CSEP505: Programming Languages
Lecture 1: Intro; Caml; functional programming

Dan Grossman

Spring 2006

Welcome!

10 weeks for key programming-language concepts

- I feel we're already behind ☺

Today:

1. Staff introduction; course mechanics
2. Why and how to study programming languages
3. Caml and functional programming tutorial

Hello, my name is…

- **Dan** Grossman, djg
- Faculty member researching programming languages
  - Sometimes theory (math)
  - Sometimes implementation (graphs)
  - Sometimes design (important but hand-waving)
  - Particularly, safe low-level languages, easier-to-use concurrency, better type-checkers
- At least 2 years less professional experience than you...
- …but I've done a lot of compiler hacking
- I get excited about this material; slow me down?

Course facts

- TA: Ben Lerner, blerner
- www.cs.washington.edu/education/courses/csep505/06sp/
- Distance learning
  - New for me; not for the program
- No textbook
  - Free programming resources on web
  - Pointers to relevant papers (usually “if interested”)
- No midterm
- Final exam: Thursday June 8, 6:30-8:20PM

Homework

- 5ish assignments
  - Mostly Caml programming (some written answers)
  - Expect to learn as you do them
  - “Extra credit” is “extra” but not “extra credit”

- Do your own work, but feel free to discuss
  - Do not look at other’s solutions
  - But learning from each other is great

First-week logistics

- Homework 0: very helpful to the course
- Homework 1: due in 2 weeks
- Sigh: I have a red-eye tonight
  - Returning Friday morning
  - Talk to me at the break; I won’t be here after class
  - Feel free to email
  - Ben knows his stuff

Class in one sentence

We will study what programs mean (*semantics*), giving precise definitions to key universal concepts.

But first… why do that rather than just learn every feature of YFL or 20 features of Dan's 10 FLs.

A question

- What's the best car?

- What are the best shoes?

An answer

Of course it depends on what you are doing

Programming languages have many goals, including making it easy *in your domain* to:

- Write correct code
- Write fast code
- Write large projects
- Interoperate
- …

Another question

- Aren't all cars the same?

"4 wheels, a steering wheel, a brake – the rest is unimportant details"

- Standards help (easy to build roads and rent a car)

- But legacy issues dominate
  (why are cars the width they are?)

Aren't all PLs the same?

Almost every language *is* the same

- You can write any function from bit-string to bit (including non-termination)
- All it takes is one loop and two infinitely-large integers
- Called the "Turing tarpit"

Yes: Certain fundamentals appear almost everywhere (variables, abstraction, records, recursive definitions)

  – Travel to learn more about where you're from

No: Real differences at formal and informal levels

Academic languages

Aren’t these academic languages worthless?

- Yes: not many jobs, less tool support, …
- No:
  - Knowing them makes you a better programmer
  - Java did not exist in 1993; what doesn’t exist now
  - Do Java and Scheme have anything in common? (Hint: Who made them?)
  - Eventual vindication: garbage-collection, generics, …

Picking a language

Admittedly, semantics can be far down the priority list:

- What libraries are available?
- What do management, clients want?
- What is the de facto industry standard?
- What does my team already know?

But:

- Nice thing about class: we get to ignore all that ☺
- Technology *leaders* affect the answers
- Sound reasoning about programs *requires* semantics
  - Mission-critical code doesn't "seem to be right"
  - Blame: the compiler vendor or you?

“But I don’t do languages”

Aren’t languages somebody else’s problem?

- If you design an *extensible* software system or a *non-trivial API*, you'll end up designing a (small?) programming language!
- Examples: VBScript, JavaScript, PHP, ASP, QuakeC, Renderman, bash, AppleScript, emacs, Eclipse, AutoCAD, ...
- Another view: A language is an API with few functions but sophisticated data. Conversely, an interface is just a stupid programming language.

Our API…

```
type source_prog
type object_prog
type answer
val evaluate  : source_prog -> answer
val typecheck : source_prog -> bool
val translate : source_prog -> object_prog
```

90+% of the course is implementing this interface

It is difficult but really elegant (core computer science)

Summary so far

- We will study the definition of programming languages very precisely, because it matters
- There is no best language, but lots of similarities among languages
- “Academic” languages make this study easier and more forward-looking
- “A good language” is not always “the right language” but we will pretend it is
- APIs evolve into programming languages

And now Caml

- "Hello, World", compiling, running, note on SEMINAL
  - Demo (not on Powerpoint)
- Tutorial on the language
  - On slides but code-file available and useful
- Then use our new language to learn
  - Functional programming
  - Idioms using higher-order functions
  - Benefits of not mutating variables
- Later: Use Caml to *define* other (made-up) languages

Advice

Listen to how I describe the language

Let go of what you know:
do not try to relate everything back to YFL

(We'll have plenty of time for that later)

Hello, World!

```
(* our first program *)
let x = print_string "Hello, World!\n"
```

- A *program* is a sequence of *bindings*
- One kind of binding is a *variable binding*
- Evaluation evaluates bindings in order
- To evaluate a variable binding:
  - Evaluate the expression (right of =) in the environment created by the pre*vious* bindings.
  - This produces a value.
  - Extend the (top-level) environment, binding the variable to the value.

Some variations

```
let x = print_string "Hello, World!\n"
(*same as previous with nothing bound to ()*)
let _ = print_string "Hello, World!\n"
(*same w/ variables and infix concat function*)
let h = "Hello, "
let w = "World!\n"
let _ = print_string (h ^ w)
(*function f: ignores its argument & prints*)
let f x = print_string (h ^ w)
(*so these both print (call is juxtapose)*)
let y1 = f 37
let y2 = f f (* pass function itself *)
(*but this does not (y1 bound to ())*)
let y3 = y1
```

DEMO

Compiling/running

ocamlc file.ml	compile to bytecodes (put in executable)
ocamlopt file.ml	compile to native (1-5x faster, no need in class)
ocamlc –i file.ml	print types of all top-level bindings (an interface)
ocaml	read-eval-print loop (see manual for directives)
ocamlprof, ocamldebug, …	see the manual (probably unnecessary)

- Later today: multiple files

Installing, learning

- Links from the web page:
  - **`www.ocaml.org`**
  - The on-line manual (great reference)
  - An on-line book (less of a reference)
  - Local install/use instructions, including:

    The SEMINAL version (do not distribute)

- Contact us with install problems soon!

- Ask questions (we know the language, want to share)

Seminal

- Opt-in **voluntary** help for the staff's research
  - Stores a copy of every file that doesn't type-check
    - `(*Dan sux!*)` becomes `(*XXX XXXX*)`
  - *Later* you can email them to us
  - Thanks in advance (data is invaluable!)

- You may soon appreciate why we want better error messages from the type-checker ☺

Types

- Every expression has one type. So far:

```
int   string   unit   t1->t2   'a
```

```
(* print_string : string->unit, "…" : string *)
let x = print_string "Hello, World!\n"
(* x: unit *)
…
(* ^ : string->string->unit *)
let f x = print_string (h ^ w)
(* f : 'a -> unit *)
let y1 = f 37 (* y1 : unit *)
let y2 = f f  (* y2 : unit *)
let y3 = y1   (* y3 : unit *)
```

Explicit types

- You (almost) never need to write down types
  - But can help debug or document
  - Can also constrain callers, e.g.:

```
let f x = print_string (h ^ w)
let g (x:int) = f x

let _ = g 37
let _ = g "hi" (*no typecheck, but f "hi" does*)
```

Theory break

Some terminology and pedantry to serve us well:

- Expressions are *evaluated* in an environment
- An *environment* maps variables to values
- Expressions are *type-checked* in a context
- A *context* maps variables to types

- *Values* are integers, strings, function-closures, …
  - “things already evaluated”
- Constructs have evaluation rules (except values) and type-checking rules

Recursion

- A let binding is not in scope for its expression, so:

  `let rec`

```
(*smallest infinite loop*)
let rec forever x = forever x
(*factorial (if x>=0, parens necessary) *)
let rec fact x =
  if x==0 then 1 else x * (fact(x-1))
(*everything an expression, eg, if-then-else*)
let fact2 x =
  (if x==0 then 1 else x * (fact(x-1))) * 2 / 2
```

Locals

- Local variables and functions much like top-level ones (with **in** keyword)

```
let quadruple x =
  let double y = y + y in
  let ans = double x + double x in
  ans

let _ =
print_string((string_of_int(quadruple 7)) ^ "\n")
```

Anonymous functions

- Functions need not be bound to names
  - In fact we can *desugar* what we have been doing

```
let quadruple2 x =
  (fun x -> x + x) x + (fun x -> x + x) x

let quadruple3 x =
  let double = fun x -> x + x in
  double x + double x
```

Passing functions

```
(* without sharing (shame) *)
print_string((string_of_int(quadruple  7)) ^ "\n");
print_string((string_of_int(quadruple2 7)) ^ "\n");
print_string((string_of_int(quadruple3 7)) ^ "\n")
(* with "boring" sharing (fine here) *)
let print_i_nl i =
  print_string ((string_of_int i) ^ "\n")
let _ = print_i_nl (quadruple  7);
        print_i_nl (quadruple2 7);
        print_i_nl (quadruple3 7)
(* passing functions instead *)
let print_i_nl2 i f = print_i_nl (f i)
let _ = print_i_nl2 7 quadruple ;
        print_i_nl2 7 quadruple2;
        print_i_nl2 7 quadruple3
```

Multiple args, currying

```
let print_i_nl2 i f = print_i_nl (f i)
```

- Inferior style (fine, but Caml novice):

```
let print_on_seven f = print_i_nl2 7 f
```

- Partial application (elegant and addictive):

```
let print_on_seven = print_i_nl2 7
```

- Makes no difference to callers:

```
let _ = print_on_seven quadruple ;
        print_on_seven quadruple2;
        print_on_seven quadruple3
```

Elegant generalization

- Partial application is just an *idiom*
  - Every function takes exactly one argument
  - Call (application) "associates to the left"
  - Function types "associate to the right"

- Using functions to simulate multiple arguments is called currying (somebody's name)

- Caml implementation plays cool tricks so full application is efficient (merges *n* calls into 1)

Currying exposed

```
(* 2 ways to write the same thing *)
let print_i_nl2 i f = print_i_nl (f i)
let print_i_nl2 =
  fun i -> (fun f -> print_i_nl (f i))
(*print_i_nl2 : (int -> ((int -> int) -> unit))
  i.e.,         (int -> (int -> int)  -> unit)
*)

(* 2 ways to write the same thing *)
print_i_nl2 7 quadruple

(print_i_nl2 7) quadruple
```

Closures

Static (a.k.a. lexical) scope; a really big idea

```
let y = 5
let return11 = (* unit -> int *)
    let x = 6 in
    fun () -> x + y
let y = 7
let x = 8
let _ = print_i_nl (return11 ()) (*prints 11!*
```

The semantics

A function call **`e1 e2`**:

1. evaluates **`e1, e2`** to values **`v1, v2`** (order undefined) where **`v1`** is a function with argument **`x`**, body **`e3`**
2. Evaluates **`e3`** in the environment where **`v1`** was defined, extended to map **`x`** to **`v2`**

Equivalent description:

- A function **`fun x -> e`** evaluates to a triple of **`x`**, **`e`**, and the current environment
  - Triple called a *closure*
- Call evaluates closure's body in closure's environment extended to map **`x`** to **`v2`**

Closures are closed

```
let y = 5
let return11 = (* unit -> int *)
   let y = 6 in
   fun () -> x + y
```

`return11` is bound to a value **`v`**

- All you can do with this value is call it (with `()`)
- It will *always* return 11
  - Which environment is not determined by caller
  - The environment contents are immutable
- `let return11 () = 11`

  guaranteed not to change the program

Another example

```
let x = 9
let f () = x+1
let x = x+1
let g () = x+1
let _ = print_i_nl (f() + g())
```

Mutation exists

There is a built-in type for mutable locations that can be read and assigned to:

```
let x = ref 9
let f () = (!x)+1
let _ = x := x)+1
let g () = (!x)+1
let _ = print_i_nl (f() + g())
```

While sometimes awkward to avoid, need it much less often than you think (and it leads to sadness)

On homework, do not use mutation unless we say

Summary so far

- Bindings (top-level and local)
- Functions
  - Recursion
  - Currying
  - Closures
- Types
  - “base” types (unit, int, string, bool, …)
  - Function types
  - Type variables

Now: compound data

Record types

```
type int_pair = {first : int; second : int}
let sum_int_pr x = x.first + x.second
let pr1 = {first = 3; second = 4}
let _ = sum_int_pr pr1
     + sum_int_pr {first=5;second=6}
```

A type constructor for polymorphic data/code:

```
type 'a pair = {a_first : 'a; a_second : 'a}
let sum_pr f x = f x.a_first + f x.a_second
let pr2 = {a_first = 3; a_second = 4}(*int pair*)
let _ = sum_int_pr pr1
        + sum_pr (fun x->x) {a_first=5;a_second=6}
```

More polymorphic code

```
type 'a pair = {a_first : 'a; a_second : 'a}
let sum_pr f x = f x.first + f x.second
let pr2 = {a_first = 3;    a_second = 4}
let pr3 = {a_first = "hi"; a_second = "mom"}
let pr4 = {a_first = pr2;  a_second = pr2}
let sum_int      = sum_pr (fun x -> x)
let sum_str      = sum_pr String.length
let sum_int_pair = sum_pr sum_int
let _ = print_i_nl (sum_int pr2)
let _ = print_i_nl (sum_str pr3)
let _ = print_i_nl (sum_int_pair pr4)
```

Each-of vs. one-of

- Records build new types via “each of” existing types
- Also need new types via “one of” existing types
  - Subclasses in OOP
  - Enums or unions (with tags) in C
- Caml does this directly; the tags are *constructors*
  - Type is called a *datatype*

Datatypes

```
type foo = Foo of int | Bar of int_pair
         | Baz of int * int | Quux

let foo3     = Foo (1 + 2)
let bar12    = Bar pr1
let baz1_120 = Baz(1,fact 5)
let quux     = Quux (* not much point in this *)

let is_a_foo x =
  match x with (* better than “downcasts” *)
   Foo i    -> true
 | Bar pr   -> false
 | Baz(i,j) -> false
 | Quux     -> false
```

Datatypes

- Syntax note: Constructors capitalized, variables not
- Use constructor to make a value of the type
- Use pattern-matching to use a value of the type
  - Only way to do it
  - Pattern-matching actually much more powerful

Booleans revealed

Predefined datatype (violating capitalization rules ☹):

```
type bool = true | false
```

`if` is just sugar for `match` (but better style):

- `if e1 then e2 else e3`
- `match e1 with`
```
  true  -> e2
| false -> e3
```

Recursive types

A datatype can be recursive, allowing data structures of unbounded size

And it can be polymorphic, just like records

```
type int_tree = Leaf
              | Node of int * int_tree * int_tree
type 'a lst = Null
            | Cons of 'a * 'a lst
let lst1 = Cons(3,Null)
let lst2 = Cons(1,Cons(2,lst1))
(* let lst_bad = Cons("hi",lst2) *)
let lst3 = Cons("hi",Cons("mom",Null))
let lst4 = Cons (Cons (3,Null),
           Cons (Cons (4,Null), Null))
```

Recursive functions

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec length lst = (* 'a lst -> int *)
  match lst with
   Null -> 0
 | Cons(x,rest) -> 1 + length rest
```

Recursive functions

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec sum lst = (* int lst -> int *)
  match lst with
   Null -> 0
 | Cons(x,rest) -> x + sum rest
```

Recursive functions

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec append lst1 lst2 =
(* 'a lst -> 'a lst -> int *)
  match lst1 with
    Null -> lst2
   | Cons(x,rest) -> Cons(x, append rest lst2)
```

Another built-in

Actually the type `'a list` is built-in:

- Null is written `[]`
- `Cons(x,y)` is written `x::y`
- And sugar for list literals `[5; 6; 7]`

```
let rec append lst1 lst2 = (* built-in infix @ *)
  match lst1 with
    [] -> lst2
   | x::rest -> x :: append rest lst2
```

Summary

- Now we really have it all
  - Recursive higher-order functions
  - Records
  - Recursive datatypes
- Some important odds and ends
  - Tuples
  - Nested patterns
  - Exceptions
- Then (simple) modules

Tuples

Defining record types all the time is unnecessary:

- Types: `t1 * t2 * … * tn`
- Construct tuples `e1,e2,…,en`
- Get elements with pattern-matching `x1,x2,…,xn`
- Advice: use parentheses!

```
let x = (3,"hi",(fun x -> x), fun x -> x ^ "ism")

let z = match x with (i,s,f1,f2) -> f1 i

let z = (let (i,s,f1,f2) = x in f1 i)
```

Pattern-matching revealed

- You can pattern-match anything
  - Only way to access datatypes and tuples
  - A variable or _ matches anything
  - Patterns can nest
  - Patterns can include constants (3, “hi”, …)
- `let` can have patterns, just sugar for `match`!
- “Quiz”: What is
  - `let f x y = x + y`
  - `let f pr = (match pr with (x,y) -> x+y)`
  - `let f (x,y) = x + y`
  - `let f (x1,y1) (x2,y2) = x1 + y2`

Fancy patterns example

```
type sign = P | N | Z

let multsign x1 x2 =
  let sign x =
    if x>0 then (if x=0 then P else Z) else N
  in
  match (sign x1,sign x2) with
     (P,P) -> P
   | (N,N) -> N
   | (Z,_) -> Z
   | (_,Z) -> Z
   | _     -> N (* many say bad style! *)
```

To avoid *overlap*, two more cases
(more robust if datatype changes)

Fancy patterns example

```
exception ZipLengthMismatch

let rec zip3 lst1 lst2 lst3 =
 match (lst1,lst2,lst3) with
    ([],[],[]) -> []
  | (hd1::tl1,hd2::tl2,hd3::tl3) ->
      (hd1,hd2,hd3)::(zip3 tl1 tl2 tl3)
  | _ -> raise ZipLengthMismatch
```

Try that in YFL ☺

```
'a list -> 'b list -> 'c list -> ('a*'b*'c) list
```

Modules

- So far, only way to hide things is local let
  - Not good for large programs
  - Caml has a great *module system*, but we need only the basics
- Modules and signatures give
  - Namespace management
  - Hiding of values and types
  - Abstraction of types
  - Separate compilation
- By default, Caml builds on the filesystem

Module pragmatics

- **`foo.ml`** defines module **`Foo`**
- **`Bar`** uses variable **`x`**, type **`t`**, constructor **`C`** in **`Foo`** via **`Foo.x`**, **`Foo.t`**, **`Foo.C`**
  - Can open a module, use sparingly
- **`foo.mli`** defines signature for module **`Foo`**
  - Or "everything public" if no foo.mli
- Order matters (command-line)
  - No forward references (long story)
  - Program-evaluation order
- See manual for **`.cm[i,o]`** files, **`-c`** flag, etc.

Module example

foo.ml:

```
type t1 = X1 of int
        | X2 of int

let get_int t =
  match t with
    X1 i -> i
  | X2 i -> i

type even = int

let makeEven i = i*2
let isEven1 i = true
(* isEven2 is "private" *)
let isEven2 i = (i mod 2)=0
```

foo.mli

```
(* choose to show *)
type t1 = X1 of int
        | X2 of int

val get_int : t1->int

(* choose to hide *)
type even

val makeEven : int->even
val isEven1 : even->bool
```

Module example

bar.ml:

```
type t1 = X1 of int
        | X2 of int

let conv1 t =
  match t with
    X1 i -> Foo.X1 i
  | X2 i -> Foo.X2 i
let conv2 t =
  match t with
    Foo.X1 i -> X1 i
  | Foo.X2 i -> X2 i

let _ =
 Foo.get_int(conv1(X1 17));
 Foo.isEven1(Foo.makeEven 17)
 (* Foo.isEven1 34 *)
```

foo.mli

```
(* choose to show *)
type t1 = X1 of int
        | X2 of int

val get_int : t1->int

(* choose to hide *)
type even

val makeEven : int->even
val isEven1 : even->bool
```

Not the whole language

- Objects
- Loop forms (bleach!)
- Fancy module stuff (functors)
- Polymorphic variants
- Mutable fields
- Catching exceptions; exceptions carrying values

Just don't need any of this for class
(nor do I use it much)

Summary

- Done with Caml tutorial
  - Focus on “up to speed” while being precise
  - Much of class will be *more* precise

- Now functional-programming idioms (next time?)
  - Uses of higher-order functions (cf. objects)
  - Tail recursion
  - Life without mutation or loops

  Will use Caml but ideas are more general

5 closure idioms

1. Create similar functions
2. Pass functions with private data to iterators
3. Combine functions
4. Provide an abstract data type
5. Callbacks without fixing environment type

Create similar functions

```
let addn m n = m + n

let add_one m = addn 1

let add_two m = addn 2

let rec f m =
  if m=0
  then []
  else (addn m)::(f (m-1))
```

Private data for iterators

```
let rec map f lst =
  match lst with
     []      -> []
   | hd::tl -> (f hd)::(map f tl)

(* just a function pointer *)
let incr lst = map (fun x -> x+1) lst
let incr = map (fun x -> x+1)
(* a closure *)
let mul i lst = map (fun x -> x*i) lst
let mul i = map (fun x -> x*i)
```

A more powerful iterator

```
let rec fold_left f acc lst =
  match lst with
    []     -> acc
  | hd::tl -> fold_left f (f acc hd) tl

(* just function pointers *)
let f1 = fold_left (fun x y -> x+y) 0
let f2 = fold_left (fun x y -> x && y>0) true
(* a closure *)
let f1 lst lo hi =
 fold_left
 (fun x y -> if y>lo && y<hi then x+1 else x)
 0
 lst
```

Thoughts on fold

- Functions like fold decouple recursive traversal ("walking") from data processing
- No unnecessary type restrictions
- Similar to visitor pattern in OOP
  - Private fields of a visitor like free variables
- Very useful if recursive traversal hides fault tolerance (thanks to no mutation) and massive parallelism

MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
6th Symposium on Operating System Design and Implementation
2004

Combine functions

```
let f1 g h = (fun x -> g (h x))

type 'a option = None | Some of 'a (*predefined*)

let f2 g h x =
  match g x with
    None   -> h x
   | Some y -> y
```

Provide an ADT

- Note: This is mind-bending stuff

```
type set = { add : int -> set;
             member : int -> bool }
let empty_set =
  let exists lst j = (*could use fold_left!*)
     let rec iter rest =
        match rest with
          [] -> false
        | hd::tl -> j=hd || iter tl in
     iter lst in
  let rec make_set lst =
    S { add    = (fun i -> make_set(i::lst));
        member = exists lst } in
  make_set []
```

Callbacks

- Library takes a function to apply later, on an event:
  - When a key is pressed
  - When a network packet arrives
  - …
- Function may be a filter, an action, …
- Different callbacks may need private state of different types
- Fortunately, a function's type does not depend on the types of its free variables
- Compare OOP: subclass (often anonymous)
- Compare C: a **`void*`** argument (the environment)