

Fuse

Brian Alliet

(and some help from Paul Lorenz)

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

```
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

```
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 expressions
- 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

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

- This will fail to compile. “fac” isn’t in scope in the initializer for fac

Recursive Declarations

- The “varrec” keyword allows for recursive (and mutually recursive) declarations

```
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

```
var inc_by = lambda(x) {  
    return lambda(y) {  
        return x + y;  
    };  
};  
var f = inc_by(10);  
var x = f(5);  
// x == 15
```

Objects

- “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

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

- They can even be non-string values

```
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;
```

Scheme

- There is a 1-to-1 mapping between most Scheme/Lisp expressions and Fuse expressions

```
lambda(x){ return x*2; }  
// (lambda (x) (* x 2))  
f(1,2,3); // (f 1 2 3);
```

```
x == 2 ? “yep” : “nope”  
// (if (= x 2) “yep” “nope”)
```

Scheme (cont)

- Why not take advantage of this mapping?
- Fuse also supports scheme style syntax, and within the same file as the JavaScript style syntax

```
$(define (f x) (* x 2))$;  
print(f(2));
```

- Within the same expression even!

```
var x = 1 + $(f 2)$ * 4;
```

Scheme (cont)

- And you can even go *back* to JavaScript!

```
var f = $(lambda (xs) (map $lambda(x) {  
    if(x == 10) return "ten";  
    else return "i don't know";  
}$ list)$;
```

Scheme (cont)

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

```
$  
(define (+ x y) $x + y$)  
(define (eq? x y) $x == y$)  
(define (vector-length v) $v.length$)  
(define (vector-ref v p) $v[p]$)  
$;
```

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)

Implementation

- The fuse compiler/interpreter is broken up into several pieces
- Lexer (chars to tokens)
- Parser (tokens to AST)
- Compiler (AST to Fuse Bytecode)
- Interpreter (Evaluates Fuse Bytecode)
- Value (The runtime representation of Fuse values, strings, numbers, objects, etc)

Lexer

- Written with JLex
- Uses lexical states to implement Scheme syntax

```
<YYINITIAL> "var" { ... }
```

```
<YYINITIAL> "$" {  
    token = '$'; yybegin(SCHEME); ... }
```

```
<SCHEME> "define" { ... }
```

```
<SCHEME> "$" {  
    token = '$'; yybegin(YYINITIAL); ... }
```

Parser

- Written in Jay
- Scheme syntax implemented by rules like

```
expr: expr '+' expr  
     | ...  
     | '$' sexpr '$'
```

```
sexpr: '(' sexpr sexprs ')'  
      | ...  
      | '$' expr '$'
```

AST

- The AST built by the Parser is a giant Tree of Expr and Stmt instances

```
public class Call extends Expr {  
    public final Expr f;  
    public final Expr[] args;  
    ...  
}
```

- The visitor pattern is used to visit the tree

Fuse Bytecode

- Fuse bytecode is a bytecode language designed with three goals in mind
 - It must be simple
 - It must be easy and efficient to interpret
 - It must be easy to translate to other languages or bytecode languages
- Very similar to JVM bytecode

Fuse bytecode basics

- Stack based
- Infinite number of “registers” or local variables
- Infinite number of global variables

Example Bytecode

```
// var x = 10;
INT 10      // [10]
PUTLOCAL 0  // [10]
POP         // []
// x = 2 * (x + 20)
INT 2       // [2]
GETLOCAL 0  // [2,10]
INT 20      // [2,10,20]
ADD         // [2,30]
MUL        // [60]
PUTLOCAL 0  // [60]
POP         // []
```

Compiler

- The compiler translates Exprs and Stmnts into Fuse bytecode

```
public void push(Expr e) { ... }
public void pushCall(Expr.Call e) {
    push(e.f);
    for(int i=0;i<e.args.length;i++)
        push(e.args[i]);
    add(CALL,e.args.length);
}
```

Interpreter

- The interpreter interprets Fuse bytecodes

```
public class Interpreter {
    public Value run(Bytecode code) {
        Value[] stack = new Value[...];
        int ip = 0, sp = 0;
        for(;;) {
            var op = ..., arg = ...;
            switch(op) {
                case NEGATE:
                    stack[sp] = N(-((Number)stack[sp])); break;
                case POP: sp--;
                case ZERO: stack[++sp] = Value.ZERO; break;
                case JMP: ip = arg; continue;
            }
            ip++;
        }
    }
}
```

Value

- The Value class represents Fuse values at runtime

```
public class Value {
    public Value get(Value key) { ... }
    public void put(Value key, Value val) { ... }
    public void call(Value[] args) { ... }
    public class Hash extends Value {
        private final Hashtable ht = new Hashtable();
        public Value get(Value k) { ... }
    }
    public class Number extends Value {
        public final double d;
        ...
    }
}
```

Interesting Features

- Modules
- Enumerators
- Traps
- FastString
- XMLRPC client
- JVMCompiler

