Function implementation
Evaluation strategies
Runtime organization

The address space of a process is divided into segments:

- code (aka text)
- global data
- heap
- stack

Details depend on the OS
An aside on bits

Recent operating systems (e.g., Windows 7, OS X 10.6) are 64-bit.

What are the advantages?

What are the disadvantages?
Call stack

Each thread in a process maintains a call stack

Stack frame (aka activation record) contains:

- function parameters
- saved frame pointer of caller
- saved return address of caller
- local variables

Stack frame pushed when calling a function
Stack frame popped on return
Stack frame

**fp** - frame pointer
points to current frame

**sp** - stack pointer
points to current top of stack

**saved ip** - saved return address

**saved fp** - saved frame pointer of caller
aka *dynamic link*

sometimes also: *static link* - pointer to frame of lexically enclosing function (this is not used by most languages nowadays)

**callee frame** can access args from **caller frame**
Call

Before call:
- caller allocates callee stack frame
- caller evaluates and stores parameters in registers or on stack
- caller stores return address in register or on stack

Prologue:
- callee stores frame pointer in stack
- callee set fp to be top of stack
- callee allocates local variables (bumping sp)
Return

Epilogue:
- callee stores return value in register or on stack
- callee restores fp
- callee jumps to return address

After return:
- caller copies return value out
- caller deallocates callee frame
```python
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
```

Call stack:

```
fp ➔
1288  saved ip  -
1284  saved fp  -
1280  x          -
1276  sp ➔
1272
1268
1264
1260
1256
1252
1248
1244
1240
```
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

⇒ f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
⇒ f(x, 1)
def g(y, z) = y + z

⇒ def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)

<table>
<thead>
<tr>
<th>Saved IP</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved FP</td>
<td>-</td>
</tr>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>Param 1</td>
<td>3</td>
</tr>
<tr>
<td>Param 2</td>
<td>1</td>
</tr>
<tr>
<td>Saved IP</td>
<td>&lt;after f(x, 1)&gt;</td>
</tr>
<tr>
<td>Saved FP</td>
<td>1244</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
</tr>
</tbody>
</table>
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    ▶ val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
```scala
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)
```

---

<table>
<thead>
<tr>
<th>Saved IP</th>
<th>Cleaned IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1288</td>
<td>-</td>
</tr>
<tr>
<td>1284</td>
<td>-</td>
</tr>
<tr>
<td>1280</td>
<td>x</td>
</tr>
<tr>
<td>1276</td>
<td>param 1</td>
</tr>
<tr>
<td>1272</td>
<td>param 2</td>
</tr>
<tr>
<td>1268</td>
<td>saved ip</td>
</tr>
<tr>
<td>1264</td>
<td>saved fp</td>
</tr>
<tr>
<td>1260</td>
<td>a</td>
</tr>
<tr>
<td>1256</td>
<td>b</td>
</tr>
<tr>
<td>1252</td>
<td>param 1</td>
</tr>
<tr>
<td>1248</td>
<td>param 2</td>
</tr>
<tr>
<td>1244</td>
<td>saved ip</td>
</tr>
<tr>
<td>1240</td>
<td>saved fp</td>
</tr>
</tbody>
</table>

---

Tuesday, February 9, 2010
Call stack

\[ \text{def } g(y, z) = y + z \]

\[ \text{def } f(x, y) = \{
\text{val } a = 4 \\
\text{val } b = g(x+a, 2) \\
\text{b + y}
\}\]

\[ \text{val } x = 3 \]

\[ f(x, 1) \]
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    ➔ val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
Recursive functions very common in functional languages

Tail calls:
- call is *last* operation before return
- Can optimize:
  - before pushing new stack frame, pop the current frame
  - avoids explosive stack growth
  - essential in functional languages since they use lots of recursion
def fact(n) = {
    if (n <= 1) 1
    else fact(n-1)*n
}
def fact(n) = {
    if (n <= 1) 1
    else fact(n-1)*n
}

No.

after return from fact(n-1),
result is multiplied with n
def fact(n) = {
    if (n <= 1) 1
    else n*fact(n-1)
}

def fact(n) = {
  if (n <= 1) 1
  else n*fact(n-1)
}

No.

after return from fact(n-1),
result is multiplied with n
def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

Yes.
def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

Compiler can rewrite as a loop:

def fact2(n, acc) = {
    while (n > 1) {
        n = n-1
        acc = acc*n
    }
    acc
}
def even(n) = {
    if (n == 0) true
    else odd(n-1)
}

def odd(n) = {
    if (n == 1) false
    else even(n-1)
}
Tail call optimization

Compiler can optimize tail calls as follows:

- first pop the current stack frame
- push the arguments to the call
- push the return address (== the return address of the caller)
- make the call

If making a tail call to the same function, can just overwrite the arguments and jump to the function entrypoint
Without TCO

def fact(n) = fact2(n, 1)
def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}
def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
  if (n <= 1) acc
  else fact2(n-1, acc*n)
}

<table>
<thead>
<tr>
<th></th>
<th>param 1 (n)</th>
<th>param 2 (acc)</th>
<th>saved ip</th>
<th>saved fp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1288</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1284</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1276</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1272</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1268</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1264</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1260</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1252</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1248</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1244</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
    if (n <= 1) acc
    else fact2(n-1, acc*n)
}

<table>
<thead>
<tr>
<th>param 1 (n)</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>param 2 (acc)</td>
<td>3</td>
</tr>
<tr>
<td>saved ip</td>
<td>-</td>
</tr>
<tr>
<td>saved fp</td>
<td>-</td>
</tr>
</tbody>
</table>

Tuesday, February 9, 2010
def fact(n) = fact2(n, 1)

def fact2(n, acc) = {
  if (n <= 1) acc
  else fact2(n-1, acc*n)
}

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1288</td>
<td>param 1 (n)</td>
<td>1</td>
</tr>
<tr>
<td>1284</td>
<td>param 2 (acc)</td>
<td>6</td>
</tr>
<tr>
<td>1280</td>
<td>saved ip</td>
<td>-</td>
</tr>
<tr>
<td>1276</td>
<td>saved fp</td>
<td>-</td>
</tr>
<tr>
<td>1272</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1268</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1264</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1252</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1244</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TCO is done in nearly all functional language implementations
CLR (.NET) supports TCO
JVM does not do TCO
First-class functions

Scala
- (x:Int) => x+1
- (x:Int, y:Int) => x+y
- (x:Int) => (y:Int) => x+y

Python
- lambda x: x+1
- lambda x,y: x+y
- lambda x: lambda y: x+y

Scheme
- (lambda (x) (+ x 1))
- (lambda (x y) (+ x y))
- (lambda (x) (lambda (y) (+ x y)))

OCaml
- fun x -> x+1
- fun (x, y) -> x+y
- fun (x) -> fn(y) -> x+y

aka anonymous functions
aka closures
aka higher-order functions
Variables are either **bound** or **free** in a given expression

\[ x \quad \text{x is free} \]

\[ \text{fun } x \rightarrow x \quad \text{x is bound} \]

\[ \text{fun } x \rightarrow x + y \quad \text{x is bound, y is free} \]

A function can **capture** a free variable in its scope

\[ \text{fun } x \rightarrow x + y \quad \text{captures y, but not x} \]

\[ \text{fun } y \rightarrow \text{fun } x \rightarrow x + y \quad \text{does not capture y or x} \]
Aka “lambda”, “anonymous function”

Object that represents a function at run time

- first-class
- does not necessarily have a name
  - `fun x -> x + 1`
- but can be bound to a name
  - `let add1 = fun x -> x + 1`

Closures:
- *capture* free variables in their context
- closure is represented as a pair: (environment, code)
Closure representation

let x = 1
let y = 2
fun a -> a*x + y

Environment:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Closure:

env
code

compiled("fun a -> a*env.x + env.y")

Note: since names refer to values, not locations, the environment is constant.
Closure implementation

Environment is just a stack frame allocated on the heap

Must be on the heap since the closure might outlive the function that created it.
Evaluation strategies
Lambda calculus

Only 3 constructs:
- variables \( x \)
- functions (abstractions) \( \lambda x. e \)
  - a function that has a single formal parameter \( x \) and a body \( e \)
- function application \( (e_1 e_2) \)
  - invokes function \( e_1 \) on argument \( e_2 \)

The only values are functions
- Can encode all other languages in the lambda calculus
Function application

Beta reduction

- \((\lambda x. e_1) e_2 \longrightarrow e_1[e_2/x]\)

Read \(e_1[e_2/x]\) as “\(e_1\) with \(e_2\) for \(x\)”
- replace free occurrences of \(x\) in \(e_1\) with \(e_2\)
- (will define this more precisely next week)

Identity function

- \(\lambda x. x\)
- \((\lambda x. x) e \longrightarrow x[e/x] = e\)
The lambda calculus does not define an evaluation strategy

- evaluation order is *nondeterministic*

---

<table>
<thead>
<tr>
<th>((\lambda x. \lambda y. + x y) (/ 6 2) (* 4 2))</th>
<th>((\lambda x. \lambda y. + x y) (/ 6 2) 8)</th>
<th>((\lambda y. + (/ 6 2) y) (* 4 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda x. \lambda y. + x y) 3 (* 4 2))</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + (/ 6 2) y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) (* 4 2))</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
<tr>
<td>((\lambda y. + 3 y) 8)</td>
<td>((\lambda x. \lambda y. + x y) 3 8)</td>
<td>((\lambda y. + 3 y) 8)</td>
</tr>
</tbody>
</table>

\(\lambda y. + 3 y\) 8

11

**Note:** since all there are no side effects, the order of evaluation does not matter
CBV vs. CBN

In lambda calculus: two strategies

- **call-by-value**
  - evaluate function arguments before making a call
  - aka *eager evaluation*
  - aka *strict evaluation*

- **call-by-name**
  - evaluate function call before evaluating arguments to the call
  - aka *lazy evaluation*
In practice

Most languages support call-by-value

Some languages also support call-by-name

- Scala
- Macros (actually call-by-denotation)
- Haskell (actually call-by-need)
#define F(a) (a*a)
int x = 0;
int y = F(++x);
printf("%d %d", x, y); // prints 2 4, not 1 1

F(++x) expands to (++x) * (++x)

Actually call-by-denotation
- parameters are textually replaced
- F(a+b) expands to a+b*a+b == a+(b*a)+b
  != (a+b)*(a+b)

C macros are unhygienic.
Call-by-name in Scala

Can declare a call-by-name formal parameter using type => T

```scala
def repeat(n: Int)(body: => Unit) = {
  for (i <- 1 to n) {
    body
  }
}

repeat (10) {
  println("hello");
}

Can build your own control structures!
```
def withLock(body: => T): T {
    try {
        theLock.lock
        body
    }
    finally {
        theLock.unlock
    }
}

withLock { value = value + 1 } // atomic increment
def debug(msg: => String) = {
  if (debugEnabled)
    println(msg);
}

if debugEnabled is false, debug(bigArray.toString) will not call toString, which might be very expensive
def assert(test: => Boolean) = {
  if (assertionsEnabled)
    if (! test)
      throw new AssertionError();
}

if assertionsEnabled is false,
assert(x > 0) will not evaluate the test
An aside about assertions

In Java, assertions are guarded by a command-line option:

    java -ea MainClass

runs MainClass with assertions enabled

If assertions are not enabled the assert statement has no effect--the run-time compiler doesn’t generate any code for them

Useful idiom:

```java
boolean assertions = false;
assert assertions = true;
/* assertions == true only if running with -ea */
```
Call-by-need

Languages with lazy evaluation implement *call-by-need*
- Semantics is the same as call-by-name, except each expression is evaluated at most once
- Example: Haskell

Summary:

<table>
<thead>
<tr>
<th></th>
<th>(λ x. + x x) E</th>
<th>(λ x. 0) E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call-by-value</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Call-by-name</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Call-by-need</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Call-by-need implementation

Represent each expression as a *thunk*
- a closure that takes no arguments and *memoizes* its result.

Expression $e$ implemented as:

```javascript
var result = null;
() => { if (result == null) result = e; result }
```

Variable reference $x$ implemented as:

```javascript
x()
```
Call-by-need implementation

Optimization:

When possible Haskell compiler will eagerly evaluate code as long as its behavior is the same as call-by-need.

This eliminates much of the overhead of thunking.
Call-by-reference

Really orthogonal to CBV vs. CBN.
In most languages, call-by-reference is just call-by-value with a value that’s an address

```c
int x;
f(&x);    // passes address of stack variable x
         // letting f write until the caller’s
         // frame
```

In C, C++, can take address of variable on the stack.
In Java, C#, most other safe languages, cannot—all references are to objects in the heap.
Interpreter

Fun.scala

LazyFun.scala
Questions?