CMSC 330: Organization of Programming Languages

Final Exam Review

Review Choices

- OCaml
  - closures, currying, etc
- Threads
  - data races, synchronization, classic probs
- Java Generics
- Topics
  - garbage collection, exceptions, parameters
- Semantics and Lambda Calculus

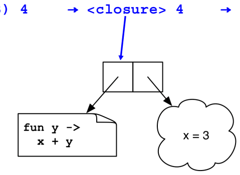
Environments and Closures

- An *environment* is a mapping from variable names to values
  - Just like a stack frame
- A *closure* is a pair (f, e) consisting of function code f and an environment e
- When you invoke a closure, f is evaluated using e to look up variable bindings

Example

```
let add x = (fun y -> x + y)
```

```
(add 3) 4   → <closure> 4   → 3 + 4   → 7
```



Curried Functions in OCaml

- OCaml has a really simple syntax for currying

```
let add x y = x + y
```

  - This is identical to all of the following:

```
let add = (fun x -> (fun y -> x + y))
let add = (fun x y -> x + y)
let add x = (fun y -> x+y)
```

- Thus:
  - `add` has type `int -> (int -> int)`
  - `add 3` has type `int -> int`
    - The return of add x evaluated with x = 3
    - `add 3` is a function that adds 3 to its argument
  - `(add 3) 4 = 7`
- This works for any number of arguments

Curried Functions in OCaml (cont'd)

- Because currying is so common, OCaml uses the following conventions:
  - `->` associates to the right
    - Thus `int -> int -> int` is the same as
    - `int -> (int -> int)`
  - function application associates to the left
    - Thus `add 3 4` is the same as
    - `(add 3) 4`

Another Example of Currying

- A curried add function with three arguments:

```
let add_th x y z = x + y + z
```

  – The same as

```
let add_th x = (fun y -> (fun z -> x+y+z))
```

- Then...
  - `add_th` has type `int -> (int -> (int -> int))`
  - `add_th 4` has type `int -> (int -> int)`
  - `add_th 4 5` has type `int -> int`
  - `add_th 4 5 6` is `15`

Data Types

```
type shape =
    Rect of float * float     (* width * length *)
  | Circle of float           (* radius *)

let area s =
  match s with
      Rect (w, l) -> w *. l
    | Circle r -> r  *. r *. 3.14

area (Rect (3.0, 4.0))
area (Circle 3.0)
```

- `Rect` and `Circle` are *type constructors*- here a `shape` is either a `Rect` or a `Circle`
- Use pattern matching to *deconstruct* values, and do different things depending on constructor

Data Types, con't.

```
type shape =
    Rect of float * float     (* width * length *)
  | Circle of float

let l = [Rect (3.0, 4.0) ; Circle 3.0; Rect (10.0,
  22.5)]
```

- What's the type of `l`? `shape list`
- What's the type of `l`'s first element? `shape`

Polymorphic Data Types

```
type 'a option =
    None
  | Some of 'a

let add_with_default a = function
    None -> a + 42
  | Some n -> a + n

add_with_default 3 None       (* 45 *)
add_with_default 3 (Some 4)   (* 7 *)
```

- This option type can work with any kind of data
  – In fact, this option type is built-in to OCaml

Recursive Data Types

- Do you get the feeling we can build up lists this way?

```
type 'a list =
   Nil
 | Cons of 'a * 'a list

let rec length l = function
   Nil -> 0
 | Cons (_, t) -> 1 + (length t)

length (Cons (10, Cons (20, Cons (30, Nil))))
```

  – Note: Don't have nice [1; 2; 3] syntax for this kind of list

Creating a Module

```
module Shapes =
  struct
    type shape =
       Rect of float * float     (* width * length *)
     | Circle of float           (* radius *)

    let area = function
       Rect (w, l) -> w *. l
     | Circle r -> r  *. r *. 3.14

    let unit_circle = Circle 1.0
  end;;

unit_circle;;    (* not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
open Shapes;;    (* import all names into current scope *)
unit_circle;;    (* now defined *)
```

Module Signatures

Entry in signature

Supply function types

Give type to module

```
module type FOO =
  sig
    val add : int -> int -> int
  end;;

module Foo : FOO =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;;

Foo.add 3 4;;     (* OK *)
Foo.mult 3 4;;    (* not accessible *)
```

Abstract Types in Signatures

```
module type SHAPES =
  sig
    type shape
    val area : shape -> float
    val unit_circle : shape
    val make_circle : float -> shape
    val make_rect : float -> float -> shape
end;;

module Shapes : SHAPES =
  struct
    ...
    let make_circle r = Circle r
    let make_rect x y = Rect (x, y)
  end
```

- Now definition of **`shape`** is hidden

Imperative OCaml

- There are three basic operations on memory:
  - **`ref : 'a -> 'a ref`**
    - Allocate an updatable reference
  - **`! : 'a ref -> 'a`**
    - Read the value stored in reference
  - **`:= : 'a ref -> 'a -> unit`**
    - Write to a reference

```
let x = ref 3   (* x : int ref *)
let y = !x
x := 4
```

Semicolon Revisited; Side Effects

- Now that we can update memory, we have a real use for ; and () : unit
  - e1; e2 means evaluate e1, throw away the result, and then evaluate e2, and return the value of e2
  - () means “no interesting result here”
  - It’s only interesting to throw away values or use () if computation does something besides return a result
- A *side effect* is a visible state change
  - Modifying memory
  - Printing to output
  - Writing to disk

Exceptions

```
exception My_exception of int

let f n =
  if n > 0 then
    raise (My_exception n)
  else
    raise (Failure "foo")

let bar n =
  try
    f n
  with My_exception n ->
      Printf.printf "Caught %d\n" n
    | Failure s ->
      Printf.printf "Caught %s\n" s
```

Threads

Thread Creation in Java

- To explicitly create a thread:
  - Instantiate a Thread object
    - An object of class Thread *or* a subclass of Thread
  - Invoke the object's start() method
    - This will start executing the Thread's run() method concurrently with the current thread
  - Thread terminates when its run() method returns

Data Race Example

```
public class Example extends Thread {
  private static int cnt = 0;  // shared state
  public void run() {
    int y = cnt;
    cnt = y + 1;
  }
  public static void main(String args[]) {
    Thread t1 = new Example();
    Thread t2 = new Example();
    t1.start();
    t2.start();
  }
}
```

Locks (Java 1.5)

```
interface Lock {
  void lock();
  void unlock();
  ... /* Some more stuff, also */
}
class ReentrantLock implements Lock { ... }
```

- Only one thread can hold a lock at once
  - Other threads that try to acquire it *block* (or become suspended) until the lock becomes available
- *Reentrant lock* can be reacquired by same thread
  - As many times as desired
  - No other thread may acquire a lock until has been released same number of times it has been acquired

Avoiding Interference: Synchronization

```
public class Example extends Thread {
  private static int cnt = 0;
  static Lock lock = new ReentrantLock();
  public void run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
    }
  }
  ...
}
```

***Lock**, for protecting the shared state*

***Acquires** the lock; Only succeeds if not held by another thread*

***Releases** the lock*

Deadlock

- *Deadlock* occurs when no thread can run because all threads are waiting for a lock
  - No thread running, so no thread can ever release a lock to enable another thread to run

This code can deadlock…
-- when will it work?
-- when will it deadlock?

```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();

Thread 1          Thread 2

l.lock();         m.lock();
m.lock();         l.lock();
...               ...
m.unlock();       l.unlock();
l.unlock();       m.unlock();
```

Synchronized

- This pattern is really common
  - Acquire lock, do something, release lock under any circumstances after we're done
    - Even if exception was raised etc.
- Java has a language construct for this
  - `synchronized (obj) { body }`
    - Every Java object has an implicit associated lock
  - Obtains the lock associated with `obj`
  - Executes ***`body`***
  - Release lock when scope is exited
    - Even in cases of exception or method return

Example

```
static Object o = new Object();

void f() throws Exception {
  synchronized (o) {
    FileInputStream f =
      new FileInputStream("file.txt");
    // Do something with f
    f.close();
  }
}
```

- Lock associated with o acquired before body executed
  - Released even if exception thrown

Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time
- Threads can share data
  - In Java, only fields can be shared
  - Need to prevent interference
    - Rule of thumb 1: You must hold a lock when accessing shared data
    - Rule of thumb 2: You must not release a lock until shared data is in a valid state
  - Overuse use of synchronization can create deadlock
    - Rule of thumb: No deadlock if only one lock

The Dining Philosophers Problem

- Philosophers either eat or think
- They must have two forks to eat
- Can only use forks on either side of their plate
- Avoid deadlock and starvation!

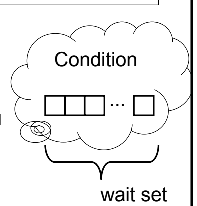
Producer/Consumer Problem

- Suppose we are communicating with a shared variable
  - E.g., some kind of a fixed size buffer holding messages
- One thread *produces* input to the buffer
- One thread *consumes* data from the buffer
- Rules:
  - producer can't add input to the buffer if it's full
  - consumer can't take input from the buffer if it's empty

Conditions (Java 1.5)

```
interface Lock { Condition newCondition(); ... }
interface Condition {
  void await();
  void signalAll(); ... }
```

- Condition created from a Lock
- await called with lock held
  - Releases the lock (on the fork or buffer)
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread



when philosopher is waiting for a fork or consumer is waiting for non empty buffer

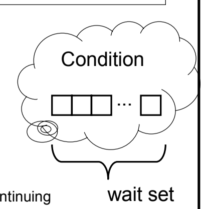
Conditions (Java 1.5)

```
interface Lock { Condition newCondition(); ... }
interface Condition {
  void await();
  void signalAll(); ... }
```

- Condition created from a Lock

when philosopher is done eating or when buffer is non empty:

- signalAll called with lock held
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don't need to do it explicitly)



Producer/Consumer Example

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean bufferReady = false;
Object buffer;
```

```
void produce(Object o) {
    lock.lock();
    while (bufferReady){
      ready.await(); }
    buffer = o;
    bufferReady = true;
    ready.signalAll();
    lock.unlock();
  }
```

```
Object consume() {
    lock.lock();
    while (!bufferReady){
      ready.await(); }
    Object o = buffer;
    bufferReady = false;
    ready.signalAll();
    lock.unlock();
  }
```

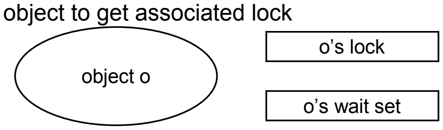
More on the Condition Interface

```
 interface Condition {
  void await();
  boolean await (long time, TimeUnit unit);
  void signal();
  void signalAll();
... }
```

- away(t, u) waits for time t and then gives up
  - Result indicates whether woken by signal or timeout
- signal() wakes up only *one* waiting thread
  - Tricky to use correctly
    - Have all waiters be equal, handle exceptions correctly
  - Highly recommended to just use signalAll()

Wait and NotifyAll (Java 1.4)

- Recall that in Java 1.4, use synchronize on object to get associated lock



- Objects also have an associated wait set

Java Generics

Subtyping

- Both inheritance and interfaces allow one class to be used where another is specified
  - This is really the same idea: subtyping
- We say that A is a *subtype* of B if
  - A extends B or a subtype of B, or
  - A implements B or a subtype of B

Parametric Polymorphism for Stack

```
class Stack<ElementType> {
  class Entry {
    ElementType elt; Entry next;
    Entry(ElementType i, Entry n) { elt = i; next = n; }
  }
  Entry theStack;
  void push(ElementType i) {
    theStack = new Entry(i, theStack);
  }
  ElementType pop() throws EmptyStackException {
    if (theStack == null)
      throw new EmptyStackException();
    else {
      ElementType i = theStack.elt;
      theStack = theStack.next;
      return i;
  }}}
```

Stack<Element> Client

```
Stack<Integer> is = new Stack<Integer>();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();
```

- No downcasts
- Type-checked at compile time
- No need to duplicate Stack code for every usage
  - line i = is.pop(); can stay the same even if the type of is isn't an integer in every path through the program

Subtyping and Arrays

- Java has one funny subtyping feature:
  - If S is a subtype of T, then
  - S[] is a subtype of T[]
- Lets us write methods that take arbitrary arrays

```
public static void reverseArray(Object [] A) {
  for(int i=0, j=A.length-1; i<j; i++,j--) {
    Object tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
  }
}
```

Problem with Subtyping Arrays

```
public class A { ... }
public class B extends A { void newMethod(); }
...
  void foo(void) {
     B[] bs = new B[3];
     A[] as;

     as = bs;            // Since B[] subtype of A[]
     as[0] = new A();    // (1)
     bs[0].newMethod();  // (2) Fails since not type B
  }
```

- Program compiles without warning
- Java must generate run-time check at (1) to prevent (2)
  - Type written to array must be subtype of array contents

Subtyping for Generics

- Is Stack<Integer> a subtype of Stack<Object>?
  - We could have the same problem as with arrays
  - Thus Java forbids this subtyping
- Now consider the following method:

```
int count(Collection<Object> c) {
  int j = 0;
  for (Iterator<Object> i = c.iterator(); i.hasNext(); ) {
    Object e = i.next(); j++;
  }
  return j;
}
```

  - Not allowed to call count(x) where x has type Stack<Integer>

Bounded Wildcards

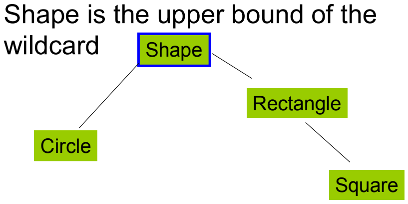
- We want drawAll to take a Collection of anything that is a *subtype* of shape
  - this includes Shape itself

```
void drawAll(Collection<? extends Shape> c) {
  for (Shape s : c)
    s.draw();
}
```

  - This is a *bounded wildcard*
  - We can pass Collection<Circle>
  - We can safely treat e as a Shape

Upper Bounded Wild Cards

- ? extends Shape actually gives an *upper bound* on the type accepted
- Shape is the upper bound of the wildcard



Bounded Wildcards (cont'd)

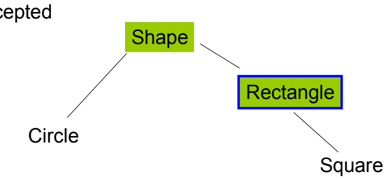
- Should the following be allowed?

```
void foo(Collection<? extends Shape> c) {
  c.add(new Circle());
}
```

  - No, because c might be a Collection of something that is not compatible with Circle
  - This code is forbidden at compile time

Lower Bounded Wildcards

- Dual of the upper bounded wildcards
- ? super Rectangle denotes a type that is a supertype of Rectangle
  - T is included
- ? super Rectangle gives a lower bound on the type accepted



Garbage Collection

Memory attributes

- Memory to store data in programming languages has several attributes:
  - Persistence (or lifetime) – How long the memory exists
  - Allocation – When the memory is available for use
  - Recovery – When the system recovers the memory for reuse
- Most programming languages are concerned with some subset of the following 4 memory classes:
  - Fixed (or static) memory
  - Automatic memory
  - Programmer allocated memory
  - Persistent memory

Memory classes

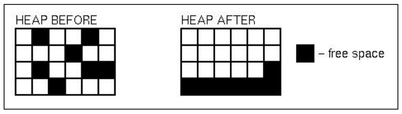
- Static memory – Usually a fixed address in memory
  - Persistence – Lifetime of execution of program
  - Allocation – By compiler for entire execution
  - Recovery – By system when program terminates
- Automatic memory – Usually on a stack
  - Persistence – Lifetime of method using that data
  - Allocation – When method is invoked
  - Recovery – When method terminates

Memory classes

- Allocated memory – Usually memory on a heap
  - Persistence – As long as memory is needed
  - Allocation – Explicitly by programmer
  - Recovery – Either by programmer or automatically (when possible and depends upon language)
- Persistent memory – Usually the file system
  - Persistence – Multiple execution of a program (e.g., files or databases)
  - Allocation – By program or user, often outside of program execution
  - Recovery – When data no longer needed
  - This form of memory usually outside of programming language course and part of database area (e.g., CMSC 424)

Garbage collection goal

- Process to reclaim memory. (Also solve Fragmentation problem.)



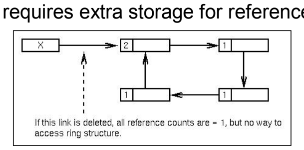
- Algorithm: You can do garbage collection and memory compaction if you know where every pointer is in a program. If you move the allocated storage, simply change the pointer to it.
- This is true in LISP, OCAML, Java, Prolog
- Not true in C, C++, Pascal, Ada

Reference Counting

- Old technique (1960)
- Each object has count of number of pointers to it from other objects and from the stack
  - When count reaches 0, object can be deallocated
- Counts tracked by either compiler or manually
- To find pointers, need to know layout of objects
  - In particular, need to distinguish pointers from ints
- Method works mostly for reclaiming memory;

Tradeoffs

- Advantage: incremental technique
  - Generally small, constant amount of work per memory write
  - With more effort, can even bound running time
- Disadvantages:
  - Cascading decrements can be expensive
  - Can't collect cycles, since counts never go to 0
  - Also requires extra storage for reference counts



Mark and Sweep GC

- Idea: Only objects reachable from stack could possibly be live
  - Every so often, stop the world and do GC:
    - Mark all objects on stack as live
    - Until no more reachable objects,
      - Mark object reachable from live object as live
    - Deallocate any non-reachable objects
- This is a *tracing* garbage collector
- Does not handle fragmentation problem

Tradeoffs with Mark and Sweep

- Pros:
  - No problem with cycles
  - Memory writes have no cost
- Cons:
  - Fragmentation
    - Available space broken up into many small pieces
      - Thus many mark-and-sweep systems may also have a *compaction* phase (like defragmenting your disk)
  - Cost proportional to heap size
    - Sweep phase needs to traverse whole heap – it touches dead memory to put it back on to the free list
  - Not appropriate for real-time applications
    - You wouldn't like your auto's braking system to stop working for a GC while you are trying to stop at a busy intersection

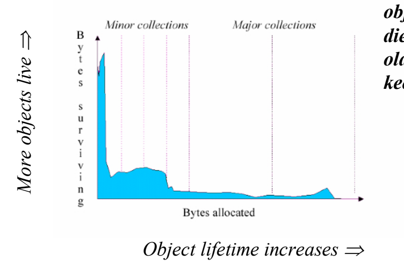
Stop and Copy GC

- Like mark and sweep, but only touches live objects
  - Divide heap into two equal parts (semispaces)
  - Only one semispace active at a time
  - At GC time, flip semispaces
    - Trace the live data starting from the stack
    - Copy live data into other semispace
    - Declare everything in current semispace dead; switch to other semispace

Stop and Copy Tradeoffs

- Pros:
  - Only touches live data
  - No fragmentation; automatically compacts
    - Will probably increase locality
- Cons:
  - Requires twice the memory space
  - Like mark and sweep, need to “stop the world”
    - Program must stop running to let garbage collector move around data in the heap

The Generational Principle



“Young objects die quickly; old objects keep living”

Errors and Exceptions

Signaling Errors

- Style 1: Return invalid value

```
// Returns value key maps to, or null if no
// such key in map
Object get(Object key);
```

  - Disadvantages?

Signaling Errors (cont’d)

- Style 2: Return an invalid value and status

```
static int lock_rdev(mdk_rdev_t *rdev) {
  ...
  if (bdev == NULL)
    return -ENOMEM;
  ...
}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char *path, const char *mode);
```

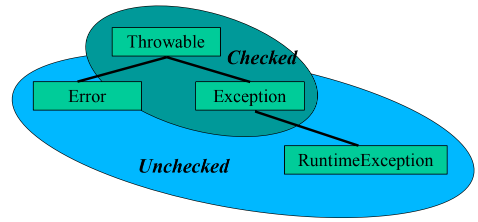
Problems with These Approaches

- What if all possible return values are valid?
  - E.g., `findMax` from earlier slide
  - What about errors in a constructor?
- What if client forgets to check for error?
  - No compiler support
- What if client can’t handle error?
  - Needs to be dealt with at a higher level
- Poor modularity- exception handling code becomes scattered throughout program
- 1996 Ariane 5 failure classic example of this …

Better approaches: Exceptions in Java

- On an error condition, we *throw* an exception
- At some point up the call chain, the exception is *caught* and the error is handled
- Separates normal from error-handling code
- A form of non-local control-flow
  - Like goto, but structured

Exception Hierarchy



Unchecked Exceptions

- Subclasses of RuntimeException and Error are unchecked
  - Need not be listed in method specifications
- Currently used for things like
  - NullPointerException
  - IndexOutOfBoundsException
  - VirtualMachineError
- Is this a good design?

Call-by-Value

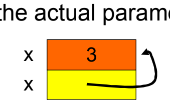
- In *call-by-value* (*cbv*), arguments to functions are fully evaluated before the function is invoked
  - Also in OCaml, in let x = e1 in e2, the expression e1 is fully evaluated before e2 is evaluated
- C, C++, and Java also use call-by-value

```
int r = 0;
int add(int x, int y) { return r + x + y; }
int set_r(void) {
  r = 3;
  return 1;
}
add(set_r(), 2);
```

Call-by-Reference

- Alternative idea: Implicitly pass a *pointer* or *reference* to the actual parameter
  - If the function writes to it the actual parameter is

```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```



Call-by-Name

- *Call-by-name* (*cbn*)
  - First described in description of Algol (1960)
  - Generalization of Lambda expressions (to be discussed later)
  - Idea simple: In a function:
    Let add x y = x+y
    add (a*b) (c*d)
    Then each use of x and y in the function definition is just a literal substitution of the actual arguments, (a*b) and (c*d), respectively
  - But implementation: Highly complex, inefficient, and provides little improvement over other mechanisms, as later slides demonstrate

Example:
add (a*b) (c*d) =
(a*b) + (c*d) ← executed function

Three-Way Comparison

- Consider the following program under the three calling conventions
  - For each, determine i's value and which a[i] (if any) is modified

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

Example: Call-by-Value

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]	f	g
1	0	1	2		
				1	1
				5	2

Example: Call-by-Reference

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i/g	a[0]	a[1]/f	a[2]
1	0	1	2
2		10	
2		10	

Example: Call-by-Name

```
int i = 1;

void p(int f, int g) {
  g++;          } i++;
  f = 5 * i;    } a[i] = 5*i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]
1	0	1	2
2			10
2			10

The expression a[i] isn't evaluated until needed, in this case after i has changed.

Call-by-Name and Exam Questions

- Even though the example we just showed suggests call-by-name and side effects can be made to work together, they just don't make sense
- We will *not* ask you any exam questions where you need to explain what call-by-name would do in a language with side effects
  - Answering these questions usually requires a great deal of specification, including deciding whether variable bindings evaluate their arguments, and the order of evaluation of function calls
  - They're just not good questions

Tail Recursion

- Recall that in OCaml, all looping is via recursion
  - Seems very inefficient
  - Needs one stack frame for recursive call
- A function is *tail recursive* if it is recursive and the recursive call is a tail call

Names and Binding

- Programs use *names* to refer to things
  - E.g., in x = x + 1, x refers to a variable
- A *binding* is an association between a name and what it refers to

```
– int x;                   /* x is bound to a stack
                              location containing an
                              int */
– int f (int) { ... }      /* f is bound to a
                              function */
– class C { ... }          /* C is bound to a class */
– let x = e1 in e2         (* x is bound to e1 *)
```

Free and Bound Variables

- The *bound variables* of a scope are those names that are declared in it
- If a variable is not bound in a scope, it is *free*
  - The bindings of variables which are free in a scope are "inherited" from declarations of those variables in outer scopes in static scoping

```
{ /* 1 */
  int j;          <- j is bound in scope 1
  { /* 2 */
    float i;      <- i is bound in scope 2
    j = (int) i;  <- j is free in scope 2
  }
}
```

Static vs. Dynamic Scope

Static scoping
- Local understanding of function behavior
- Know at compile-time what each name refers to
- A bit trickier to implement

Dynamic scoping
- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (just keep a global table of stacks of variable/value

Semantics

Operational Semantics Rules

n → *n*

true → *true*

false → *false*

[] → *[]*

- Each basic entity evaluates to the corresponding value

Operational Semantics Rules (cont'd)

- How about built-in functions?

  (+) n m → *n + m*

  - We're applying the + function
    - (we put parens around it because it's not in infix notation; will skip this from now on)
    - Ignore currying for the moment, and pretend we have multi-argument functions
  - On the right-hand side, we're computing the mathematical sum; the left-hand side is source code
  - But what about + (+ 3 4) 5 ?
    - We need recursion

Rules with Hypotheses

- To evaluate $+ E_1 E_2$, we need to evaluate E_1, then evaluate E_2, then add the results
  - This is call-by-value

$$\frac{E_1 \rightarrow n \qquad E_2 \rightarrow m}{+ E_1 E_2 \rightarrow n + m}$$

  - This is a "natural deduction" style rule
  - It says that if the *hypotheses* above the line hold, then the *conclusion* below the line holds
    - i.e., if E_1 executes to value n and if E_2 executes to value m, then $+ E_1 E_2$ executes to value $n+m$

Error Cases

$$\frac{E_1 \rightarrow n \qquad E_2 \rightarrow m}{+ E_1 E_2 \rightarrow n + m}$$

- Because we wrote n, m in the hypothesis, we mean that they must be integers
- But what if E_1 and E_2 aren't integers?
  - E.g., what if we write + false true ?
  - It can be parsed, but we can't execute it
- We will have no rule that covers such a case
  - Convention: If there is not rule to cover a case, then the expression is erroneous
  - A program that evaluates to a stuck expression produces a run time error in practice

Trees of Semantic Rules

- When we apply rules to an expression, we actually get a tree
  - Corresponds to the recursive evaluation procedure
    - For example: + (+ 3 4) 5

$$\frac{\dfrac{3 \rightarrow 3 \qquad 4 \rightarrow 4}{(+\ 3\ 4) \rightarrow 7} \qquad 5 \rightarrow 5}{+\ (\ +\ 3\ 4)\ 5 \rightarrow 12}$$

Semantics with Environments

- Extend rules to the form $A; E \rightarrow v$
  - Means in environment A, the program text E evaluates to v
- Notation:
  - We write • for the empty environment
  - We write A(x) for the value that x maps to in A
  - We write A, x:v for the same environment as A, except x is now v
    - x might or might not have mapped to anything in A
  - We write A, A' for the environment with the bindings of A' added to and overriding the bindings of A
  - The empty environment can be omitted when things are clear, and in adding other bindings to an empty environment we can write just those bindings if things are clear

Lambda Calculus

Lambda Calculus

- A lambda calculus expression is defined as

e ::= x	variable
| λx.e	function
| e e	function application

- λx.e is like `(fun x -> e)` in OCaml
- That's it! Only higher-order functions

Three Conveniences

- Syntactic sugar for local declarations
  - let x = e1 in e2 is short for (λx.e2) e1
- The scope of λ extends as far to the right as possible
  - λx. λy.x y is λx.(λy.(x y))
- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml

Operational Semantics

- All we've got are functions, so all we can do is call them
- To evaluate (λx.e1) e2
  - Evaluate e1 with x bound to e2
- This application is called "beta-reduction"
  - (λx.e1) e2 → e1[x/e2] (the eating rule)
    - e1[x/e2] is e1 where occurrences of x are replaced by e2
    - Slightly different than the environments we saw for Ocaml
      - Do substitutions to replace formals with actuals, instead of carrying around environment that maps formals to actuals
  - We allow reductions to occur anywhere in a term

Static Scoping and Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - The rightmost "x" refers to the second binding
  - This is a function that takes its argument and applies it to the identity function
- This function is "the same" as (λx.x (λy.y))
  - Renaming bound variables consistently is allowed
    - This is called *alpha-renaming* or *alpha conversion* (color rule)
  - Ex. λx.x = λy.y = λz.z   λy.λx.y = λz.λx.z

Beta-Reduction, Again

- Whenever we do a step of beta reduction...
  - (λx.e1) e2 → e1[x/e2]
  - ...alpha-convert variables as necessary
- Examples:
  - (λx.x (λx.x)) z = (λx.x (λy.y)) z → z (λy.y)
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z

Booleans

The lambda calculus was created by logician Alonzo Church in the 1930's to formulate a mathematical logical system

true = λx.λy.x

false = λx.λy.y

if a then b else c is defined to be the λ expression: a b c

- Examples:
  - if true then b else c → (λx.λy.x) b c → (λy.b) c → b
  - if false then b else c → (λx.λy.y) b c → (λy.y) c → c

Pairs

(a,b) = λx.if x then a else b

fst = λf.f true

snd = λf.f false

- Examples:
  - fst (a,b) = (λf.f true) (λx.if x then a else b) →
    (λx.if x then a else b) true →
    if true then a else b → a
  - snd (a,b) = (λf.f false) (λx.if x then a else b) →
    (λx.if x then a else b) false →
    if false then a else b → b

Natural Numbers (Church*)

*(Named after Alonzo Church, developer of lambda calculus)

0 = λf.λy.y
1 = λf.λy.f y
2 = λf.λy.f (f y)
3 = λf.λy.f (f (f y))
i.e., n = λf.λy.<apply f n times to y>

succ = λz.λf.λy.f (z f y)
iszero = λg.g (λy.false) true

- Recall that this is equivalent to λg.((g (λy.false)) true)

Natural Numbers (cont'd)

- Examples:

succ 0 =
(λz.λf.λy.f (z f y)) (λf.λy.y) →
λf.λy.f ((λf.λy.y) f y) →
λf.λy.f y = 1

iszero 0 =
(λz.z (λy.false) true) (λf.λy.y) →
(λf.λy.y) (λy.false) true →
(λy.y) true →
true

Arithmetic defined

- Addition, if M and N are integers (as λ expressions):
  M + N = λx.λy.(M x)((N x) y)
  Equivalently: + = λM.λN.λx.λy.(M x)((N x) y)
- Multiplication: M * N = λx.(M (N x))
- Prove 1+1 = 2.
  1+1 = λx.λy.(1 x)((1 x) y) →
  λx.λy.((λx.λy.x y) x)(((λx.λy.x y) x) y) →
  λx.λy.(λy.x y)(((λx.λy.x y) x) y) →
  λx.λy.(λy.x y)((λy.x y) y) →
  λx.λy.x ((λy.x y) y) →
  λx.λy.x (x y) = 2
- With these definitions, can build a theory of integer arithmetic.

The "Paradoxical" Combinator

Y = λf.(λx.f (x x)) (λx.f (x x))

- Then
  Y F =
  (λf.(λx.f (x x)) (λx.f (x x))) F →
  (λx.F (x x)) (λx.F (x x)) →
  F ((λx.F (x x)) (λx.F (x x)))
  = F (Y F)

- Thus Y F = F (Y F) = F (F (Y F)) = ...

Example

fact = λf. λn.if n = 0 then 1 else n * (f (n-1))

- The second argument to fact is the integer
- The first argument is the function to call in the body
  - We'll use Y to make this recursively call fact

(Y fact) 1 = (fact (Y fact)) 1
→ if 1 = 0 then 1 else 1 * ((Y fact) 0)
→ 1 * ((Y fact) 0)
→ 1 * (fact (Y fact) 0)
→ 1 * (if 0 = 0 then 1 else 0 * ((Y fact) (-1))
→ 1 * 1 → 1

Simply-Typed Lambda Calculus

- e ::= n | x | λx:t.e | e e
  - We've added integers n as primitives
    - Without at least two disinct types (integer and function), can't have any type errors
  - Functions now include the type of their argument
- t ::= int | t → t
  - int is the type of integers
  - t1 → t2 is the type of a function that takes arguments of type t1 and returns a result of type t2
  - t1 is the *domain* and t2 is the *range*
  - Notice this is a recursive definition, so that we can give types to higher-order functions

Type Judgments

- We will construct a type system that proves *judgments* of the form

$$A \vdash e : t$$

  – "In type environment A, expression e has type t"

- If for a program e we can prove that it has some type, then the program type checks
  – Otherwise the program has a type error, and we'll reject the program as bad

Type Environments

- A *type environment* is a map from variables names to their types
  – Just like in our operational semantics for Scheme
- • is the empty type environment
- A, x:t is just like A, except x now has type t
- When we see a variable in the program, we'll look up its type in the environment

Type Rules

$e ::= n \mid x \mid \lambda x{:}t.e \mid e\ e$

$$\frac{}{A \vdash n : int} \qquad \frac{x \in A}{A \vdash x : A(x)}$$

$$\frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}$$

Example

A = + : int → int → int
B = A, x : int

$$\dfrac{\dfrac{\dfrac{\dfrac{B \vdash + : i \to i \to i \qquad B \vdash x : int}{B \vdash +\ x : int \to int} \qquad B \vdash 3 : int}{B \vdash +\ x\ 3 : int}}{A \vdash (\lambda x{:}int.+\ x\ 3) : int \to int} \qquad A \vdash 4 : int}{A \vdash (\lambda x{:}int.+\ x\ 3)\ 4 : int}$$