CSEP505: Programming Languages
Lecture 1: Intro; OCaml; Functional Programming

Dan Grossman

Autumn 2016

Welcome!

10 weeks for key programming-language concepts

- Focus on the universal foundations

Today:

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

Hello, my name is…

- **Dan** Grossman, djg@cs
- 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, ***other***
- Approximately 0 years professional experience...
  - …but I’ve done a lot of compiler hacking
- Father of two boys < 3 years old
- …

Course facts (overview)

- http://courses.cs.washington.edu/courses/csep505/16au/
- TA: John Toman, Ph.D. student advised by me
- Pre-course survey
- Homework 0 and Homework 1
- No textbook
- 5 homeworks
- OCaml/F#/Haskell
- Take-home final exam much later

Then onto actual course motivation and content

Course web page

- Read syllabus
  - includes some advice

- Read advice for approaching homework
  - Homework code is not industry code
  - Functional programming is not imperative/OOP

- Course web page will have slides, code, homework, programming resources, etc.

TA

John

- Knows his stuff ☺
- In general, email both of us with questions to reduce latency
- John will do the grading
- …?

Survey

- An optional, brief and extremely useful survey
- On the web page (Google form)
- Things like what you do and what your concerns are
- (Also helps me learn your names)

Homework 0

- Install software, edit file, compile, run
- Not worth any points, but highly recommended before next week

Homework 1

- A real homework
- Due in 2 weeks
  - Will generally do every-other-week because Life.
  - Encourage you to start for real before next week.

Wide background

- Homework 1 will likely demonstrate a wide range of background
  - So some material will be simultaneously too remedial and too advanced
  - Still let me know (politely ☺)
  - “Challenge problems” help some
    - Affect your grade, but only a little

- Speaking of background, no need for PMP/5th-year mutual fear

Segue to a sermon

- I’m here to teach the essential beauty of the foundations of programming languages
- If you’re here because
  - The other courses looked even worse
  - You can get out of the house on Thursday nights
  - “A Master’s degree” will get you a raise

  then you risk taking “longcuts” and being miserable
- Advice: If you must be <100% engaged, try to wait as long as possible – the material builds more than it seems
  - Catching up is hard

No textbook

- There just isn't a book that covers this stuff well
  - And the classic research papers are too old to be readable
- Pierce book: Very good, with about 25% overlap with the course
- Many undergraduate-level books, none of which I've used or liked
- O'Reilly book on OCaml is free (in English)
- Will post relevant recent papers as interesting optional reading (rarely good for learning material)
- I do have videos from 2009, but I plan to change ~30% and I've learned a lot since then

Homework

- 5 assignments
  - Mostly OCaml/F# programming (some written answers)
    - Probably one in Haskell
  - Expect to learn as you do them
  - “Not a lot of lines”
  - Again, challenge problems are optional

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

- OCaml vs. F#
  - See also lots of detail on web page

Final exam

- Please do not panic about taking an exam
- Worth 2/7 of the course grade (2x 1 homework)
- Why an exam?
  - Helps you learn material as the course goes on
  - Helps you learn material as you study for it
- I'll post a sample [much] later

OCaml

- OCaml is an awesome, high-level language
- We'll use a small core subset that is well-suited to manipulating recursive data structures (like programs)
- Tutorial will demonstrate its *mostly functional* nature
  - Most data immutable
  - Recursion instead of loops
  - Lots of passing/returning functions
- Again, will support F# as a fine alternative

Welcome!

10 weeks for key programming-language concepts

- Focus on the universal foundations

Today:

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

A question

- What’s the best kind of car?

- What’s the best kind of 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 code fast
- 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-string (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
- OCaml lets these essentials shine
  - Like the DEC Alpha in computer architecture

No: Real differences at formal and informal levels

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?
- Who will I be able to recruit?

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?

And some stuff is just cool

- We certainly should connect the theory in this course to real-world programming issues
  - Though maybe more later in the course after the basics

- But even if we don't, some truths are so beautiful and perspective-altering they are worth learning anyway
  - Watching Hamlet should affect you
    - Maybe very indirectly
    - Maybe much later
    - And maybe you need to re-watch it

Academic languages

Aren’t academic languages worthless?

- Yes: fewer jobs, less tool support, etc.
  - But a lot has changed in the last decade
- No:
  - Knowing them makes you a better programmer
  - Java did not exist in 1993; what doesn’t exist now
  - Eventual vindication (on the leading edge):

    garbage-collection, generics, function closures, iterators, universal data format, … (what’s next?)
  - We don’t conquer; we assimilate
    - And get no credit (fine by me)
  - Functional programming is “finally cool”-ish

“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!
- 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 defining 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
  - Learn to specify all your corner cases via elegant composition

Last Motivation: “Fan Mail”

Today I had to do some work with a minimal browser shell around Internet Explorer (for work), and found that I didn't have my usual Javascript debugging tools. So I tried to write a small "immediate window" for Javascript so I could conveniently execute commands. I started off knowing I'd probably use some eval(), but only a little while in, I realized the naive approach wasn't going to work, because eval() does its evaluation in the current context… [snip] I eventually got it to work using some eval tricks and some closure tricks. I am 100% sure that if I had not taken your mind-bending class, there's no way I could have figured this out, so I wanted to share it with you.

Last Motivation: "Fan Mail"

I was starting my first week at Google, all fresh-faced and eager to impress. As a the newest employee on the team, my co-workers gave me the task of sanity-checking the newly written Dart language spec (and it would be a good way to introduce me to the language). The specification was filled with operational and denotational semantics, and thanks to what I learned in 505 I was able to reasonably easily read through the document and get up to speed on Dart!

Last Motivation: “Fan Mail”

Hi Dan, I've been meaning to get around to doing this, but I wanted to tell you about the impact that your class had on me when I took it back in 2008. I'm not exaggerating when I say that I've been digesting it for the last six years and I've gone through the course notes at least once a year. I continue to learn more and more as time goes on.

The one thing I'd say is that it is immediately clear when you enter industry that there are two types of programmers - ones that have a basic understanding of PL fundamentals and ones that do not. The conversations you'd have with each of these types are extremely different. If someone lacks a basic understanding of PL, they're much more likely to dogmatically adhere to patterns and practices that are suboptimal or, more typically, just don't matter that much.

Last Motivation: “Fan Mail”

Long time, no see ;) I figured I'd drop you a line about the latest project I've been working on for a few months: [snip]. I took [snip] and added a streaming SQL layer on top. Finally, a chance to apply my hard-won 505 knowledge to something out here in the so-called "real world." I even had to pull out the Pierce book at one point.

Last Motivation: “Fan Mail”

I also wanted to mention that even though I was against the idea of an exam before the quarter started, I thought your exam was fair and even fun. It was stressful to study for, but I'm hopeful that the concepts have sunk in better now than if I hadn't studied.

Last Motivation: “Fan Mail”

Dan, I just wanted to thank you for a truly mind-stretching semester. I enjoyed it a lot; it was worth every penny (out of my own pocket).
You've given me insight and perspective on so many things.

I've even been caught twice now by my colleagues, speaking in terms of, "well, that would depend on the intended semantics of the programming language". :)

Last Motivation: “Fan Mail”

*I just came across continuations by accident while I was looking at comparisons of lua with other languages. I completely forgot we had gone over those in your class, and am beating myself up for not using them *ALL THE TIME* in my code - they are awesome! Why are languages the coolest?!*

Last Motivation: “Fan Mail”

This class has changed the way I think about programming - even if I don’t get to use all of the concepts we explored in OCaml (I work in C++ most of the time), understanding more of the theory makes a tremendous difference to how I go about solving a problem.

Welcome!

10 weeks for key programming-language concepts

- Focus on the universal foundations

Today:

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

And now OCaml

- “Hello, World”, compiling, running, etc.
  - Demo
- Tutorial on the language
  - Mostly via demo but slides has similar/identical code
  - *Heavily* skewed toward what we need to study PL
- Then use our new language to learn
  - Functional programming
  - Idioms using higher-order functions
  - Benefits of not mutating variables
- Then use OCaml to *define* other (made-up) languages
  - Probably next week?

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 pr*evious* 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
```

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:
  - P505-specific instructions
  - **`www.ocaml.org`**
  - The on-line manual (fine reference)
  - An on-line book (less of a reference)

- Contact us with install problems soon!

- Ask questions (we know the language, want to share)
  - But 100 rapid-fire questions not the way to learn

Types

- Every expression has a 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->string *)
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 (optional in F#)

```
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
  - Anonymous functions cannot be recursive

```
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 *)
(*note 2-args and useful but unused polymorphism*)
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 OCaml 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
```

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
```

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)

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

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 x = 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 (compelling uses next time)
- 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.a_first + f x.a_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 food = 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 len lst = (* 'a lst -> int *)
  match lst with
    Null -> 0
  | Cons(x,rest) -> 1 + len 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 -> 'a lst *)
  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**
- 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
  - Standard-library
  - Common higher-order function idioms
  - Tuples
  - Nested patterns
  - Exceptions
- Then (simple) modules

Standard library

- Values (e.g., functions) bound to **`foo`** in module **`M`** are accessed via **`M.foo`**
- Standard library organized into modules
- For Homework 1, will use **`List`**, **`String`**, and **`Char`**
  - Mostly **`List`**, for example, **`List.fold_left`**
  - And we point you to the useful functions
- Standard library a mix of "primitives" (e.g., **`String.length`**) and useful helpers written in Caml (e.g., **`List.fold_left`**)
- **`Pervasives`** is a module implicitly "opened"

- F# differs the most here:
  - Different function names
  - Sometimes more OO
  - No Pervasives

Higher-order functions

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

let lst234 = mymap (fun x -> x+1) [1;2;3]
let lst345 = List.map (fun x -> x+1) [1;2;3]
let incr_list = mymap (fun x -> x+1)
```

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 (*poor style *)

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", …)

- Patterns are not expressions, though syntactically a subset
  - Plus some bells/whistles (as-patterns, or-patterns)

- Exhaustiveness and redundancy checking at compile-time!

- **`let`** can have patterns, just sugar for one-branch **`match`**!

Fancy patterns example

```
type sign = P | N | Z

let multsign x1 x2 =
  let sign x =
    if x>0 then (if x=0 then Z else P) 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 type changes)

Fancy patterns example (and exns)

```
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
```

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

Pattern-matching in general

- Full definition of matching is recursive
  - Over a value and a pattern
  - Produce a binding list or fail
  - You implement a simple version in homework 1

- Example:

  (*p1*,*p2*,*p3*) matches **(*v1*,*v2*,*v3*)**

  if ***pi*** matches ***vi*** for **1<=*i*<=3**

  – Binding list is 3 subresults appended together

“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
```

Exceptions

See the manual for:

- Exceptions that carry values
  - Much like datatypes but *extends* **`exn`**
- Catching exceptions
  - **`try e1 with`** …
  - Much like pattern-matching but cannot be exhaustive
- Exceptions are not *hierarchical* (unlike Java/C# subtyping)

Modules

- So far, only way to hide things is local **`let`**
  - Not good for large programs
  - Caml has a fancy *module system*, but we need only the basics

- Modules and signatures give
  - Namespace management
  - Hiding of values and types
  - Abstraction of types
  - Separate type-checking and compilation

- By default, OCaml 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 (e.g., functors)
- Polymorphic variants
- Mutable fields
- …

Just don’t need much of this for class

(nor do I use it much)

- May use floating-point, etc. (easy to pick up)

Summary

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

- Next: functional-programming idioms
  - Uses of higher-order functions (cf. objects)
  - Tail recursion
  - Life without mutation or loops

  Will use OCaml but ideas are more general

- Then: On to implementing PLs and *semantics*