Where are we

Programming:
• To finish: OCaml tutorial (roughly slides 68- from Lecture 1)
• Idioms using higher-order functions
  – Similar-ish to objects
• Tail recursion

Languages:
• Abstract syntax, Backus-Naur Form
• Definition via an interpreter
• Next time: Small-step interpreter and via translation [and more]

Picking up our tutorial

• We did:
  – Recursive higher-order functions
  – Records
  – Recursive datatypes
• "[Lecture 1]" Now some important odds and ends, quickly:
  – Standard-library
  – Tuples
  – Nested patterns
  – Exceptions
• "[Lecture 1]" Then:
  – (Simple) Modules
• Then the-slides-that-follow

6 closure idioms

Closure: Function plus environment where function was defined
  – Environment matters when function has free variables
1. Create similar functions
2. Combine functions
3. Pass functions with private data to iterators
4. Provide an abstract data type
5. Currying and partial application
6. Callbacks

Create similar functions

```ocaml
let addn m n = m + n
let add_one = addn 1
let add_two = addn 2
let rec f m = 
  if m=0
  then []
  else (addn m)::(f (m-1))
let lst65432 = List.map (fun x -> x 1) (f 5)
```

Combine functions

```ocaml
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
  (* just a function pointer *)
let print_int = f1 print_string string_of_int
  (* a closure *)
let truncate1 lim f = f1 (fun x -> min lim x) f
let truncate2 lim f = f1 (min lim) f
```
Also: Pipeline Operator

```.ml
let (|>) x f = f x
```

```ml
-34 |> abs |> string_of_int |> compare "34"
```

(* versus *)

```ml
compare "34" (string_of_int (abs (-34))
```

---

Private data for iterators

```ml
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

```ml
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 f3 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

---

Provide an ADT

```ml
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
  let rec make_set lst =
    { add    = (fun i -> make_set(i::lst));
      member = exists lst } in
  make_set []
```

---

Thoughts on ADT example

- Note: This is mind-bending stuff
- By "hiding the list" behind the functions, we know clients do not assume the representation
  - Why? All you can do with a function is apply it
    - No other primitives on functions
    - No reflection
    - No aspects
    - …
### Currying

- We’ve been using currying a lot
  - Efficient and convenient in OCaml
  - (Partial application not efficient, but still convenient)
- Just remember that the semantics is to build closures:
  - More obvious when desugared:

  ```ocaml
  let f = fun x -> (fun y -> (fun z -> ... ))
  let a = ((f 1) 2) 3
  ```

### 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, ...
- Various callbacks need private state of different types
- Fortunately, a function’s type does not depend on the types of its free variables

### Recursion and efficiency

- Recursion is more powerful than loops
  - Just pass loop state as another argument
- But isn’t it less efficient?
  - Function calls more time than branches?
    - Compiler’s problem
      - An O(1) detail irrelevant in 99+% of code
  - More stack space waiting for return
    - Shared problem: use tail calls where it matters
    - An O(n) issue (for recursion-depth n)

### Tail recursion example

(* factorial *)

```ocaml
let rec fact1 x =
  if x==0 then 1 else x * (fact1(x-1))
```

- More complicated, more efficient version

```ocaml
let rec fact2 x =
  let rec f acc x =
    if x==0 then acc else f (acc*x) (x-1)
  in
  f 1 x
```

- **Accumulator** pattern (base-case becomes initial accumulator)

### Another example

```ocaml
let rec sum1 lst =
  match lst with
  | []     -> 0
  | hd::tl -> hd + (sum1 tl)
let sum2 lst =
  let rec f acc lst =
    match lst with
    | []     -> acc
    | hd::tl -> f (acc+hd) tl
  in
  f 0 lst
```

- Again O(n) stack savings
- But input was already O(n) size
Half-example

```
type tree = Leaf of int | Node of tree * tree

let sum tr =
    let rec f acc tr =
        match tr with
        | Leaf i -> acc+i
        | Node(left,right) -> f (f acc left) right
    in
    f 0 tr
```

- One tail-call, one non
- Tail recursive version will build O(n) worklist
  - No space savings
  - That’s what the stack is for!
- O(1) space requires mutation and no re-entrancy

Informal definition

If the result of \( f \) is the result of the enclosing function, then the call is a tail call (in tail position):

- In \((\text{fun x -> e})\), the \( e \) is in tail position.
- If \( \text{if e1 then e2 else e3} \) is in tail position, then \( e2 \) and \( e3 \) are in tail position.
- If \( \text{let p = e1 in e2} \) is in tail position, then \( e2 \) is in tail position.
- ...

- Note: for call \( e1 \ e2 \), neither is in tail position

Defining languages

- We have built up some terminology and relevant programming prowess
- Now
  - What does it take to define a programming language?
  - How should we do it?

Syntax vs. semantics

Need: what every string means:
“Not a program” or “produces this answer”

Typical decomposition of the definition:
1. Lexing, a.k.a. tokenization, string to token list
2. Parsing, token list to labeled tree (AST)
3. Type-checking (a filter)
4. Semantics (for what got this far)

For now, ignore (3) (accept everything) and skip (1)-(2)

Abstract syntax

To ignore parsing, we need to define trees directly:

- A tree is a labeled node and an ordered list of (zero or more) child trees.
- A PL’s abstract syntax is a subset of the set of all such trees:
  - What labels are allowed?
  - For a label, what children are allowed?

Advantage of trees: no ambiguity, i.e., no need for parentheses

Syntax metalanguage

- So we need a metalanguage to describe what syntax trees are allowed in our language.
- A fine choice: OCaml datatypes

```
  type exp = Int of int | Var of string
             | Plus of exp * exp | Times of exp * exp

  type stmt = Skip | Assign of string * exp
               | Seq of stmt * stmt
               | If of exp * stmt * stmt
               | While of exp * stmt
```

- +: concise and direct for common things
- -: limited expressiveness (silly example: nodes labeled Foo
must have a prime-number of children)
- In practice: push such limitations to type-checking
We defined a subset?

- Given a tree, does the datatype describe it?
  - Is root label a constructor?
  - Does it have the right children of the right type?
  - Recur on children
- Worth repeating: a finite description of an infinite set
  - (all?) PLs have an infinite number of programs
  - Definition is recursive, but not circular!
- Made no mention of parentheses, but we need them to “write a tree as a string”

BNF

A more standard metalanguage is Backus-Naur Form

- Common: should know how to read and write it

```
e ::= c | x | e + e | e * e
s ::= skip | x := e | s; s | if e then s else s | while e s
(x in (x1,x2,...,y1,y2,...,z1,z2,...,...))
(c in (...,-2,-1,0,1,2,...))
```

Also defines an infinite set of trees. Differences:
- Different metanotation ( ::= and | )
- Can omit labels (constructors), e.g., “every c is an e”
- We changed some labels (e.g., := for Assign)

Ambiguity revisited

- Again, metalanguages for abstract syntax just assume there are enough parentheses
- Bad example:
  ```
  if x then skip else y := 0; z := 0
  ```
- Good example:
  ```
  y:=1; (while x (y:=y*x; x:= x-1))
  ```

Our first PL

- Let’s call this dumb language IMP
  - It has just mutable ints, a while loop, etc.
  - No functions, locals, objects, threads, …

Defining it:
1. Lexing (e.g., what ends a variable)
2. Parsing (make a tree from a string)
3. Type-checking (accept everything)
4. Semantics (to do)

You’re not responsible for (1) and (2)! Why…

Syntax is boring

- Parsing PLs is a computer-science success story
- “Solved problem” taught in compilers
- Boring because:
  - “If it doesn’t work (efficiently), add more keywords/parentheses”
  - Extreme: put parentheses on everything and don’t use infix
    - 1950s example: LISP (foo …)
    - 1990s example: XML <foo> … </foo>
- So we’ll assume we have an AST

(Counter-argument: Parsing still a pain and source of security vulnerabilities in practice.)

Toward semantics

```
e ::= c | x | e + e | e * e
s ::= skip | x := e | s; s | if e then s else s | while e s
(x in (x1,x2,...,y1,y2,...,z1,z2,...,...))
(c in (...,-2,-1,0,1,2,...))
```

Now: describe what an AST “does/is/computes”
- Do expressions first to get the idea
- Need an informal idea first
  - A way to “look up” variables (the heap)
- Need a metalanguage
  - Back to OCaml (for now)
An expression interpreter

- Definition by interpretation: Program means what an interpreter written in the metalanguage says it means

```
type exp = Int of int | Var of string | Plus of exp * exp | Times of exp * exp

type heap = (string * int) list

let rec lookup h str = ... (*lookup a variable*)

let rec interp_e (h:heap) (e:exp) =
  match e with
  | Int i       -> i
  | Var str     -> lookup h str
  | Plus(e1,e2) -> (interp_e h e1)+(interp_e h e2)
  | Times(e1,e2)-> (interp_e h e1)*(interp_e h e2)
```

Not always so easy

```
let rec interp_e (h:heap) (e:exp) =
  match e with
  | Int i       -> i
  | Var str     -> lookup h str
  | Plus(e1,e2) -> (interp_e h e1)+(interp_e h e2)
  | Times(e1,e2)-> (interp_e h e1)*(interp_e h e2)
```

• By fiat, “IMP’s plus/times” is the same as OCaml’s
- A metalanguage exception may be inappropriate
- So define lookup to return 0 by default?
- What if we had division?

On to statements

- A wrong idea worth pursuing:

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
  | Skip -> ()
  | Seq(s1,s2) -> interp_s h s1 ; interp_s h s2
  | If(e,s1,s2) -> if interp_e h e then interp_s h s1 else interp_s h s2
  | Assign(str,e) -> (* ??? *)
  | While(e,s1) -> (* ??? *)
```

What went wrong?

- In IMP, expressions produce numbers (given a heap)
- In IMP, statements change heaps, i.e., they produce a heap (given a heap)

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
  | Skip -> ()
  | Seq(s1,s2) -> let h2 = interp_s h s1 in interp_s h2 s2
  | If(e,s1,s2) -> if (interp_e h e) <> 0 then interp_s h s1 else interp_s h s2
  | Assign(str,e) -> update h str (interp_e h e)
  | While(e,s1) -> (* ??? *)
```

About that heap

- In IMP, a heap maps strings to values
- Yes, we could use mutation, but that is:
  - less powerful (old heaps do not exist)
  - less explanatory (interpreter passes current heap)

```
type heap = (string * int) list

let rec lookup h str =
  match h with
  | [] -> 0 (* kind of a cheat *)
  | [(s,i)]:tl -> if s=str then i else lookup tl str

let update h str i = ...
```

• As a definition, this is great despite terrible waste of space

Meanwhile, while

- Loops are always the hard part!

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
  | While(e,s1) -> if (interp_e h e) <> 0 then let h2 = interp_s h s1 in interp_s h2 s else h
```

- s is While(e,s1)
  - Semi-troubling circular definition
    - That is, interp_s might not terminate
Finishing the story

- Have interp_e and interp_s
- A “program” is just a statement
- An initial heap is (say) one that maps everything to 0

```ocaml
type heap = (string * int) list
let empty_heap = []
let interp_prog s =
  lookup (interp_s empty_heap s) "ans"
```

Fancy words: We have defined a large-step operational-semantics using OCaml as our metalanguage

Fancy words

- Operational semantics
  - Definition by interpretation
  - Often implies metalanguage is “inference rules” (a mathematical formalism we’ll learn in a couple weeks)
- Large-step
  - Interpreter function “returns an answer” (or diverges)
  - So definition says nothing about intermediate computation
  - Simpler than small-step when that’s okay

Language properties

- A semantics is necessary to prove language properties
- Example: Expression evaluation is total and deterministic
  “For all heaps h and expressions e, there is exactly one integer i such that interp_e h e returns i.”
  - Rarely true for “real” languages
  - But often care about subsets for which it is true
- Prove for all expressions by induction on the tree-height of an expression

Small-step [In Lecture 3]

- Now redo our interpreter with small-step
  - An expression/statement “becomes a slightly simpler thing”
  - A less efficient interpreter, but has advantages as a definition (discuss after interpreter)

<table>
<thead>
<tr>
<th></th>
<th>Large-step</th>
<th>Small-step</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp_e</td>
<td>heap-&gt;exp-&gt;int</td>
<td>heap-&gt;exp-&gt;exp</td>
</tr>
<tr>
<td>interp_s</td>
<td>heap-&gt;stmt-&gt;heap</td>
<td>heap-&gt;stmt-&gt;(heap*stmt)</td>
</tr>
</tbody>
</table>