{|Funk|}

Lambda Calculus + Pattern Matching

= Object Oriented Programming

github.com/Ahnfelt/funk

Lambda functions in **Funk** vs. JavaScript (ES5)

```
{|x y| x + y}
function(x) { return function(y) { return x + y; } }

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

{_ + 1}
function(whatever) { return whatever + 1; }

{1}
function(whatever) { return 1; }
```

The last form {...} without a parameter list is often used for delaying evaluation, which is useful when implementing your own control structures.

```
fib := {
    |0| 0
    |1| 1
    |n| fib(n - 1) + fib(n - 2)
}
```

JavaScript equivalent:

```
function fib(n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

```
color := {
    |Red| "#ff0000"
    |Green| "#00ff00"
    |Blue| "#0000ff"
}
```

JavaScript equivalent:

```
function color(c) {
    if(c == "Red") return "#ff0000";
    if(c == "Green") return "#00ff00";
    if(c == "Blue") return "#0000ff";
    throw "Illegal argument: " + c;
}
```

```
foo.Bar(1, 2, 3)
foo("Bar", 1, 2, 3)
foo Bar 1 2 3
```

The three lines above are equivalent in Funk.

It's all function application.

JavaScript equivalent:

```
foo("Bar")(1)(2)(3)
```

- No keywords
- No built-in if, while, etc.
- Bring your own control structures

```
if := {
    |True body _| body()
    |False _ body| body()
}


max := {|x y|
    if(x > y) {
        x
    } {
        y
    }
}
```

```
when := {
    |True body| body()
    |False _|
}


when(celcius > 30) {
    system.Log("It's hot!")
}
```

```
while := {|condition body|
    when(condition()) {
        body()
        while(condition, body)
    }
}


x := 10
while {x > 0} {
    system.Log(x)
    x -= 1
}
```

- Object Oriented Programming
- Without objects

```
vector := {|x y| {
    |X| x
    |Y| y
    |Add v| vector(x + v.X, y + v.Y)
}}

v1 := vector(2, 3)
v2 := vector(4, 5)
v3 := v1.Add(v2)
```

- "Method invocation" is function application:
  - v1.Add(v2)   ==   v1("Add", v2)   ==   v1 "Add" v2

- Define your own operators

```
vector := {|x y| {
    |X| x
    |Y| y
    |"+" v| vector(x + v.X, y + v.Y)
}}

v1 := vector(2, 3)
v2 := vector(4, 5)
v3 := v1 + v2
```

- Operators is function application:
  - v1 + v2   ==   v1("+", v2)   ==   v1 "+" v2

- "Inheritance" is function application

```
newMonster := {|name hitpoints| {
    |Name| name
    |Hurt damage| hitpoints -= damage
}}

newCreeper := {
    super := newMonster("Creeper", 80)
    {
        |Explode area|
            area.NearbyMonsters.Each {|monster|
                monster.Hurt(50)
            }
        |otherMethod|
            super(otherMethod)
    }
}
```

- Make your own "this pointer"

```
newMonster := {|hitpoints|
    self := {
        |Hurt damage| hitpoints -= damage
        |Die| self.Hurt(hitpoints)
    }
}
```

- Variable definitions return the value of the right hand side.

```
area.NearbyMonsters.Each {|monster|
    monster.Hurt(50)
}
```

May also be written as:

```
area NearbyMonsters Each {_ Hurt 50}
```

- ## Sum types (aka tagged unions)

```
getOrElse := {|option default|
    option {
        |None| default
        |Some value| value
    }
}


getOrElse {_ Some 42} 0 == 42
getOrElse {_ None} 0 == 0


A sum type constructor is a lambda function that when
invoked applies its argument to the constructor tag and
the constructor arguments.

Recall that {_ Some 42} == {|f| f.Some(42)}
```

- Funk is lacking:
  - A standard library
  - A package manager
  - A type system

{|Funk|}

Try Funk:

https://rawgit.com/Ahnfelt/funk/master/index.html

GitHub:

github.com/Ahnfelt/funk

~ 500 lines of code