CSEP505: Programming Languages
Lecture 2: functional programming, syntax, semantics via interpretation or translation

Dan Grossman

Spring 2006

Where are we

Programming:

- "Done": Caml tutorial, definition of functions
- Idioms using higher-order functions
  - Similar to objects
- Tail recursion, informally

Languages:

- Abstract syntax, Backus-Naur Form
- Definition via interpretation
- Definition via translation

I'll be shocked to finish these slides today; that's okay

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 = addn 1

let add_two = 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 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

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

Thoughts on ADT example

- 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 elimination forms
  - No reflection
  - No aspects
  - …

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

Callbacks cont'd

```
type event = …
val register_callback : (event->unit)->unit
```

- Compare OOP: subclassing for private state

```
abstract class EventListener {
  abstract void m(Event); //"pure virtual"
}
void register_callback(EventListener);
```

- Compare C: a `void*` arg for private state

```
void register_callback
(void*, void (*)(void*,Event);
// void* and void* better be compatible
// callee must pass back the same void*
```

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

- More complicated, more efficient version

```
let 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 -- initial accumulator)

Another example

```
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 x` is the result of the enclosing function, then the call is a tail call (in tail position):

- In `(fun x -> e)`, the `e` is in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position.
- If `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 (by definition)

Syntax metalanguage

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

- Plus: concise and direct for common things
- Minus: 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, 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 – see lex.mll)
2. Parsing (make a tree – see parse.mly)
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

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 Caml (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 Caml's
- We assume lookup always returns an int
  - 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) -> let _ = interp_s h s1 in
                 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 -> h
  |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 = (str,i)::h
```

- As a *definition*, this is great despite wasted 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

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

let mt_heap = [] (* common PL pun *)

let interp_prog s =
  lookup (interp_s mt_heap s) "ans"
```

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

Fancy words

- Operational semantics
  - Definition by interpretation
  - Often implies metalanguage is “inference rules” (a mathematical formalism I may show later)
- Large-step
  - Interpreter function “returns an answer” (or diverges)
  - So caller cannot say anything 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`"
- Easier to state/prove because our metalanguage is not using mutation, threads, exceptions, etc.
  - Want a metalanguage as simple as possible

Example "proof"

"For all heaps `h` and expressions `e`, there is exactly one integer `i` such that `interp_e h e` returns `i`"

```
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 induction on the height of the tree `e`

- Base case: height is 1 (then e is Int or Var) …
- Inductive case: height is n>1 (so e is Plus or Times)…

Small-step

- Now redo our interpreter with small-step
  - An expression or statement "becomes a slightly simpler thing"
  - A less efficient interpreter, but a more revealing definition and uses less of metalanguage
  - *Equivalent* (more later)

	Large-step	**Small-step**
`interp_e`	`heap->exp->int`	`heap->exp->exp`
`interp_s`	`heap->stmt->heap`	`heap->stmt->` `(heap*stmt)`

Example

Switching to concrete syntax, where each → is one call to `interp_e` and heap is empty:

(x+3)+(y*z) → (0+3)+(y*z)
→ 3+(y*z)
→ 3+(0*z)
→ 3+(0*0)
→ 3+0
→ 3

Small-step expressions

"We just take one little step"

```
exception AlreadyValue
let rec interp_e (h:heap) (e:exp) =
 match e with
   Int i    -> raise AlreadyValue
  |Var str -> Int (lookup h str)
  |Plus(Int i1,Int i2) -> Int (i1+i2)
  |Plus(Int i1, e2) -> Plus(Int i1,interp_e h e2)
  |Plus(e1, e2) -> Plus(interp_e h e1,e2)
  |Times(Int i1,Int i2) -> Int (i1*i2)
  |Times(Int i1, e2)-> Times(Int i1,interp_e h e2)
  |Times(e1, e2) -> Times(interp_e h e1,e2)
```

We chose "left to right", but not important

Small-step statements

```
let rec interp_s (h:heap) (s:stmt) =
 match s with
  Skip                -> raise AlreadyValue
 |Assign(str,Int i)->  ((update h str i),Skip)
 |Assign(str,e)      -> (h,Assign(str,interp_e h e))
 |Seq(Skip,s2)       -> (h,s2)
 |Seq(s1,s2)         -> let (h2,s3) = interp_s h s1
                        in (h2,Seq(s3,s2))
 |If(Int i,s1,s2)    -> (h, if i <> 0
                            then s1
                            else s2)
 |If(e,s1,s2) -> (h, If(interp_e h e, s1, s2))
 |While(e,s1) -> (*????*)
```

Meanwhile, `while`

- Loops are *always* the hard part!

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
    …
  | While(e,s1) -> (h, If(e,Seq(s1,s),Skip))
```

  - "A loop is equal to its unrolling"
  - `s` is `While(e,s1)`
  - `interp_s` always terminates
  - `interp_prog` may 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

```
type heap = (string * int) list
let mt_heap = [] (* common PL pun *)
let interp_prog s =
  let rec loop (h,s) =
     match s with
       Skip -> lookup h "ans"
     | _    -> loop (interp_s h s)
  in loop (mt_heap,s)
```

Fancy words: We have defined a small-step operational-semantics using Caml as our metalanguage

Small vs. large again

- Small is really inefficient (descends and rebuilds AST at every tiny step)
- But as a definition, it gives a trace of program states (pairs of heap*stmt)
  - Can talk about them e.g., "no state has x>17…"
- Theorem: Total equivalence: **`interp_prog`** (large) returns **`i`** for **`s`** if and only if **`interp_prog`** (small) does
- With the theorem, we can choose whatever semantics is most convenient

Where are we

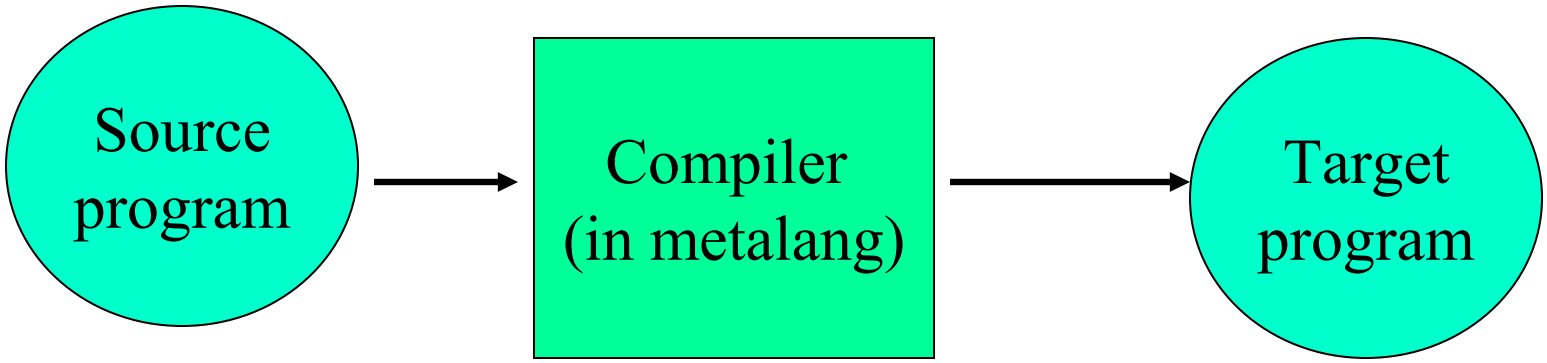
Definition by interpretation

- We have *abstract syntax* and two *interpreters* for our *source language* IMP
- Our metalanguage is Caml

Now definition by translation

- Abstract syntax and source language still IMP
- Metalanguage still Caml
- *Target language* now "Caml with just functions strings, ints, and conditionals" tricky stuff?

In pictures and equations



- If the target language has a semantics, then:
  compiler + targetSemantics = sourceSemantics

Deep vs. shallow

- Meta and target can be the same language
  - Unusual for a “real” compiler
  - Makes example harder to follow ☹
- Our target will be a subset of Caml
  - After translation, you could (in theory) “unload” the AST definition
  - This is a “deep embedding”
    - An IMP while loop becomes a function
    - Not a piece of data that says “I’m a while loop”

Goals

- **`xlate_e:`**

  `exp -> ((string->int)->int)`

  – "given an exp, produce a function that given a function from strings to ints returns an int"

  – (**`string->int`** acts like a heap)

  – An expression "is" a function from heaps to ints

- **`xlate_s`**:

  `stmt->((string->int)->(string->int))`

  – A statement "is" a function from heaps to heaps

Expression translation

```
xlate_e: exp -> ((string->int)->int)
```

```
let rec xlate_e (e:exp) =
  match e with
    Int i          ->  (fun h -> i)
  | Var str        ->  (fun h -> h str)
  | Plus(e1,e2)    ->  let f1 = xlate_e e1 in
                       let f2 = xlate_e e2 in
                       (fun h -> (f1 h) + (f2 h))
  | Times(e1,e2)   -> let f1 = xlate_e e1 in
                       let f2 = xlate_e e2 in
                       (fun h -> (f1 h) * (f2 h))
```

What just happened

```
(* an example *)
let e = Plus(Int 3, Times(Var "x", Int 4))
let f = xlate_e e (* compile *)
(* the value bound to f is a function whose body
   does not use any IMP abstract syntax! *)
let ans = f (fun s -> 0)(* run w/ empty heap *)
```

- Our target sublanguage:
  - Functions (including `+` and `*`, not `interp_e`)
  - Strings and integers
  - Variables bound to things in our sublangage
  - (later: if-then-else)
- Note: No lookup until "run-time" (of course)

Wrong

- This produces a program not in our sublanguage:

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

- Caml evaluates function bodies when called (like YFL)
- Waits until run-time to translate Plus and Times children!

Statements, part 1

```
xlate_s:
    stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with
  Skip           -> (fun h -> h)
 |Assign(str,e) ->
   let f = xlate_e e in
   (fun h -> let i = f h in
             (fun s -> if s=str then i else h s))
 |Seq(s1,s2) ->
    let f2 = xlate_s s2 in (* order irrelevant *)
    let f1 = xlate_s s1 in
    (fun h -> f2 (f1 h)) (* order relevant *)
 | …
```

Statements, part 2

```
xlate_s:
    stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with …
  |If(e,s1,s2) ->
    let f1 = xlate_s s1 in
    let f2 = xlate_s s2 in
    let f  = xlate_e e  in
    (fun h -> if (f h) <> 0 then f1 h else f2 h)
  |While(e,s1) ->
    let f1 = xlate_s s1 in
    let f  = xlate_e e  in
    (*???*)
```

- Why is translation of while tricky???

Statements, part 3

```
xlate_s:
    stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with
 …
 |While(e,s1) ->
   let f1 = xlate_s s1 in
   let f  = xlate_e e  in
   let rec loop h = (* ah, recursion! *)
     if f h <> 0
     then loop (f1 h)
     else h
   in loop
```

- Target language *must* have some recursion/loop!

Finishing the story

- Have `xlate_e` and `xlate_s`
- A "program" is just a statement
- An initial heap is (say) one that maps everything to 0

```
let interp_prog s =
  ((xlate_s s) (fun str -> 0)) "ans"
```

Fancy words: We have defined a "denotational semantics" (but target was not math)

Summary

- Three semantics for IMP
  - Theorem: they are all equivalent
- Avoided (for now?) teaching
  - Inference rules (for "real" operational semantics)
  - Recursive-function theory (for "real" denotational semantics)
- Inference rules useful for reading PL research papers (so we will probably do it)
- If we assume Caml already has a semantics, then using it as a metalanguage and target language makes sense for IMP
- Loops and recursion are deeply connected!

Packet filters

- If you're not a language semanticist, is this useful?

Almost everything I know about packet filters:

- Some bits come in off the wire
- Some applications want the "packet" and some do not (e.g., port number)
- For safety, only the O/S can access the wire
- For extensibility, only apps can accept/reject packets

Conventional solution goes to user-space for every packet and app that wants (any) packets.

Faster solution: Run app-written filters in kernel-space

Language-based approaches

1. Interpret a language
   - + clean operational semantics, portable
   - - *may* be slow, unusual interface
2. Translate (JIT) a language into C/assembly
   - + clean denotational semantics, existing optimizers,
   - - upfront (pre-1st-packet) cost, unusual interface
3. Require a conservative subset of C/assembly
   - + normal interface
   - - too conservative without help
   - Related to type systems (we'll get there!)

More generally…

Packet filters move the code to the data rather than data to the code

- General reasons: performance, security, other?
- Other examples:
  - Query languages
  - Active networks