functions, scope & closures

Daniel Jackson
functions as values
making functions
making functions

function expression
› `function (args) {body}`
making functions

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

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

just like other objects
› can bind to variables
› can put in property slots
› can add property slots
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
```
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
```
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.next = function (i) {return i + 2;}
function (i) {return i + 2;}
> seq()
2
> seq()
4
```
evaluating functions
two phases
two phases

creation

› function expression evaluated
two phases

creation
› function expression evaluated

application
› function body evaluated
two phases

creation
› function expression evaluated

application
› function body evaluated

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

creation
› function expression evaluated

application
› function body evaluated

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

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

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

let’s see!
› hmm...
evaluating the body

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

let’s see!
› hmm...

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

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

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

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

**Diagram:**

- Activation
  - context
  - stmt
  - creates
  - _body

- Statement
  - _proto

- Environment
  - !_scope

- Function
  - _proto
  - ?
  - creates
  - context
  - !_scope

**Note:**

- Scope of a function is context it was created in
an object model

- activation distinction from (syntactic) statement
- underscores emphasize: not real properties
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
examples
example 1

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

> x = 2

> f()
```
example 1

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

```javascript
> 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

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

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

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

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

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

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

\[ f = \text{function} () \{ \text{return} x; \} \]

1. \[ f = \text{function} () \{ \text{return} x; \} \]

(Activation)

(Env)

(Function)

creates

bindings

2. \[ x = 1 \]

(Activation)

(Binding)

(var)

(x (Var))

1 (Object)

_value

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

\[ f = function () \{ return x; \} \]

1. `f = function () \{ return x; \}`
(Activation)

2. `x = 1`
(Activation)

3. `f()`
(Activation)

- Context creates bindings
- `f` creates `x`
- `x` is bound

\[ > f = function () \{ return x; \} \]
function () \{ return x; \}
\[ > x = 1 \]
1
\[ > f() \]
1
\[ > x = 2 \]
2
\[ > f() \]
2
simulating example 1

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

1. f = function () {return x;}
   (Activation)

2. x = 1
   (Activation)

3. f()
   (Activation)

return x;
(Statement)

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

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

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

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

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

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

\texttt{> f(y)}  \texttt{\rightarrow 2}
Simulating example 2

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

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

> x = 1
x

1

> y = 2
y

2

> f(y)
f(y)

2
```
simulating example 2

1. \( f = \text{function}\ (x) \ {\text{return}}\ x; \)
2. \( x = 1 \)
3. \( y = 2 \)

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

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

2. `x = 1` (Activation)

3. `y = 2` (Activation)

4. `f(y)` (Activation)

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

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

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

> x = 2
2
> f()
```
example 3

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

```javascript
> 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

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

1. \( x = 1 \) (Activation) \( \rightarrow \) (Env)

\[
\text{context} \quad \text{f = (function (x) {return function () {return x;};}}) (x)
\]

\[
\text{function () {return x;}}
\]

1

1

x = 1

x = 2

> x = 1

1

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

function () {return x;}

> x = 2

2

> f()

1
simulating example 3

1. `x = 1` (Activation)

```javascript
function () {
    return x;
}

f()
```

2. `x = 2`

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

function () {
    return x;
}

f()
```

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

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

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

```
x = 1
1
```

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

```
x = 2
2
```

```
f()
1
```
simulating example 3

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

> x = 1
1
> f = (function (x) {
    return function () {
      return x;
    }();
})(x)
function () {return x;}
> x = 2
2
> f()
simulating example 3

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

1. x = 1
2. f = ...
3. x = 2
4. f()
simulating example 3

```javascript
> 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()

> f()
```

example 4

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

function () { x += 1; return x; }
> f()
1
```
simulating example 4

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

f();  // 1
```

simulating example 4

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

> f()
1
```
simulating example 4

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

> f()
1
```
simulating example 4

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

function () { x += 1; return x; }
> f()
1
```
simulating example 4

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

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

> f();
1```

[Diagram showing the process of creating and executing a function]
simulating example 4

1. \( f = \) (function (x) {
   return function () {
      x += 1; return x;
   }
}(0)
(Statement)

(Env)

- creates _scope_
- context

(Function)

2.1 return function ...
(Activation)

(FEnv)

- creates _proto_
- bindings

(Binding)

- scope
- value
- var
- x (Var)
- 1 (Object)

(Function)

3. f()
(Activation)

3.1 x += 1
(Activation)

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

function (){
  x += 1; return x;
}
> f()
1
local variables
avoiding pollution
avoiding pollution

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

function...
> sum([1,2,3])
6
```
avoiding pollution

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

function...
> sum([1,2,3])
6
what’s wrong with this function?
how to fix it?
avoiding pollution

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

function...
> sum([1,2,3])
6
> s
6
> i
3

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

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

function...
> sum([1,2,3])
6
> s
ReferenceError
> i
ReferenceError
avoiding pollution

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

function...
> sum([1,2,3])
6
> s
ReferenceError
> i
ReferenceError

why does this work?
argument mismatch

when arguments are
  › missing: initialized to undefined
  › extra: ignored
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
```
var decls

sum = function (a, s, i) {
  s = 0;
  for (i = 0; i < a.length; i += 1) s += a[i];
  return s;
}
```
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
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...
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!
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();
```
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
a language design question

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

```javascript
function g () { console.log(x); x=2; }
function f () { var x = 3; g(); }
f();
console.log(x);
```
a language design question

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

what does this print?
› _lexical_ scoping: 1, 2
› _dynamic_ scoping: 3, 1
a language design question

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
activation still does lookup

```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)
```
activation still does lookup

```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 10
activation still does lookup

```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 10

can you fix it?
summary
Functions are first-class

- Values created by expressions
- Bound to variables
- Stored as properties, and can have properties
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
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
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