Language Overview

• Simple imperative language which borrows heavily from Scheme and JavaScript

• Strongly, dynamically typed

• Allows a variety of programming styles (functional, procedural, object oriented, etc)
Types

- Strings ("foo", "bar")
- Numbers (4, 3.14, 1.2e4)
- Booleans (true, false)
- Objects ({name: "Bill", age: 30})
- Vectors ([1,2,3,4])
Syntax

• C/Java/JavaScript-style syntax

```javascript
var x = 10;
var y = x + 5;
print(y);

if(y > 12) print("x > 12")

while(x++ < 20) {
    print("x: " & x)
}
```
Variables/Scoping

- Fuse has C-style block structure (note: This is NOT the same as JavaScript)
- Variables declared with the `var` keyword
  
  ```javascript
  var x = 10;
  ```
- Variables are in scope AFTER the point of declaration for the rest of the block they’re declared in
Scoping Example

// x isn’t in scope here
var x = 10;
// now x is in scope
{
    var y = 12;
    // y is in scope (and x still is)
    var x = 20;
    // x == 20 here (different x)
}
// y isn’t in scope
// x == 10 here
Functions

• Functions are first-class objects

• A function is bound to a name with the var keyword (just like other expressions)
  
  var f = lambda(x) {
    return x * 2;
  };

• Functions are called with the () “operator”

  var four = f(2);
  var eight = f(f(2));
Recursive Functions

• Remember, variable declarations are only in scope AFTER the declaration

```javascript
var fac = lambda(x) {
    if(x==0) return 1;
    return x * fac(x-1);
};
```

• This will fail to compile. The name “fac” isn’t in scope in the initializer for fac.
Recursive Declarations

- The “varrec” (like Scheme’s letrec) keyword allows for recursive (and mutually recursive) declarations

```plaintext
varrec fac = lambda(x) {
    return x==0 ? 1 : x * fac(x-1);
};

varrec
    even = lambda(n) {n==0?true:odd(n-1); },
    odd  = lambda(n) {n==0?false:even(n-1);}
```
Closures

• Functions are lexical closures

```javascript
var inc_by = lambda(x) {
  return lambda(y) {
    return x + y;
  };
};

var f = inc_by(10);
var x = f(5);
// x == 15
```
Numbers

• All numbers are double precision floating point
• Eliminates lots of confusing aspects of mixed mode arithmetic (is \(5/2\) 2 or 2.5?)
• Double precision floating point can accurately represent the entire range of 32-bit integers, so most people would never notice
Boolean Values

• There are just two boolean values, true and false

• We borrow Scheme’s conditional rules, everything that is non-false is true in a condition

• Including the 0, the empty string “”, empty list, etc.
• “Objects” are basically hashtables (the term was stolen from JavaScript)

```
var o = {name: "Bill", age: 30};
```

• Fields within the object are accessed with the dot operator

```
print(o.name); // prints Bill
```

• Fields can be added or modified after the creation of the object

```
o.title = "Manager"; o.name = "Bob";
```
Object Keys

• The keys in an object (hashtable) do not have to be constant strings

```javascript
var o = {name: “Bill”, age:30};
var field = wantAge ? “age” : “name”;  
var result = o[field];
```

• They can even be non-string values (unlike JS)

```javascript
o[2] = “two”;  
o[false] = “false”;  
o[o] = “i’m a key in myself”;
```
Object Syntactic Sugar

var o = {name: “Bill”, age: 30};

• o.name is sugar for o[“name”];

• Object literal syntax above is sugar for:

  var o = {};  
o[“name”] = “Bill”;  
o[“age”] = 30;
Vectors

• Vectors are fixed length containers accessed by index (like Java arrays or Scheme vectors)

```javascript
var v = [1, 2, 3, 4];
println(v.length); // 4
println(v[1]); // 2

var v2 = newVector(10);
println(v2.length); // 10
v2[0] = "foo";
println(v2[0]); // 10
```
Scheme Expr Mapping

- 1-to-1 mapping between Fuse and Scheme Expressions

```scheme
(lambda(x){ return x*2; })
// (lambda (x) (* x 2))
```

```scheme
f(1,2,3);
// (f 1 2 3);
```

```scheme
x == 2 ? “yep” : “nope”
// (if (= x 2) “yep” “nope”)
```
Scheme Type Mapping

- Scheme types map well to Fuse types
- Numbers, booleans, strings, and vectors have a Fuse counterpart.
- Pairs and Symbols can be written in terms of Fuse vectors or objects.
- Why not take advantage of these mappings?
Scheme Syntax

- Fuse supports scheme style syntax in addition to JS syntax

```scheme
// Note the $s
($
(define (twice x) (+ x x))
(define foo (lambda ()
    (display "Hello ")
    (display "World")
    (newline)))
$;
```
Scheme Syntax

- Scheme and JS share the same Lexical scope, so scheme bindings are accessible in JS

$$(\text{define (\texttt{twice x}) (+ x x)) (define (f) (display \text{"Hi"}) (newline))); println(twice(2)); f();$$
It even works at the expression level.

```plaintext
var twice = lambda(x) { return x*2; }
var eight = 1 + $(f 2)$ + 3;
println$(twice 10); // prints 20$

var mylist = $'("a" "scheme" "list")$;
```
Scheme Syntax

- And you can even go back to JavaScript!
  
  $(define foo $1 + 2$)$; // 1 + 2 is JS

  // argument to map is a JS lambda
  var f = $(lambda (xs) (map $lambda(x) {
    if(x == 10) return “ten”;
    else return “i don’t know”;
  }$ xs))$;
Scheme Implementation

- The scheme primitive functions (+, -, eq?, etc) can be implemented in terms of standard Fuse operators

```scheme
(define (+ x y) (+ x y))
(define (eq? x y) (= x y))
(define (vector-length v) (vector-length v))
(define (vector-ref v p) (vector-ref v p))
```

R5RS (Scheme Spec)

• The standard Fuse library provides an implementation of most of R5RS

• Notable exceptions are
  • Continuations
  • Full numeric tower (complex, real, rational, integer, exact, inexact, etc)
OOP in Fuse

- Fuse has no built-in support for OOP
- Fuse objects and first class functions allow for OOP style programming
  - Objects are Fuse “Objects” (hashtables)
  - Fields are entries in the hashtable
  - Methods are fields with a lambda value
OOP Example

```javascript
var person = lambda(n,s) {
    var this = {name: n, salary: s }; 
    this.giveRaise = lambda(percent) {
        this.salary *= (1+percent);
    };
    return this;
}
var bob = person("Bob", 50000);
bob.giveRaise(0.2);
println(bob.salary); // prints 60000
```
OOP Data Hiding

- In the previous example all fields were public
- Lexical closures can be used for data hiding

```javascript
var person = lambda(n,s) {
    var private = {name: n, salary: s};
    var public = {};
    public.giveRaise = lambda(x) {
        private.salary *= (1+x);
    }
    return public;
};
```
Fuse Value API

- Every Fuse object provides 3 primitive operations
  - GET (o.foo, v[0], etc)
  - PUT (o.foo = “bar”, v[0] = “baz”, etc)
  - CALL (o(“Hello”, “World”), o(), etc)
- Some types (numbers) also support other primitive operations (addition, etc.)
Traps

• Traps allow the user to override the GET, PUT, and CALL operations.

```javascript
var o = trap(
    lambda(k) { println("GET: ") & k; return 42; },
    lambda(k,v) { println("PUT: " & k & " " & v); },
    lambda(arg) { println("CALL: ") & arg});

println(o.foo); // prints "GET: foo" then 42
o.foo = "bar"; // prints "PUT: foo bar"
o(10); // prints "CALL: 10"
```
Uses for Traps

- Implementing more complex object systems (JavaScript’s prototypes for example)
- Implementing containers that don’t act like hashtables or fixed length vectors (JavaScript style arrays)
- Using the “dot notation” for structures with an infinite number of fields

```javascript
var dir = url("http://www.brianweb.net/");
var myfile = dir.path.to["myfile.txt"];```

XML with Traps

- Traps allow a very elegant API to XML trees

```javascript
var doc =
  "<movies>
    <movie>
      <name>V for Vendetta</name>
      <time>2:00pm</time> <time>4:00pm</time>
    </movie>
    ...
  </movies>"

var movies = xml(doc);
for(i=0;i<movies.length;i++) {
  println("Name: " & movies[i].name);
  for(j=0;i<movie[i].time.length;j++)
    println("Time: " & movies[i].time[j]);
}
```
Other Interesting Features

- Modules `(import my.module)`
- Enumerators `(for(key in obj) { ... })`
- FastString (O(log n) string concatenation)
- XMLRPC Client
Viewing Compiler Internals

- **Expression Tree** - “var z = x + 3”

```javascript
Stmt.VarDecls: {
  recursive: false
  decls: {
    Expr.Binding: {
      name: "z"
      val: Expr.BinOp: {
        op: +
        left: Expr.Ident: {
          var: "x"
        }
        right: Expr.IntegerLit: {
          i: 3
        }
      }
    }
  }
}
```
Viewing Compiler Internals

- Pretty Printer - "test ? x : test2 ? y : z + 2"

```javascript
var z = (test ? x : (test2 ? y : (z + 2)));
```
Viewing Compiler Internals

• Bytecode

// lambda(n) { var accum=1; for(n>0;n--) accum *= n; return accum; }

0: ONE: 0
1: PUTLOCAL: 1
2: POP: 0
3: GETLOCAL: 0
4: ZERO: 0
5: GT: 0
6: JF: 20
7: GETLOCAL: 1
8: GETLOCAL: 0
9: MUL: 0
10: PUTLOCAL: 1
11: POP: 0
12: GETLOCAL: 0
13: DUP: 0
14: ONE: 0
15: SUB: 0
16: PUTLOCAL: 0
17: POP: 0
18: POP: 0
19: JMP: 3
20: GETLOCAL: 1
21: RETURN: 0
Fuse as a Teaching Language

- Many (most?) students are unfamiliar with functional programming
- Fuse can be used to “ease into” FP
  - Implement an entire program in JavaScript
  - Rewrite it function by function (or even expression by expression) in Scheme
  - Stick to JS when you get stuck