine OO and Functional
  - ... although it is rare to use both.
Homework

- Thursday: Cool Reference Manual
- Thursday: Backus Speedcoding
- Friday: PA0 due