functions, scope & closures

Daniel Jackson
functions as values
making functions

function expression
› **function** (args) {body}

functions are ‘polymorphic’
› implicit typed
› depends on how args used

> three = function () {return 3;}
function () {return 3;}
> three
function () {return 3;}
> three()
3
> id = function (x) {return x;}
function (x) {return x;}
> id(3)
3
> id(true)
true
> id(id)
function (x) {return x;}
> (id(id))(3)
3
functions are first class

just like other objects
› can bind to variables
› can put in property slots
› can add property slots

```javascript
> seq = function () {
  seq.c += 1; return seq.c;
}
function () {seq.c += 1; return seq.c;}
> seq.c = 0
0
> seq()
1
> seq()
2

> seq = function () {return (seq.c = seq.next(seq.c));}
function () {return (seq.c = seq.next(seq.c));}
> seq.c = 0
0
> seq.next = function (i) {return i + 2;}
function (i) {return i + 2;}
> seq()
2
> seq()
4
```

note: bad lack of encapsulation! will fix later with closures
can you explain
› how a recursive definition works?
› when exactly is the function defined?

> fact = function (i) {if (i===0) return 1; else return i * fact(i-1);}
function (i) {if (i===0) return 1; else return i * fact(i-1);} 
> fact (4)
24
a puzzle: repeated applications

suppose you see an expression e
› eg, e is \texttt{f()}
› what might expression do?

evaluation can have 3 effects
› value is returned (or exception thrown)
› objects are modified
› environment is updated

a puzzle
› declare \texttt{f} so that \texttt{f()==f()} evals to false
evaluating functions
two phases

creation
› function expression evaluated

application
› function body evaluated

evaluation order for applications
› first evaluate arguments, left to right
› then evaluate body

> log = function (s) {console.log(s + seq());}
> (function () {log('c')}) (log('a'),log('b'))
a1
b2
c3
evaluating the body

what environment is body evaluated in?
 › same environment application is evaluated in?

let’s see!
 › hmm...

```javascript
> x = 1
1
> f = (function (x) {return function () {return x;};}) (x)
function () {return x;}
> f()
1
> x = 2
2
> f()
1
```
two environments

when function is created
› keeps environment as a property
› called ‘function scope’
› uses this environment to evaluate body in

what about arguments?
› new environment (‘frame’) with bindings for args
› linked to function scope
an object model

- activation distinction from (syntactic) statement
- underscores emphasize: not real properties

body of function is evaluated in context of function's scope

scope of a function is context it was created in
aah, nostalgia!

Figure 3.11
Sqrt procedure with internal definitions.

expression (sqrt 2) where the internal procedure good-enough? has been called for the first time with guess equal to 1.

Observe the structure of the environment. Sqrt is a symbol in the

Courtesy of Harold Abelson and Gerald Jay Sussman. Used with permission.
examples
example 1

what happens here?

› function scope is top-level environment
› assignment to x modifies binding in top-level environment
› so in this case x refers to x of application environment too
simulating example 1
example 2

what happens here?
› function scope is top-level environment
› when application is evaluated, argument x is bound to 2
› local x said to shadow global x
simulating example 2

1. `f = function (x) {return x;}` (Activation)
   - `stmt` (Statement)
   - context
   - creates bindings
   - `function` (Env)
   - `Function`
   - `f` (Var)
   - `value`
   - `1` (Object)
   - `var`

2. `x = 1` (Activation)
   - `stmt` (Statement)
   - bindings
   - `_scope`
   - `bindings`
   - `value`
   - `1` (Object)
   - `var`

3. `y = 2` (Activation)
   - `stmt` (Statement)
   - bindings
   - `value`
   - `2` (Object)
   - `var`

4. `f(y)` (Activation)
   - `stmt` (Statement)
   - `f` (Var)
   - `value`
   - `2` (Object)
   - `var`

> f = function (x) {return x;}
> function (x) {return x;}
> x = 1
1
> y = 2
2
> f(y)
2
what happens here?
› when `f` is applied, `x` is bound to 1 in new frame
› anonymous function has scope with `x` bound to 1
› assignment to top-level `x` does not modify this scope
simulating example 3

```javascript
> x = 1
1
> f = (function (x) {
return function () {
return x;
};
}) (x)

function () {
return x;
}
> x = 2
2
> f()
1
```
what if we modify x?
> when f is applied, x is bound to 0 in new frame
> anonymous function has scope with x bound to 0
> this ‘internal’ x is updated every time f is called
f = (function (x) {
    return function () {
        x += 1; return x;
    };
}) (0)

1. f = ...
   (Activation)
   creates
   _scope

2.1 return function ...
   (Activation)
   creates
   _scope

3. f()
   (Activation)

3.1 x += 1
   (Activation)

> f = (function (x) {
    return function () {
        x += 1; return x;
    };
}) (0)

> f()

function () {
    x += 1; return x;
}

> f()

1
local variables
avoiding pollution

```javascript
> sum = function (a, s, i) {
    s = 0;
    for (i = 0; i < a.length; i += 1) s += a[i];
    return s;
}
```

```javascript
> sum([1,2,3])
6
```

```javascript
> s
ReferenceError
```

```javascript
> i
ReferenceError
```

why does this work?
argument mismatch

when arguments are
› missing: initialized to undefined
› extra: ignored

```javascript
> inc = function (x, y) {return y ? x+y : x+1;}
function (x, y) {return y ? x+y : x+1;}
> inc(1)
2
> inc(1,2)
3
> inc(1,2,3)
3
```
var decls

> sum = function (a, s, i) {
    s = 0;
    for (i = 0; i < a.length; i += 1) s += a[i];
    return s;}

function...

don’t want bogus arguments
> so Javascript has a special statement
> “var x” creates a binding for x in the immediate env

> sum = function (a) {
    var s = 0;
    for (var i = 0; i < a.length; i += 1) s += a[i];
    return s;}

function...

note: doesn’t matter where var decl occurs in function even in dead code!
function declarations

function declaration syntax
› function f() {} short for var f = function(){}
› but not quite, so don’t use it!

```javascript
var f = function(){
  if (true) {
    function g() { return 1;};
  } else {
    function g() { return 2;};
  }
  var g = function() { return 3;}
  return g();
  function g(){ return 4;}
}
var result = f();

› ECMA: 2
› Safari, Chrome: 3
› Mozilla: 4
```
lexical vs dynamic scoping
a language design question

```javascript
x = 1;
g = function(){ console.log(x); x=2; }
f = function(){ var x = 3; g(); }
f();
console.log(x);
```

what does this print?

- lexical scoping: 1, 2
- dynamic scoping: 3, 1

lexical scoping now preferred
- harder to implement
- better for programmer
a common misunderstanding
lookup at activation time

```javascript
var multipliers = function makeMultipliers (max) {
    var result = [];
    for (var i = 0; i < max; i++)
        result.push (function (x) {return x * i;});
    return result;
}

> multipliers(10) [2] (5)
???
```

what's the value?

› 50, not 5

can you fix it?
summary

functions are first-class
› values created by expressions
› bound to variables
› stored as properties, and can have properties

lexical closures
› free variables bound in ‘declaration’ environment

local vars
› added to local environment, just like function args

next
› exploiting functions & closures in programming
6.170 Software Studio
Spring 2013

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.