functions, scope & closures

Daniel Jackson
making functions

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

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

```javascript
> three = function () {return 3;}
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
```

```javascript
> 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
```
two phases

creation
› function expression evaluated

application
› function body evaluated

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

> (function (x) {return x + 1;}) (3)
4

> log = function (s) {console.log(s + seq());}
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
example 1

> f = function () {return x;}
function () {return x;}
> x = 1
1
> f()
1
> x = 2
2
> f()
2

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

\[
\begin{align*}
\text{\(f = \text{function} () \{ \text{return} \ x; \}\) (Statement)} \\
\text{1. \(f = \text{function} () \{ \text{return} \ x; \}\) (Activation)} \\
\text{2. \(x = 1\) (Activation)} \\
\text{3. \(f()\) (Activation)} \\
\text{4. \(x = 2\) (Activation)} \\
\text{5. \(f()\) (Activation)} \\
\text{5.1 \(\text{return} \ x;\) (Activation)}
\end{align*}
\]

> \(f = \text{function} () \{ \text{return} \ x; \}\)
> \(\text{function} () \{ \text{return} \ x; \}\)
> \(x = 1\)
> 1
> \(f()\)
> 1
> \(x = 2\)
> 2
> \(f()\)
> 2
example 2

```javascript
> f = function (x) {return x;}
function (x) {return x;}
> x = 1
1
> y = 2
2
> f(y)
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
   - (Function)
   - (Env)
   - (Binding)
   - var
   - 1 (Object)

2. `x = 1`
   - Activation
   - (Binding)
   - var
   - 1 (Object)

3. `y = 2`
   - Activation
   - (Binding)
   - var
   - 2 (Object)

4. `f(y)`
   - Activation
   - (Function)
   - (Env)
   - (Binding)
   - var
   - x (Var)
   - 2 (Object)
   - _proto
   - x (Var)
   - 2 (Object)

JavaScript Code Snippet:

```javascript
> f = function (x) {return x;}
> function (x) {return x;}
> x = 1
1
> y = 2
2
> f(y)
2
```
example 3

> x = 1
1
> f = (function (x) {return function () {return x;};}) (x)
function () {return x;}
> f()
1
> x = 2
2
> f()
1

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

f = (function (x) {
  return function () {
    return x;
  };
}) (x)

1. x = 1
   (Activation)

2. f = ...
   (Activation)

2.1 return function ...
   (Activation)

3. x = 2
   (Activation)

4. f()
   (Activation)

4.1 return x
   (Activation)

> x = 1
1
> f = (function (x) {
    return function () {
      return x;
    };
}) (x)
function () {
  return x;
}
> x = 2
2
> f()
1
example 4

```javascript
> f = (function (x) {return function () {x += 1; return x;};})(0)
function () {x += 1; return x;}
> f()
1
> f()
2
```

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
simulating example 4

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

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

f();
```

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

> f()
1
avoiding pollution

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

function...

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

what’s wrong with this function?
how to fix it?

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**

```javascript
> sum = function (a) {
    var s = 0;
    for (var i = 0; i < a.length; i += 1) s += a[i];
    return s;
}
function...
> sum([1,2,3])
6
> s
ReferenceError
```

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

a strange thing
> doesn’t matter where var decl occurs in function
> even in dead code
lexical scoping

this kind of scoping is ‘lexical’
› variable bound in environment in which declared

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

x = 2;
f();
```
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
conclusions

first class functions & closures
› powerful modularity mechanism
› use to localize variables

Javascript mixed
› nice: closures, lexical scoping
› nasty: no block-level scoping, global vars