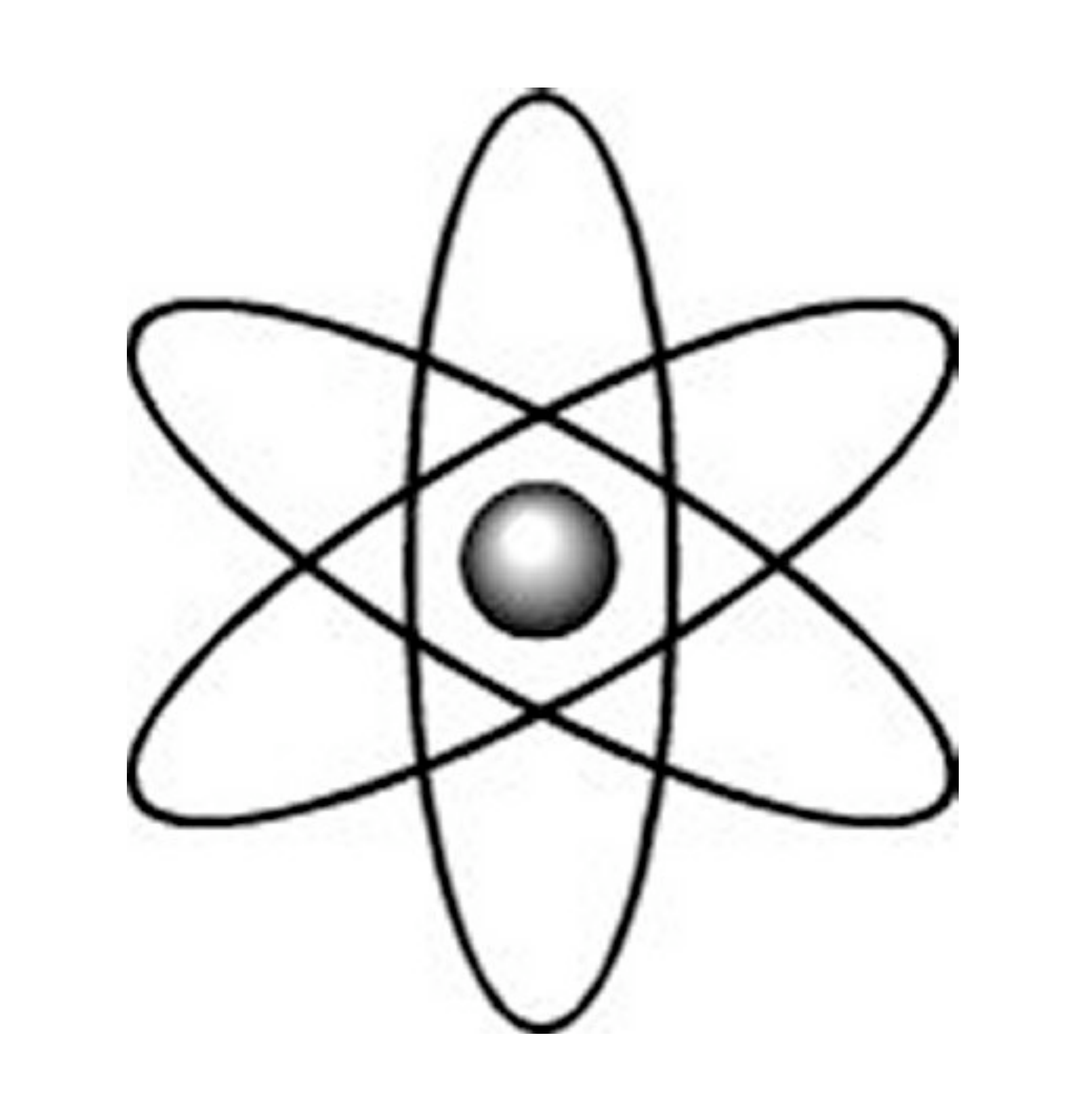
(parenthetically) (speaking)

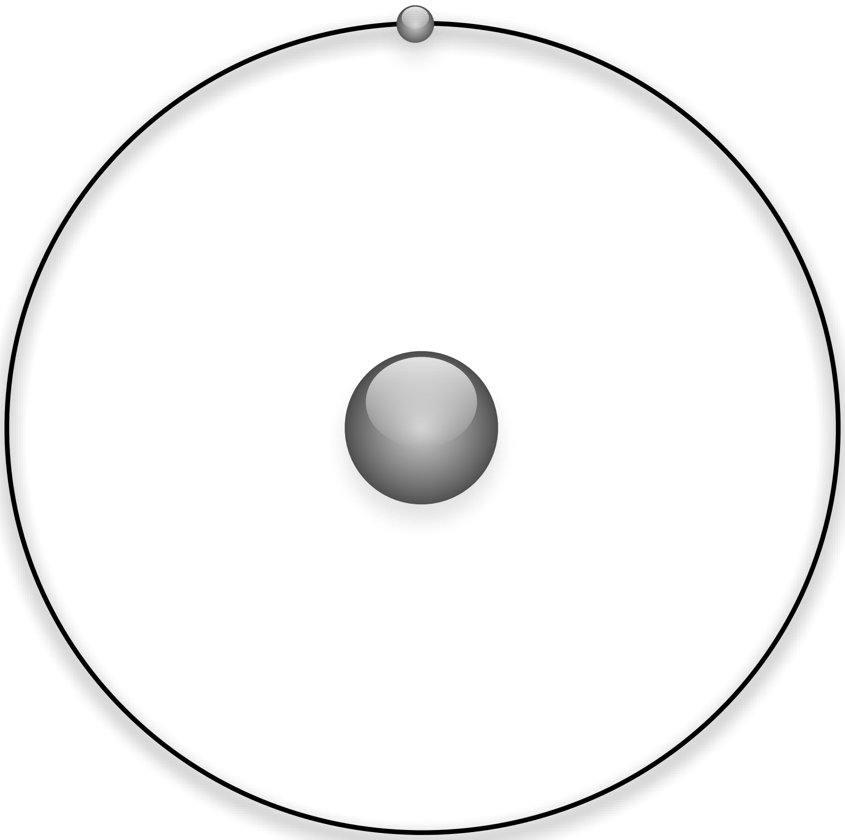
Jim Weirich
Chief Scientist / EdgeCase
jim@edgecase.com
@jimweirich

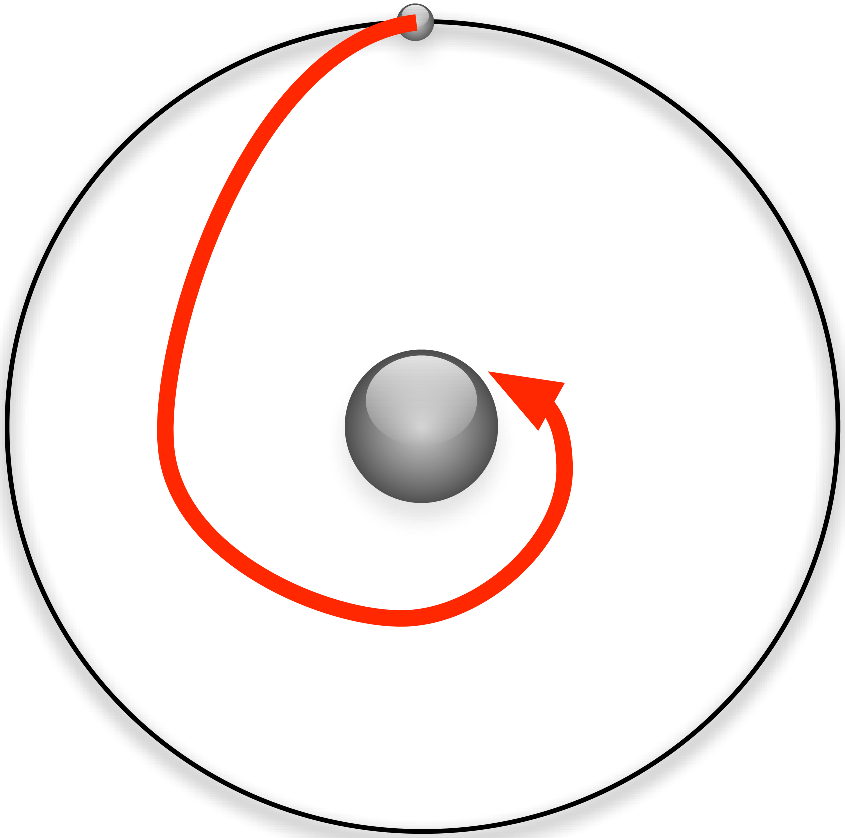
Niels Bohr

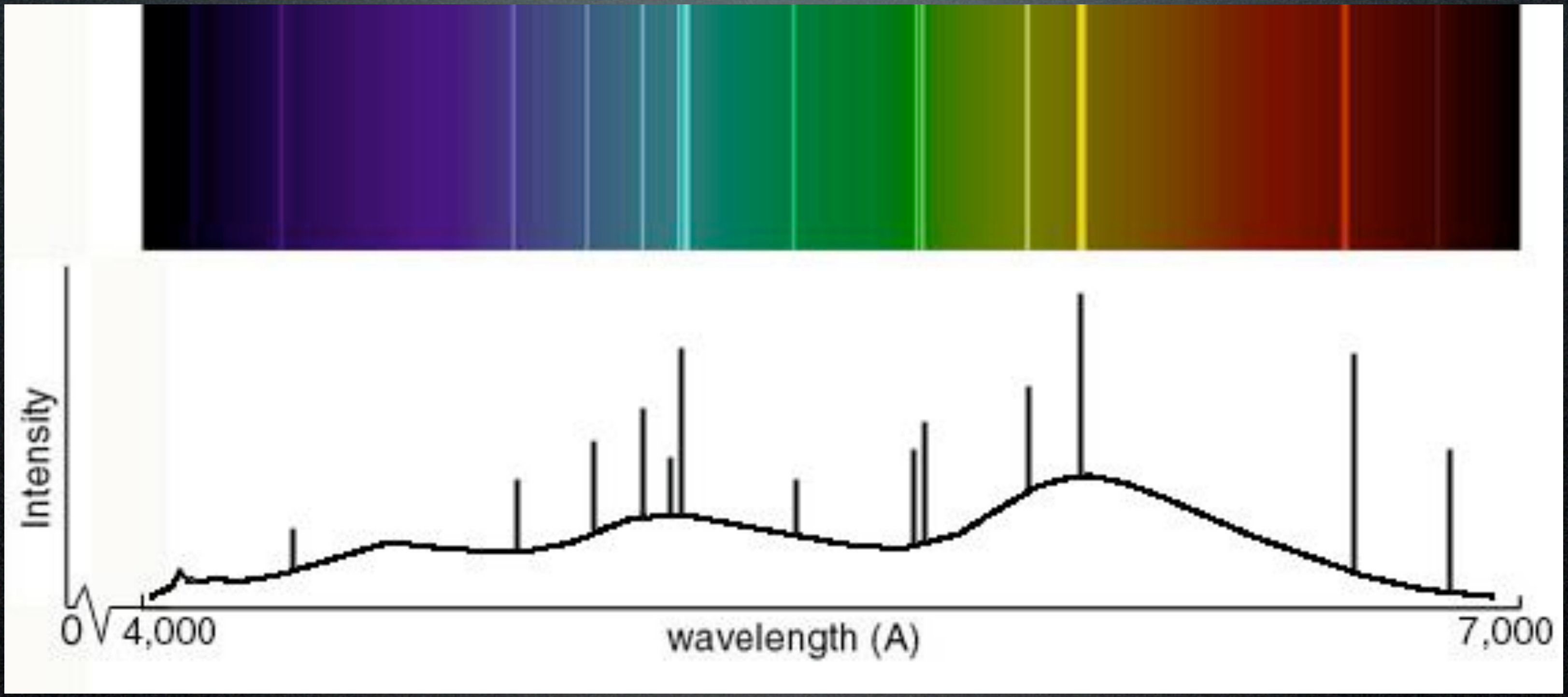
1885 – 1962

Bohr / Einstein Debates

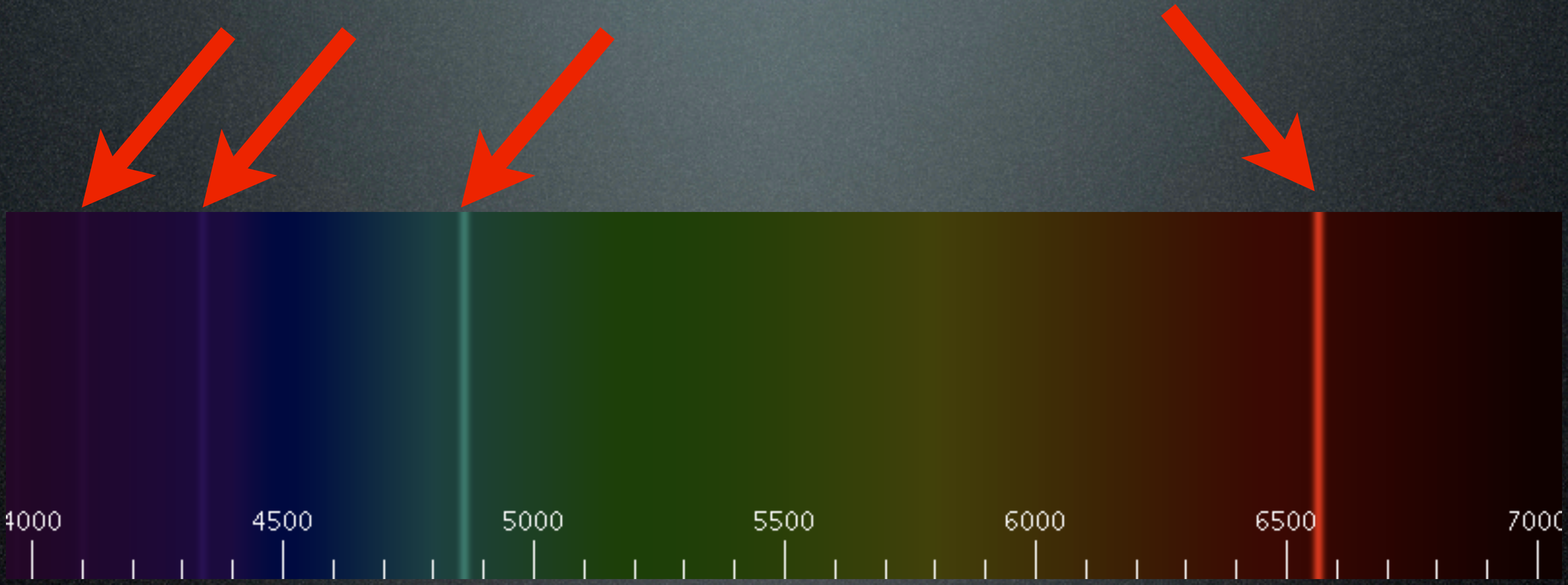




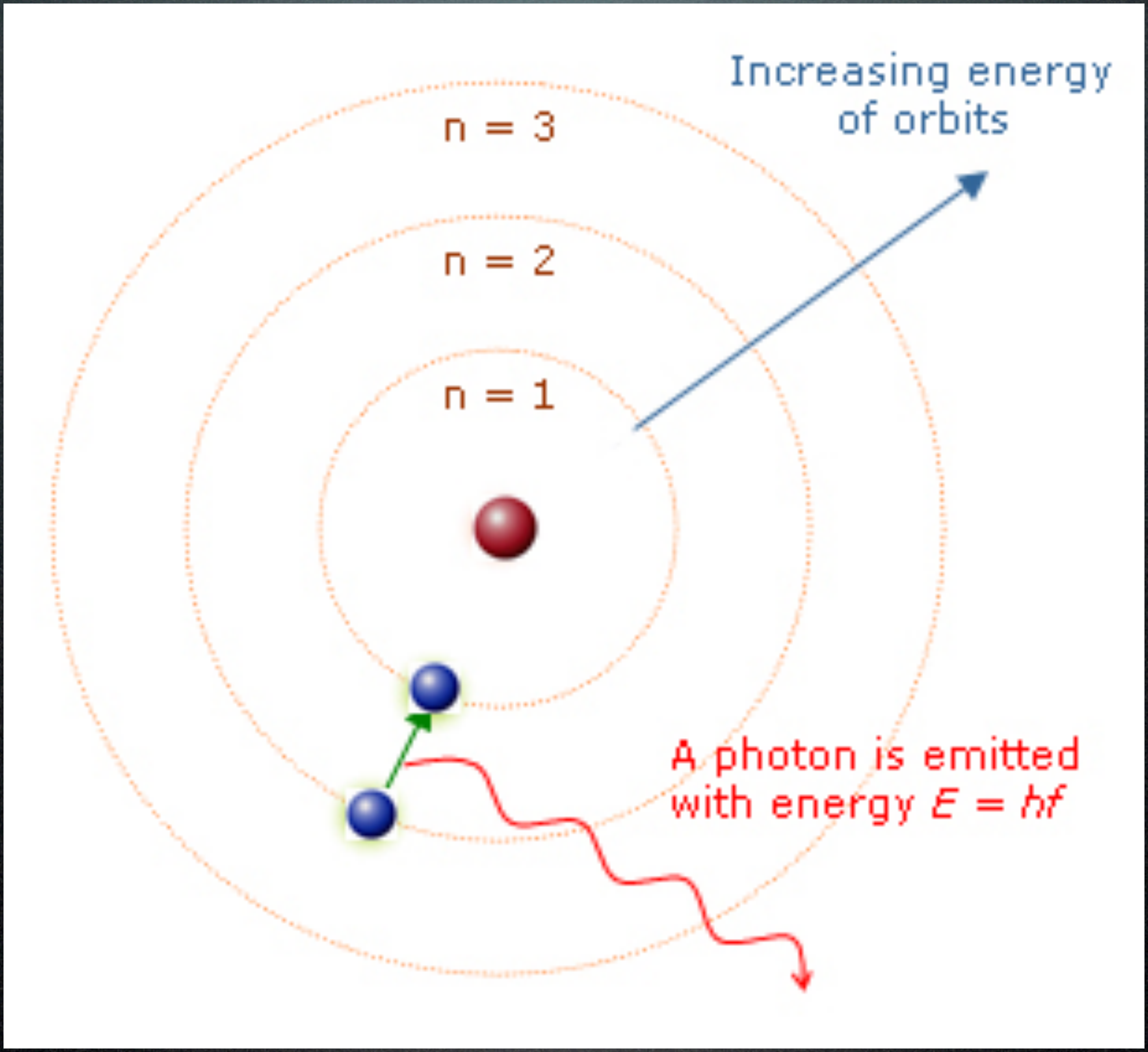


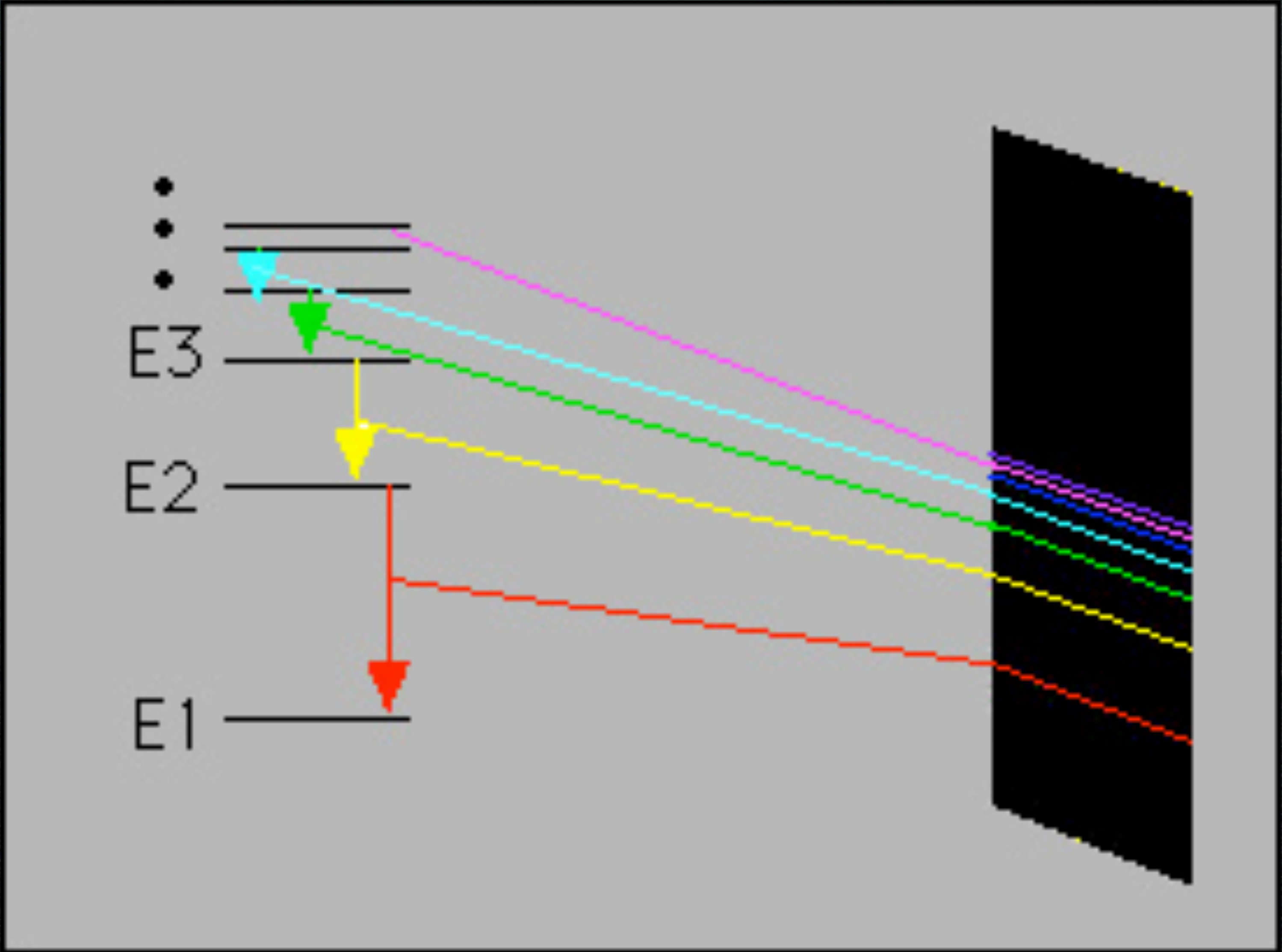


Carbon

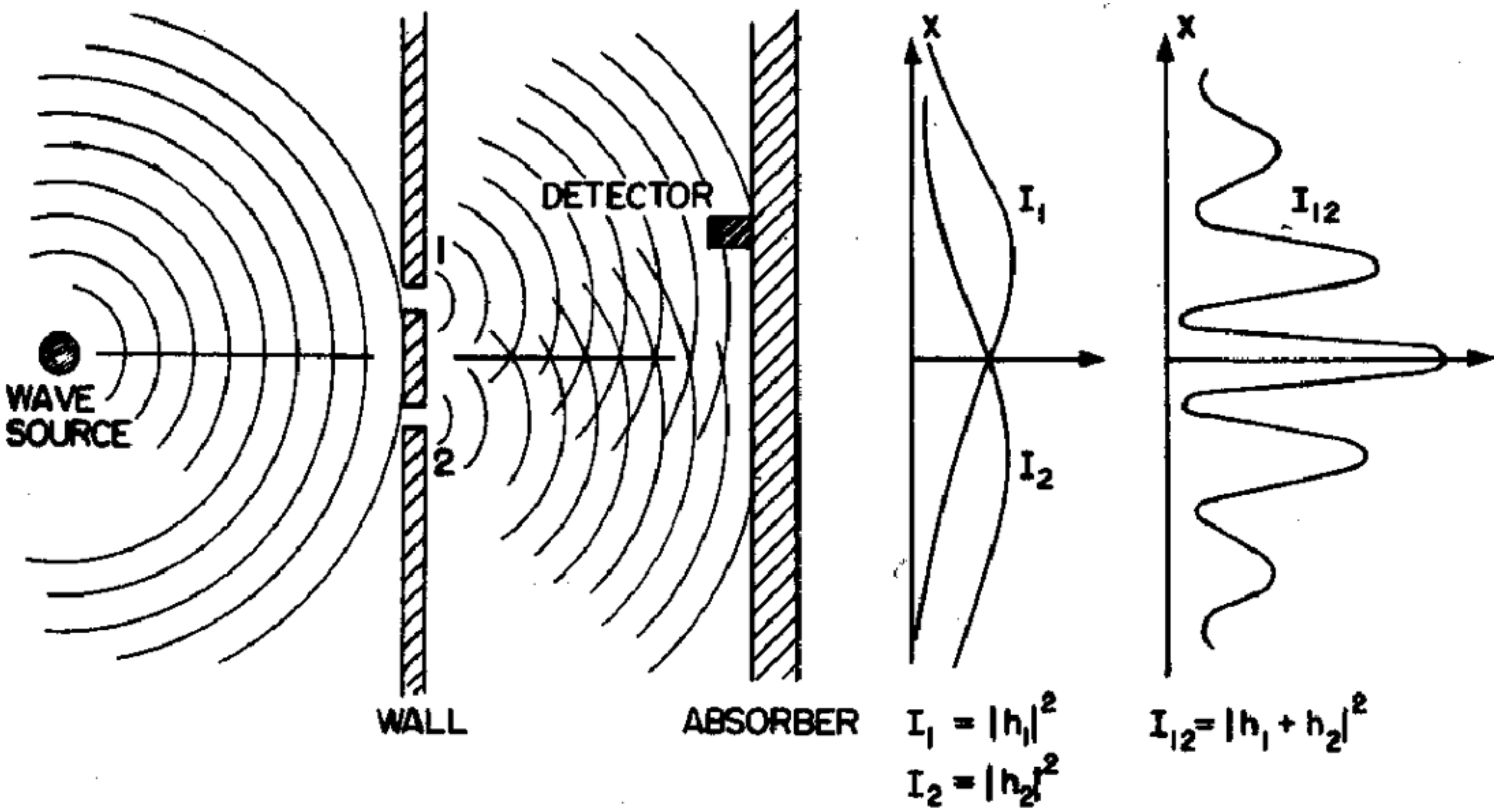


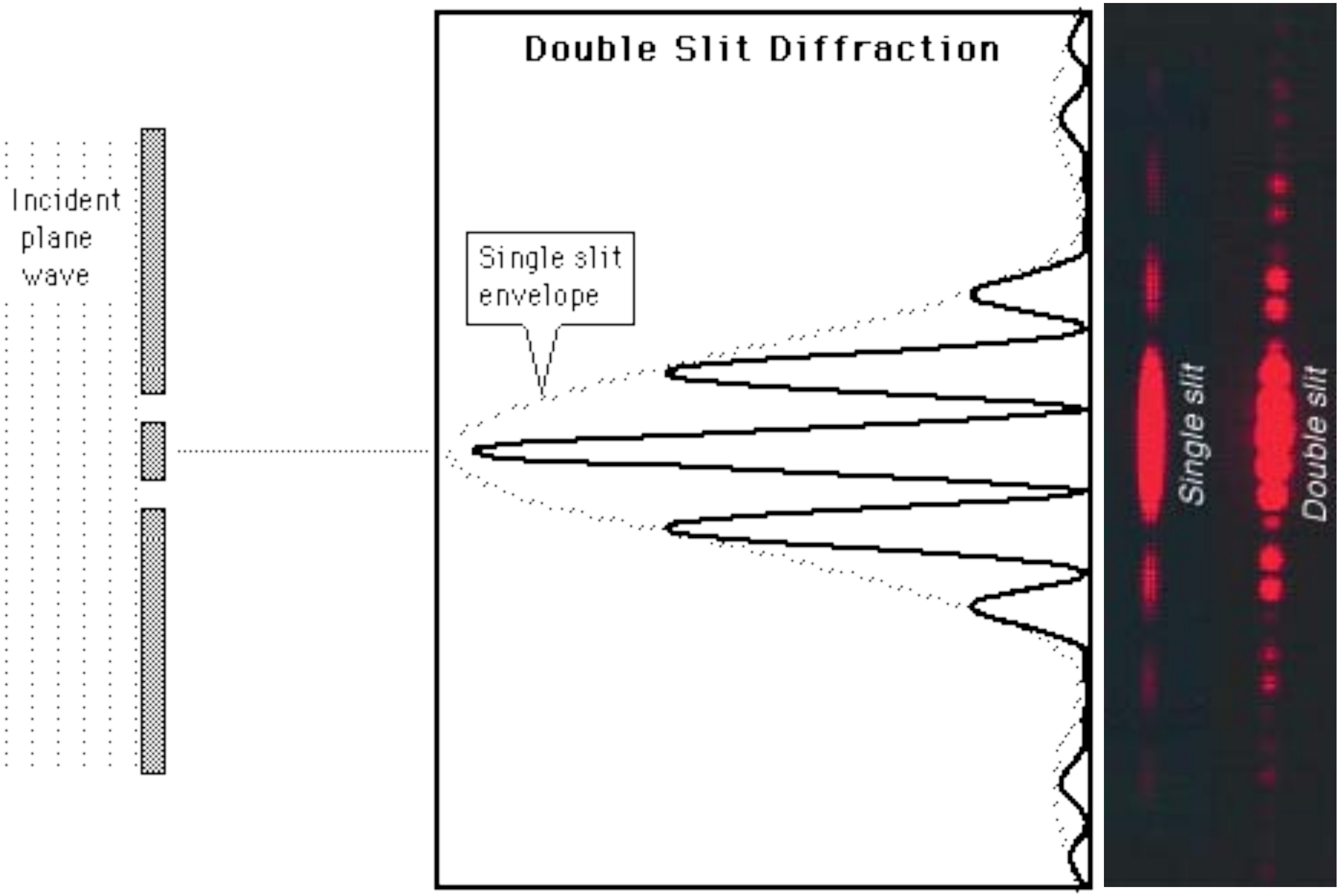
Hydrogen


Increasing energy of orbits
n = 3
n = 2
n = 1
A photon is emitted with energy $E = hf$


E3
E2
E1

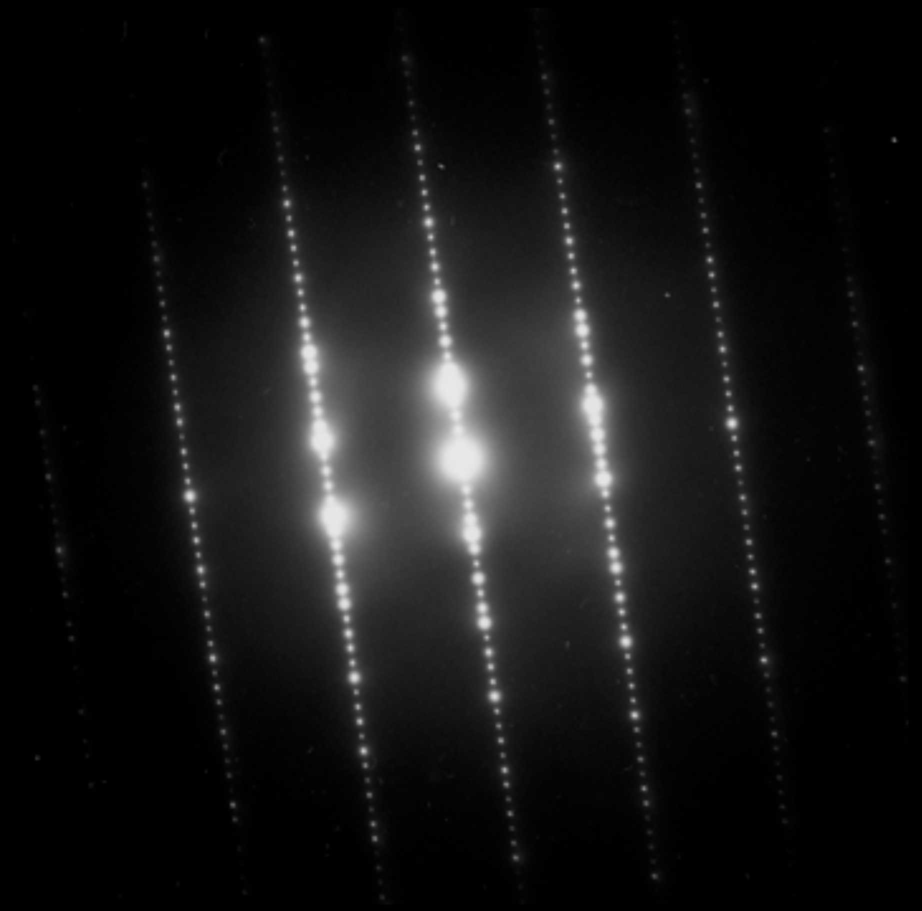
Lateral Thinking


DETECTOR
1
2
WAVE SOURCE
WALL
ABSORBER
x
I_1
I_2
$I_1 = |h_1|^2$
$I_2 = |h_2|^2$
x
I_{12}
$I_{12} = |h_1 + h_2|^2$

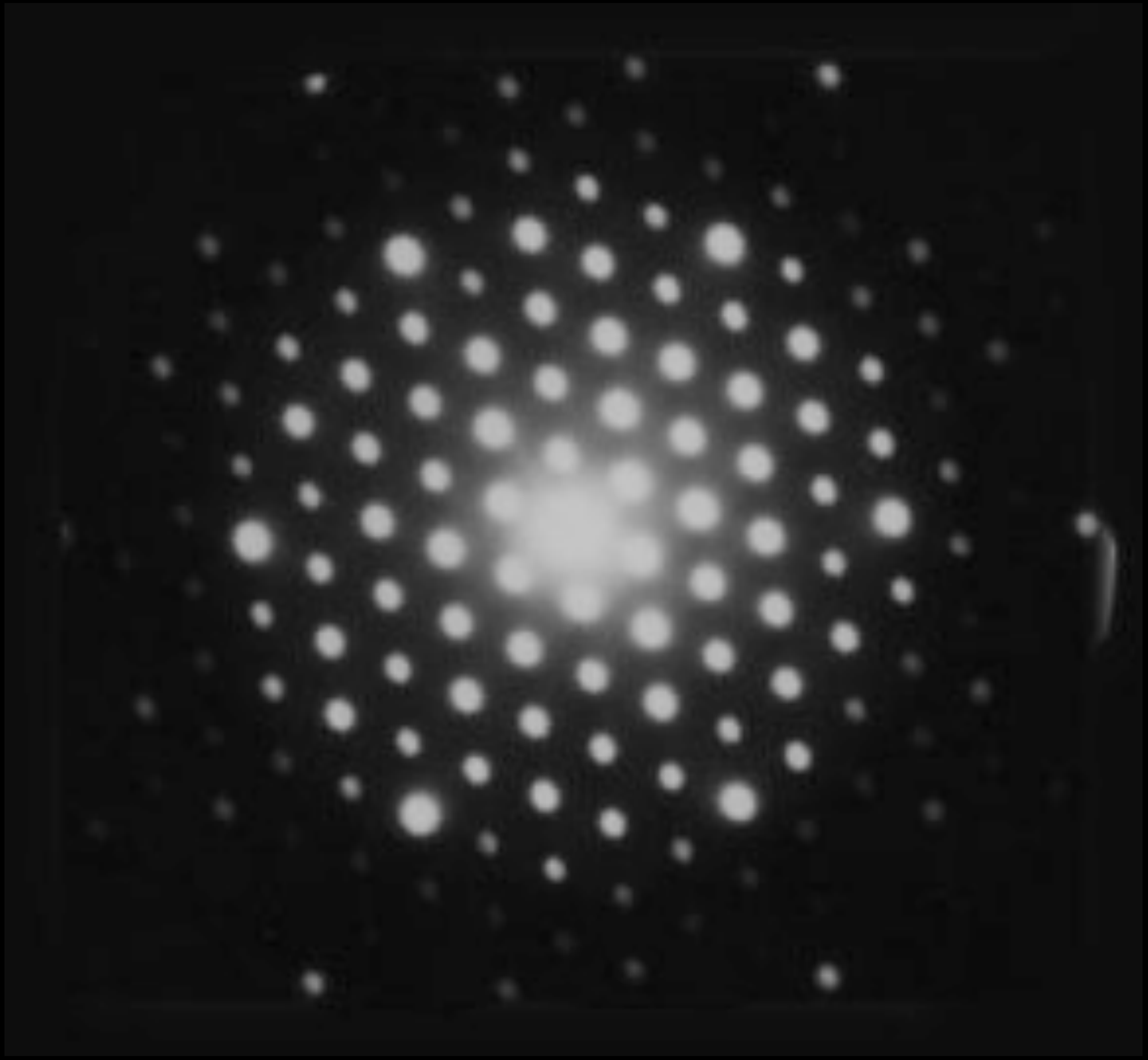

Incident plane wave
Double Slit Diffraction
Single slit envelope
Single slit
Double slit

Seems Simple

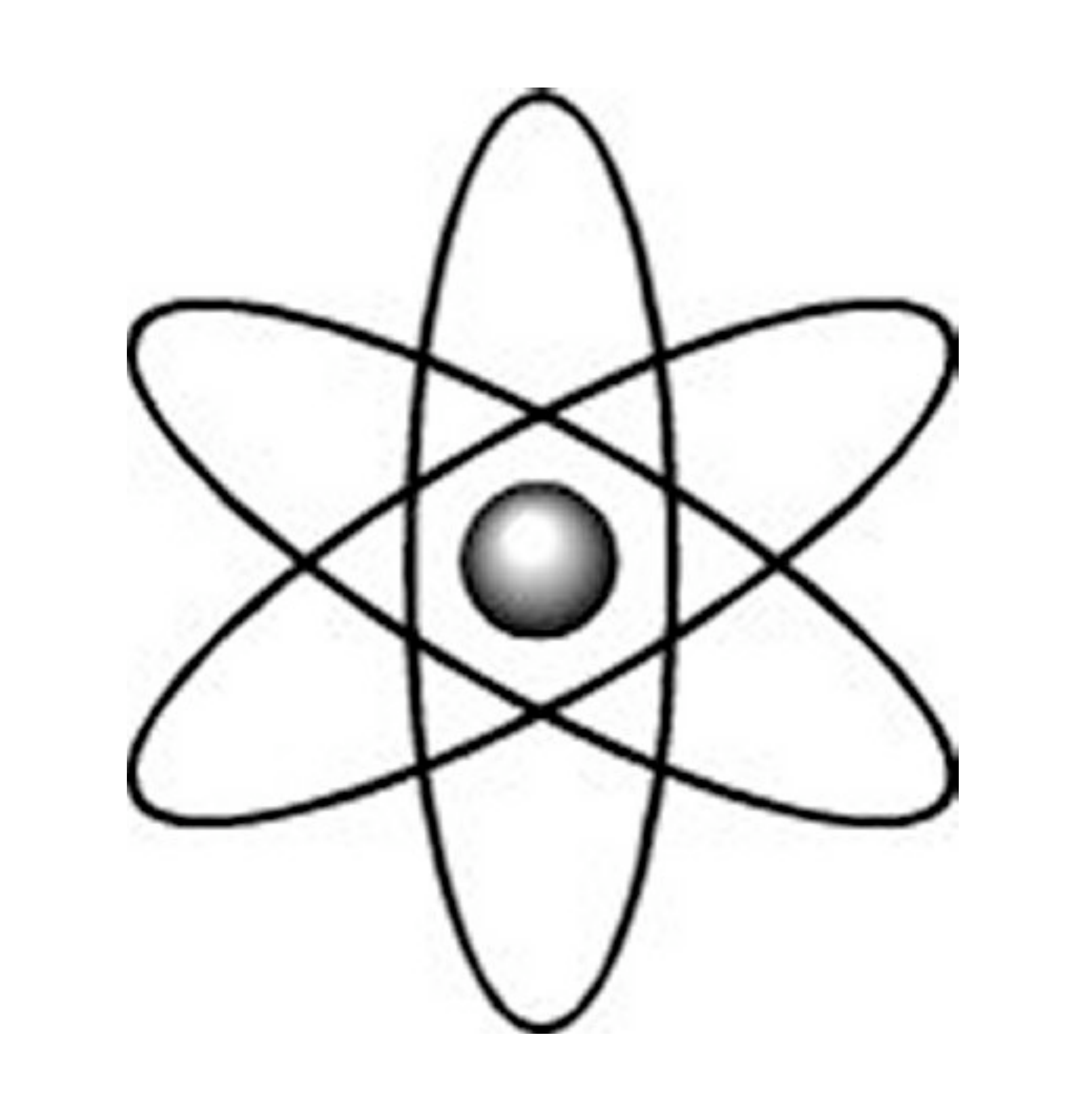
But ...

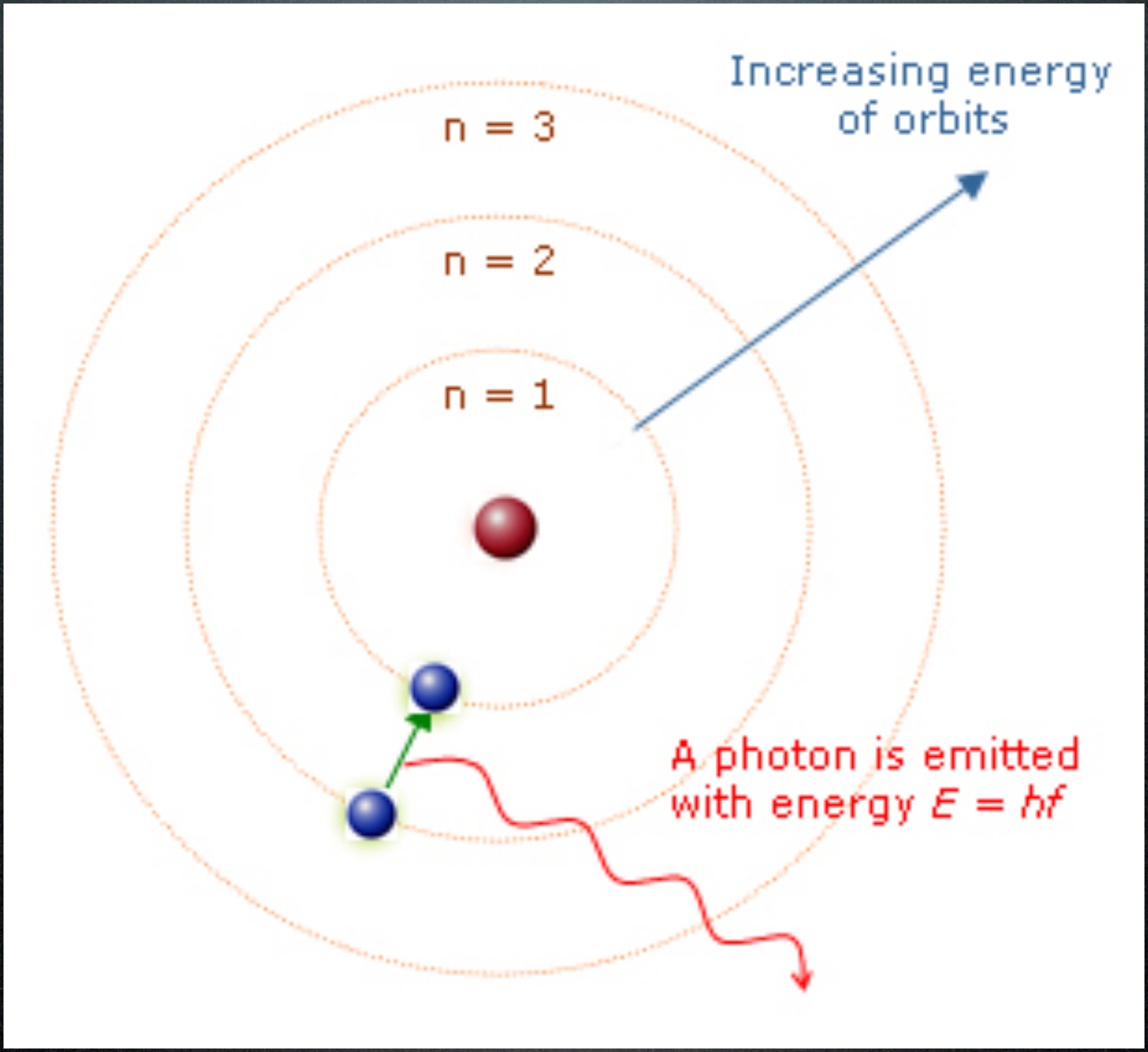


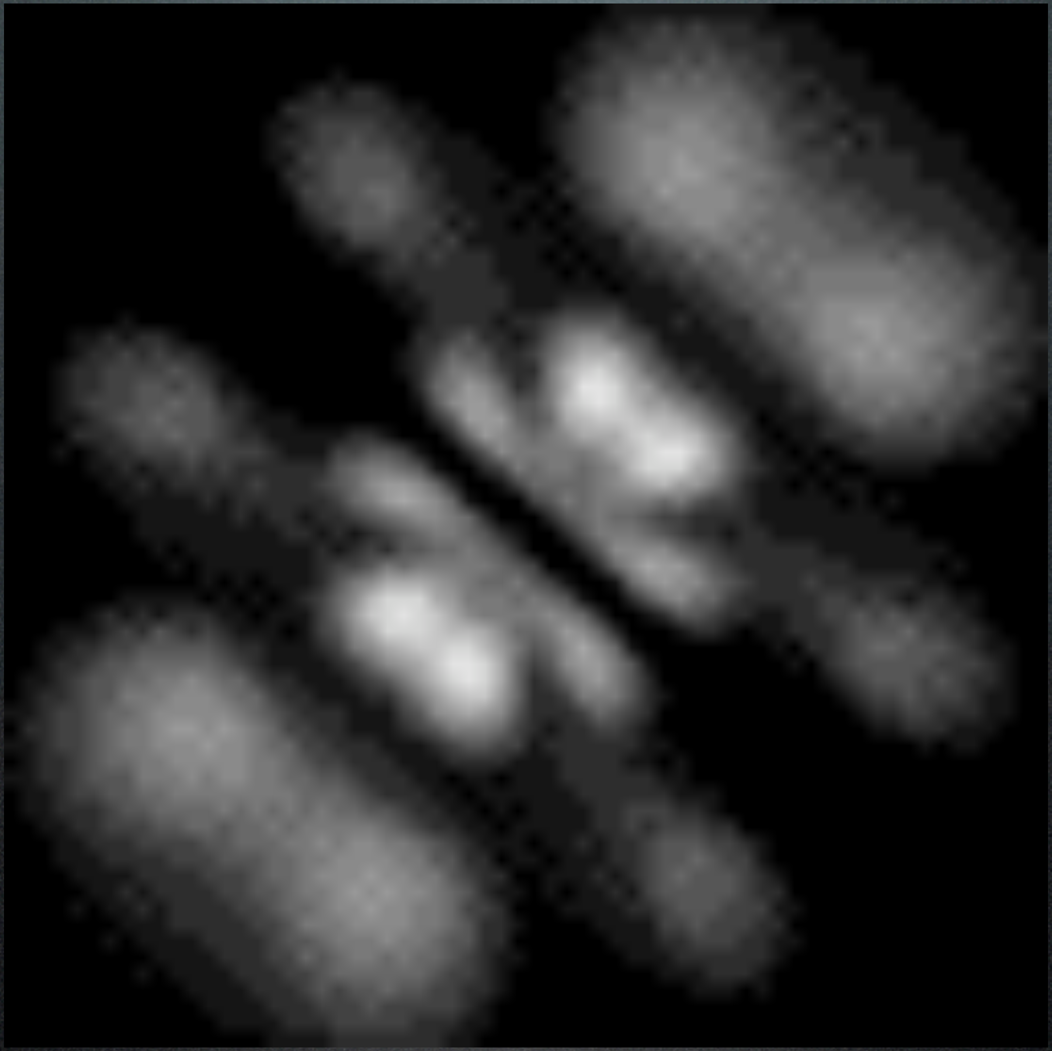
Electron Diffraction



Wave VS Particle Duality




Increasing energy of orbits
n = 3
n = 2
n = 1
A photon is emitted with energy $E = hf$



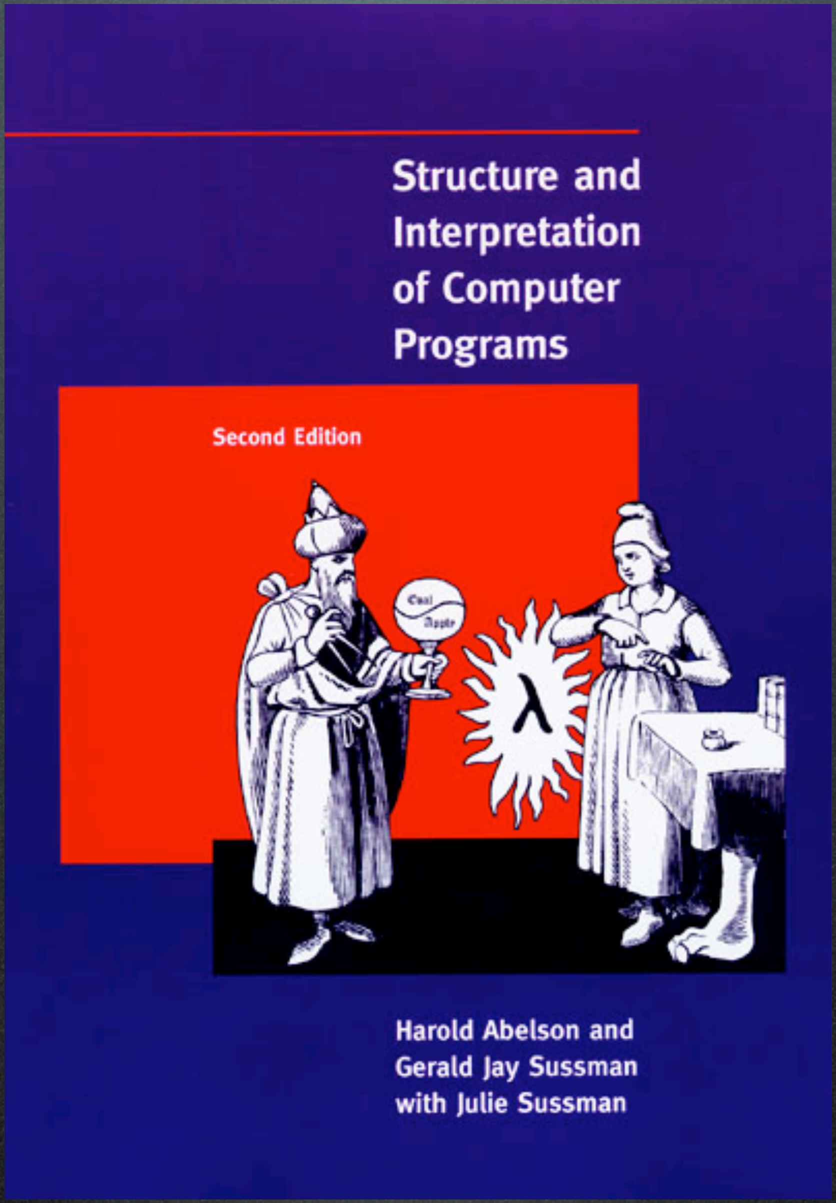
Abstraction

Summary

- Lateral Thinking
- Dualities
- Abstractions

It All Started Here

Rule of 3



Structure and Interpretation of Computer Programs

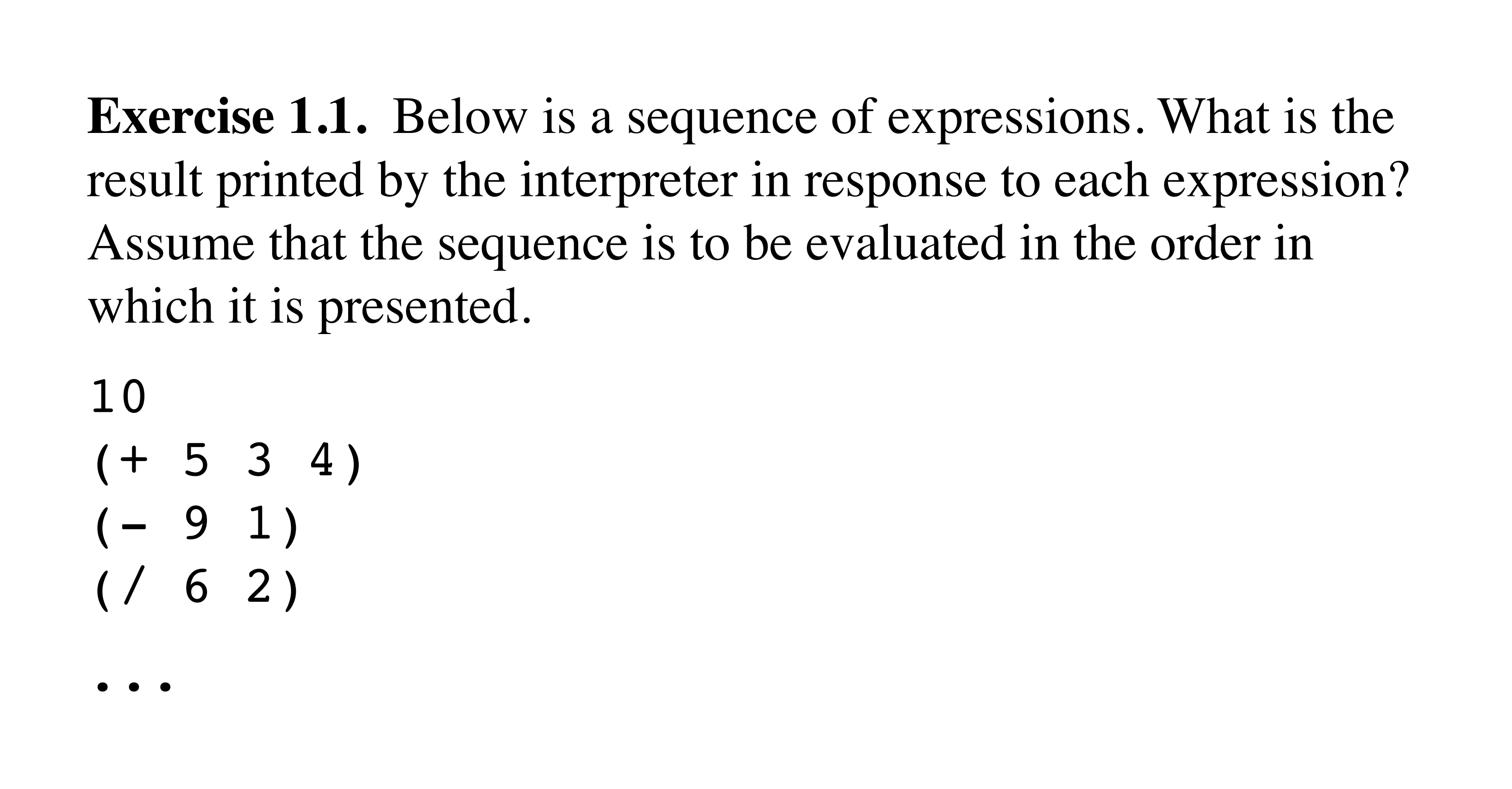
Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

Introductory Text at MIT

Introductory Text
at MIT

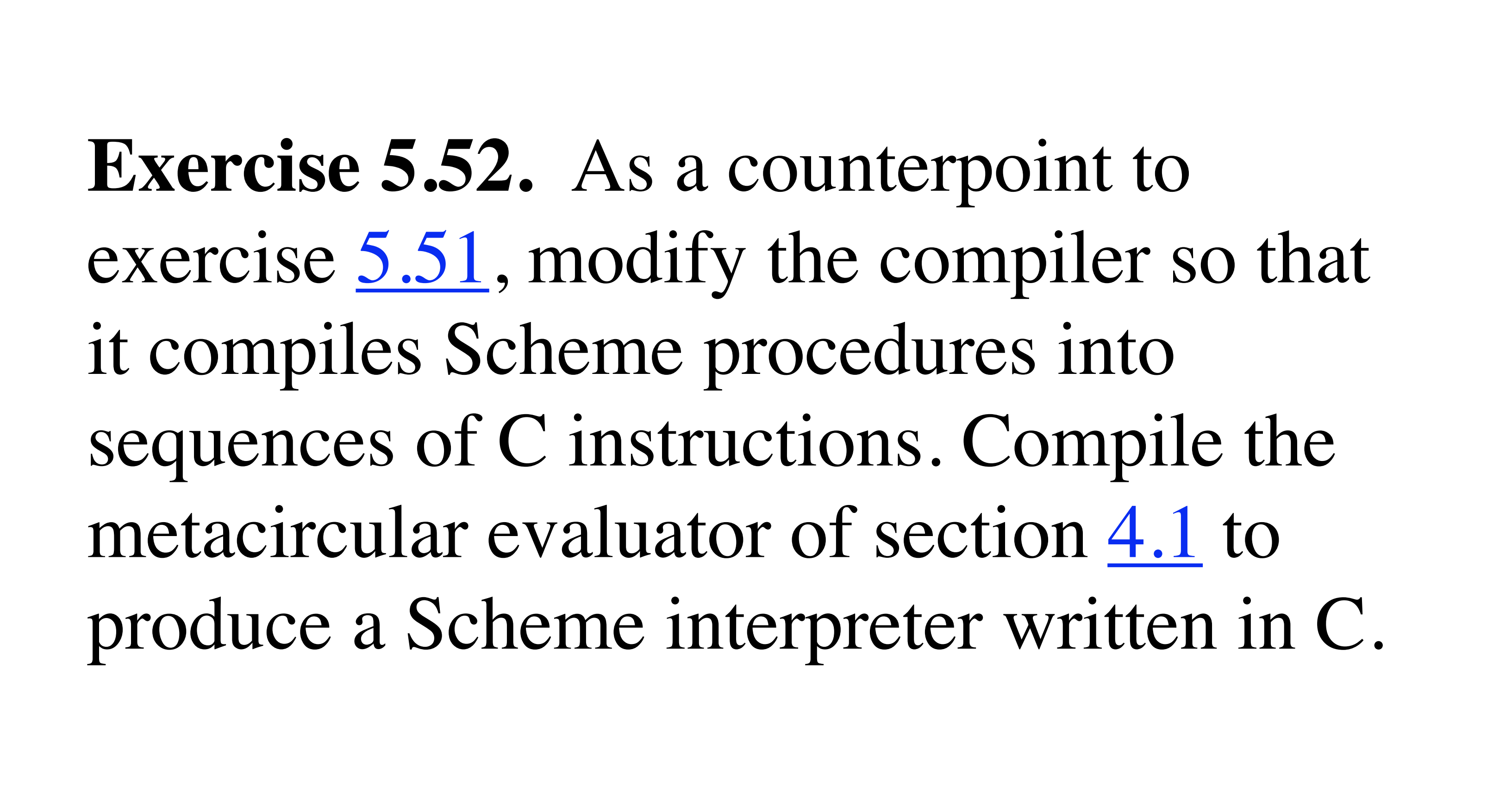
First Exercise



Exercise 1.1. Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
10
(+ 5 3 4)
(- 9 1)
(/ 6 2)

...
```

Last Exercise



Exercise 5.52. As a counterpoint to exercise 5.51, modify the compiler so that it compiles Scheme procedures into sequences of C instructions. Compile the metacircular evaluator of section 4.1 to produce a Scheme interpreter written in C.

Some Cool Things I Learned from Chapter 1

Examples in Ruby!
(not Scheme)

But ... not very idiomatic Ruby

```
def sqrt(x)
  guess = 1
  while (guess*guess - x).abs > 0.01
    guess = average(guess, x / guess)
  end
  guess
end
```

```
sqrt(100)
```

```
$ ruby -I. newton00.rb
10.0000052895642693
```

```
def sqrt(x)
  guess = 1
  while (guess*guess - x).abs > 0.01
    guess = average(guess, x / guess)
  end
  guess
end
```

Is the guess good enough?

```
def sqrt(x)
  guess = 1
  while (guess*guess - x).abs > 0.01
    guess = average(guess, x / guess)
  end
  guess
end
```

```
def sqrt(x)
  guess = 1
  while ! good_enough?(guess, x)
    guess = average(guess, x / guess)
  end
  guess
end
```

Improve the guess

```
def sqrt(x)
  guess = 1
  while ! good_enough?(guess, x)
    guess = average(guess, x / guess)
  end
  guess
end
```

```
def sqrt(x)
  guess = 1
  while ! good_enough?(guess, x)
    guess = improve_guess(guess, x)
  end
  guess
end
```

```
def good_enough?(guess, x)
  (guess*guess - x).abs <= 0.01
end

def improve_guess(guess, x)
  average(guess, x / guess)
end
```

Where is the SQRT Logic?

```
def sqrt(x)
  guess = 1
  while ! good_enough?(guess, x)
    guess = improve_guess(guess, x)
  end
  guess
end
```

Where is the SQRT Logic?

```
def sqrt(x)
  guess = 1
  while ! good_enough?(guess, x)
    guess = improve_guess(guess, x)
  end
  guess
end
```

Generalize

```
def find_root(x,
      good_enough,
      improve_guess)
  guess = 1
  while ! good_enough.(guess, x)
    guess = improve_guess.(guess, x)
  end
  guess
end
```

```
def good_enough?(guess, x)
  (guess*guess - x).abs <= 0.01
end

def improve_guess(guess, x)
  average(guess, x / guess)
end
```

```
sqrt_good_enough = ->(guess, x) {
  (guess*guess - x).abs <= 0.01
}

sqrt_improve_guess = ->(guess, x) {
  average(guess, x / guess)
}
```

```
find_root(100,
    sqrt_good_enough,
    sqrt_improve_guess)
```

```
$ ruby -I. newton02.rb
10.000052895642693
```

Awkward

```
find_root(100,
    sqrt_good_enough,
    sqrt_improve_guess)
```

Back to find_root

```
def find_root(x,
      good_enough,
      improve_guess)
  guess = 1
  while ! good_enough.(guess, x)
    guess = improve_guess.(guess, x)
  end
  guess
end
```

Create a find_root

```
def make_find_root(good_enough, improve_guess)
  ->(x) {
    guess = 1
    while ! good_enough.(guess, x)
      guess = improve_guess.(guess, x)
    end
    guess
  }
end
```

Create a Specific find_root

```
sqrt = make_find_root(
  sqrt_good_enough,
  sqrt_improve_guess)

sqrt.(100)
```

Revisit SQRT Logic

```
sqrt_good_enough = ->(guess, x) {
  (guess*guess - x).abs <= 0.01
}

sqrt_improve_guess = ->(guess, x) {
  average(guess, x / guess)
}
```

Generate the Logic?

```
square = ->(x) { x * x }

sqrt_good_enough =
  make_good_enough(square)

sqrt_improve_guess =
  make_improve_guess(square)
```

Generate the Logic?

```
def make_good_enough(function)
  ->(guess, x) {
    ???
  }
end

def make_improve_guess(function)
  ->(guess, x) {
    ???
  }
end
```

Final Root Finder

```
def make_root_finder(f)
  find_root(
    make_good_enough(f),
    make_improve_guess(f))
end

square = ->(x) { x * x }
sqrt = make_root_finder(square)

sqrt.(100)
```

Odd Direction
(for OO programmers)

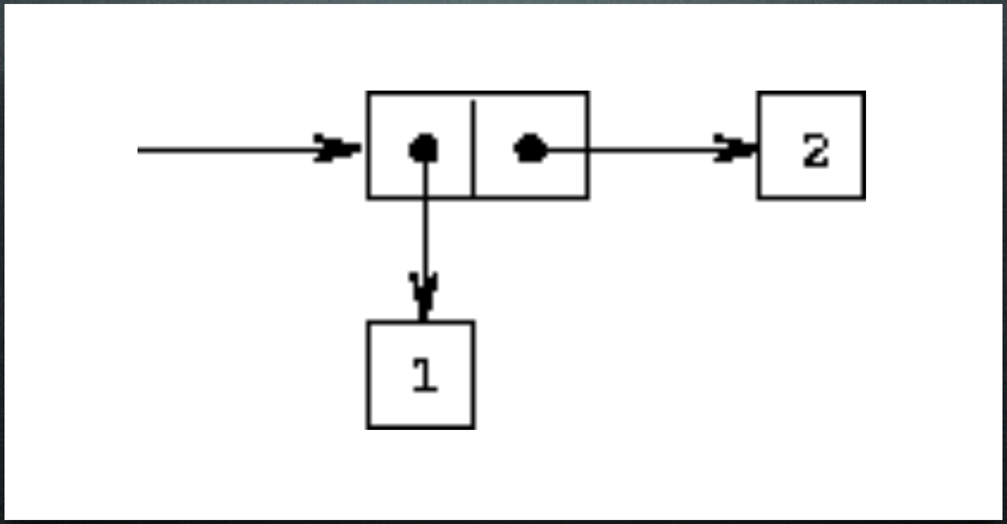
Odd Direction (for OO programmers)

Lateral Thinking

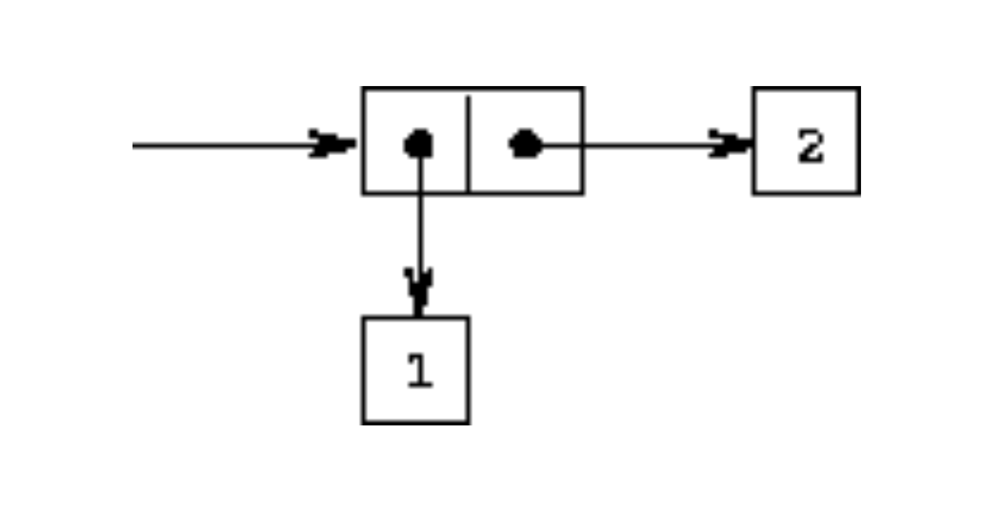
Functional Abstractions are Powerful

Some Cool Things I Learned from Chapter 2

The Cons Cell

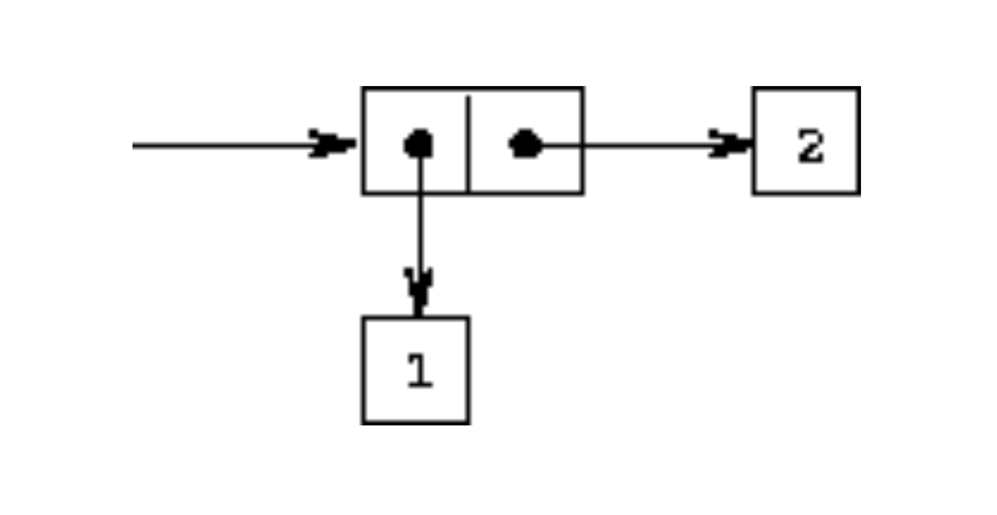


The Cons Cell



(cons 1 2) => (1 . 2)

The Cons Cell



```
(car (cons 1 2)) => 1
(cdr (cons 1 2)) => 2
```

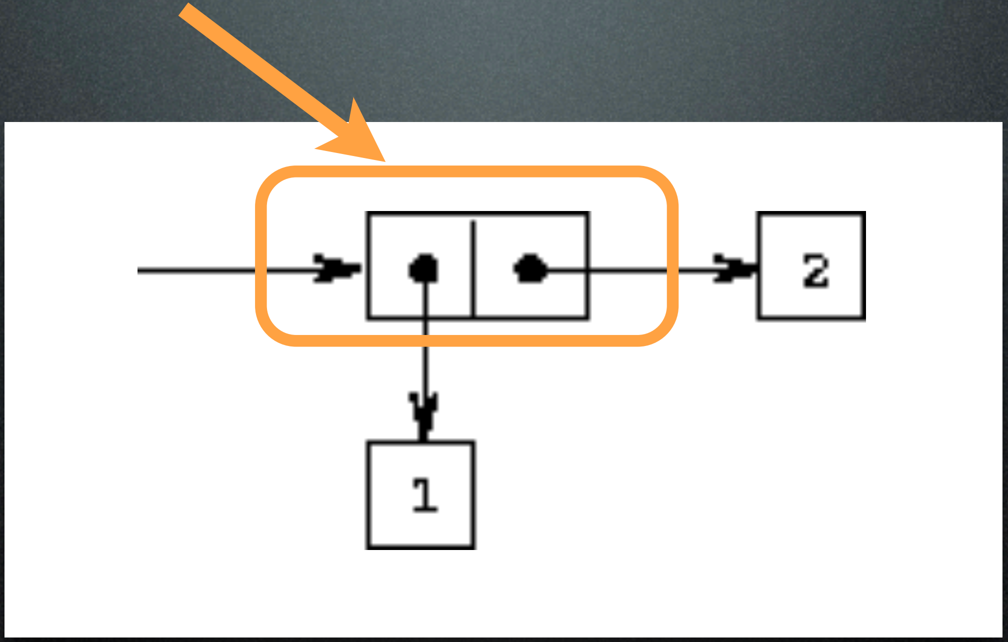
The Cons Cell

```
(cons 3 nil) => (3)
(cons 2 '(3)) => (2 3)
(cons 1 '(2 3)) => (1 2 3)
```

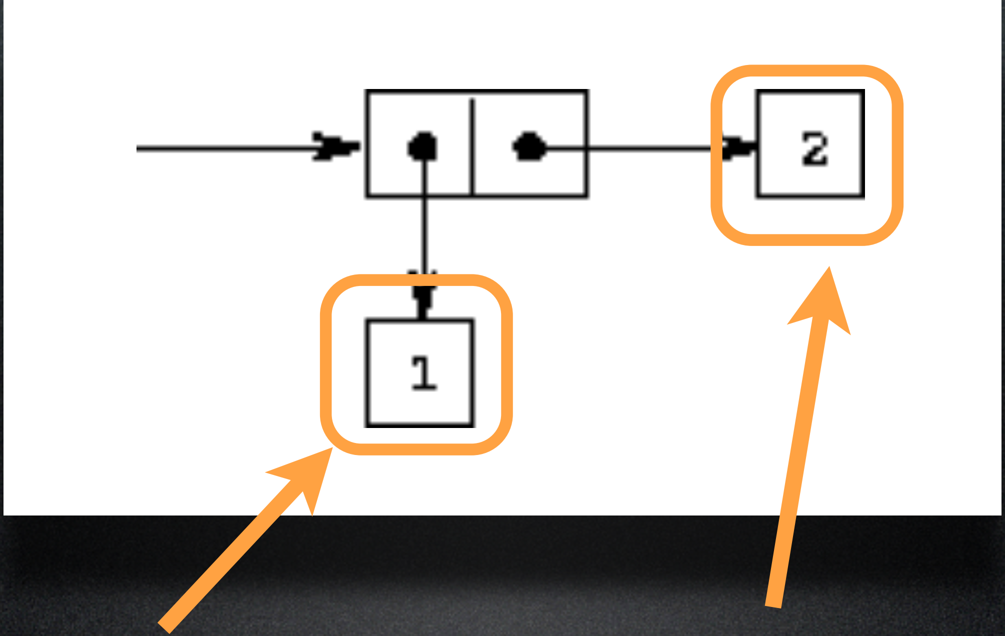
```
(car '(1 2 3)) => 1
(cdr '(1 2 3)) => (2 3)
(cdr '(3)) => nil
```

The Cons Cell

cons: Constructs a cell



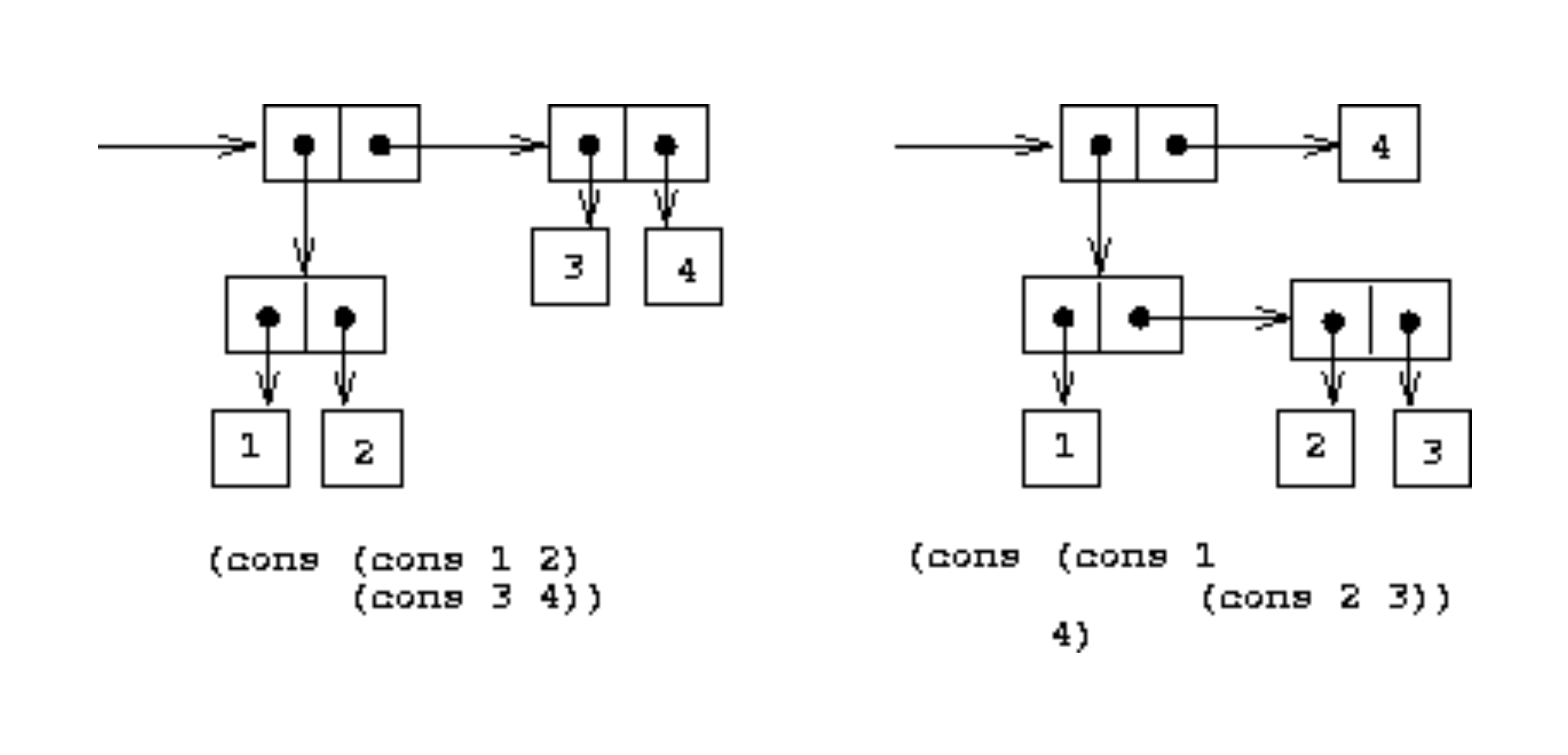
The Cons Cell



car: get the head

cdr: get the tail

The Cons Cell



```
(cons (cons 1 2)
      (cons 3 4))
```

```
(cons (cons 1
            (cons 2 3))
      4)
```

Using Struct

```
List = Struct.new(:head, :tail)

Cons = ->(h,t)  { List.new(h,t) }
Car  = ->(list) { list.head }
Cdr  = ->(list) { list.tail }
```

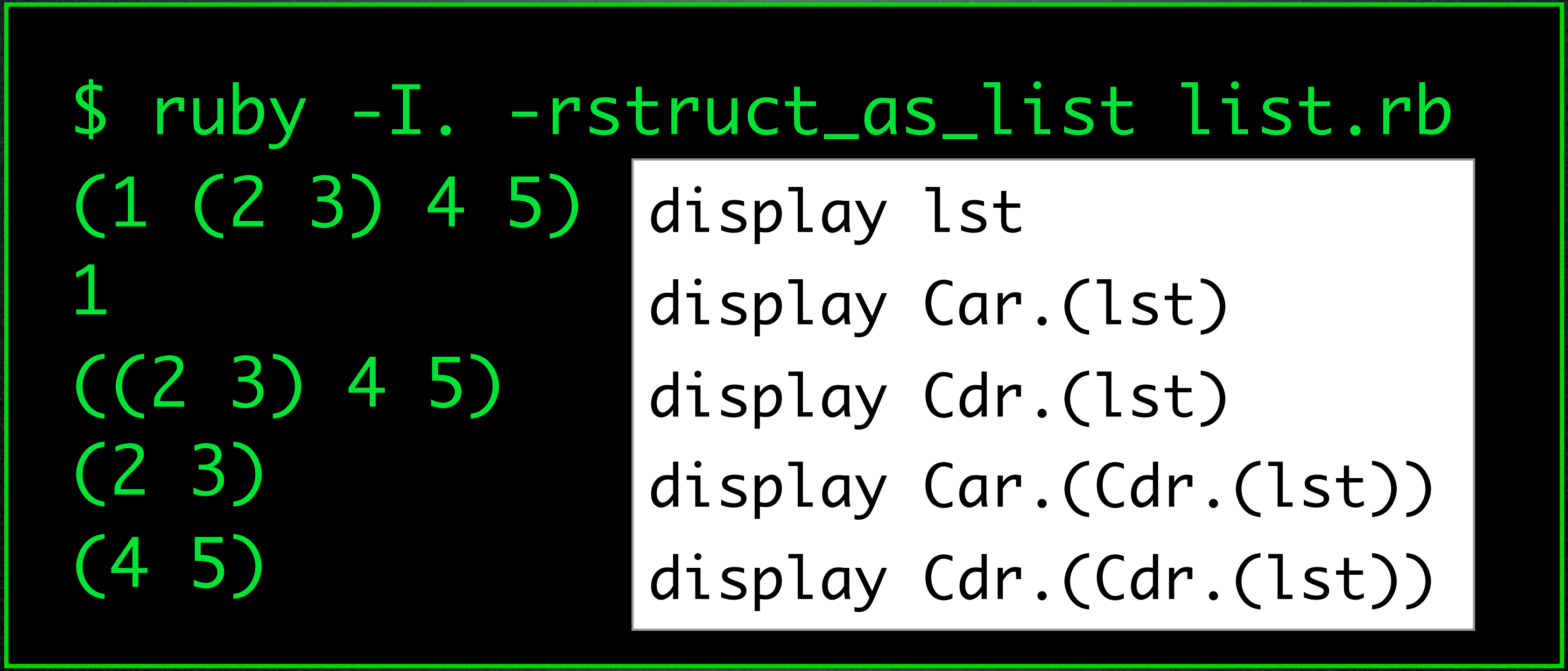
Some Support Functions

```
class Array
  def to_list
    # code to convert an array to a list
  end
end
```

Using Lists

```
require 'list_support'

lst = [1,[2,3],4,5].to_list

display lst
display Car.(lst)
display Cdr.(lst)
display Car.(Cdr.(lst))
display Cdr.(Cdr.(lst))
```

```
$ ruby -I. -rstruct_as_list list.rb
(1 (2 3) 4 5)
1
((2 3) 4 5)
(2 3)
(4 5)
```



```
$ ruby -I. -rstruct_as_list list.rb
(1 (2 3) 4 5)
1
((2 3) 4 5)
(2 3)
(4 5)
```

```
display lst
display Car.(lst)
display Cdr.(lst)
display Car.(Cdr.(lst))
display Cdr.(Cdr.(lst))
```

Time for a Twist

Is This REALLY Needed?

```
List = Struct.new(:head, :tail)

Cons = ->(h,t)  { List.new(h,t) }
Car  = ->(list) { list.head }
Cdr  = ->(list) { list.tail }
```

Using Procedures

```
Cons = ->(h,t)  { ->(s) { (s==:h) ? h : t } }
Car  = ->(list) { list.(:h) }
Cdr  = ->(list) { list.(:t) }
```

No Changes to Output

```
$ ruby -I. -rproc_as_list list.rb
(1 (2 3) 4 5)
1
((2 3) 4 5)
(2 3)
(4 5)
```

Abstractions:

Freedom From Implementation Details

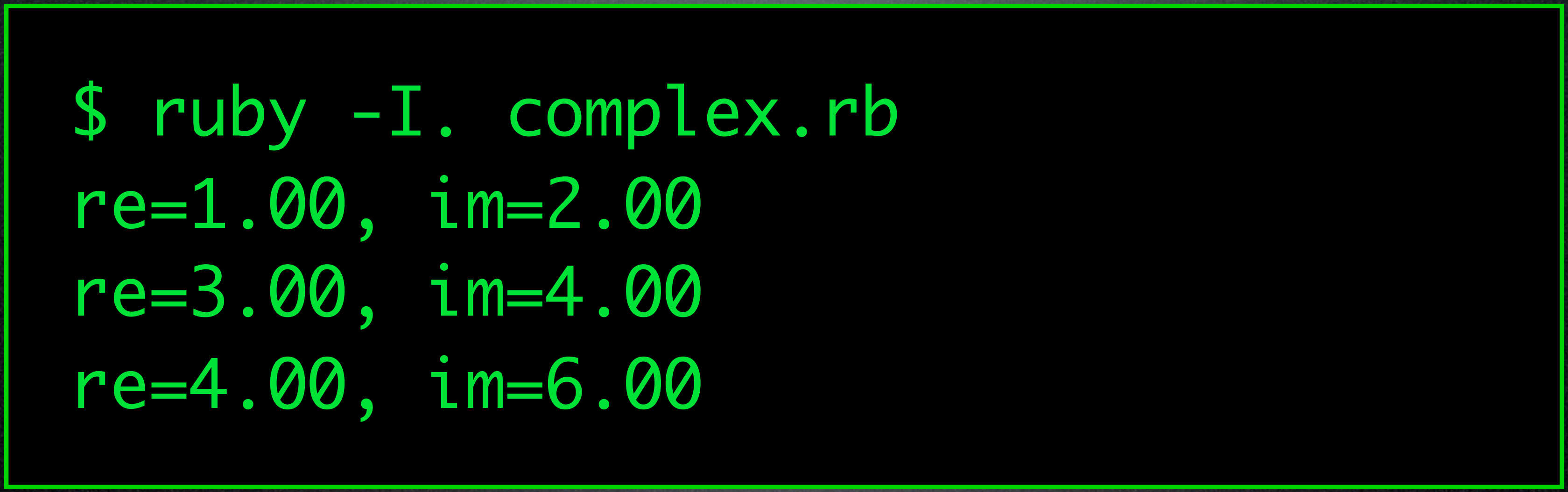
Code VS Data Duality

More Cool Chapter 2 Stuff

```
def make_complex(r,i)
  Cons.(r,i)
end

def re(c)
  Car.(c)
end

def im(c)
  Cdr.(c)
end
```

```
def complex_add(n1, n2)
  make_complex(
    re(n1) + re(n2),
    im(n1) + im(n2))
end

def complex_subtract(n1, n2)
  make_complex(
    re(n1) - re(n2),
    im(n1) - im(n2))
end
```

```
num1 = make_complex(1,2)
num2 = make_complex(3, 4)

print_complex(num1)
print_complex(num2)
print_complex(complex_add(num1, num2))
```



```
$ ruby -I. complex.rb
re=1.00, im=2.00
re=3.00, im=4.00
re=4.00, im=6.00
```

Limitations?

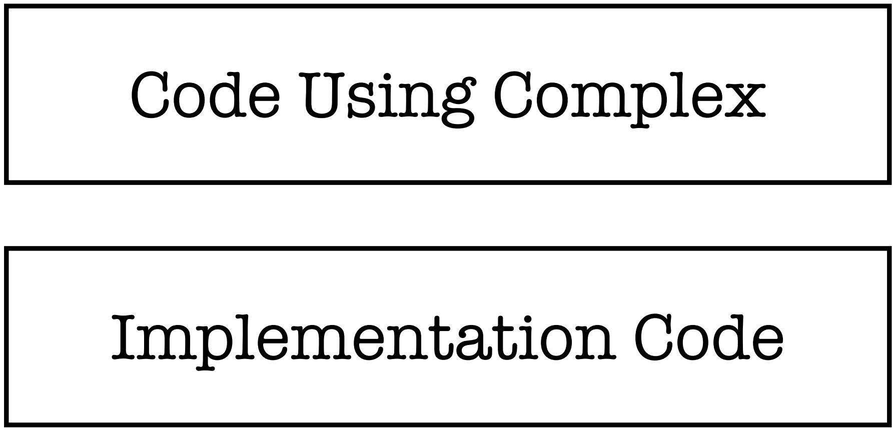
Alternate Representations

```
def make_complex(r,i)
  ->(method) {
    { re: r, im: i }[method]
  }
end

def re(c)
  c.(:re)
end

def im(c)
  c.(:im)
end
```

```
def make_polar_complex(mag,ang)
  angle_in_radians = Math::PI * ang / 180
  ->(method) {
    {
      re: ->() {
        Math.cos(angle_in_radians) * mag
      },
      im: ->() {
        Math.sin(angle_in_radians) * mag
      },
    }[method].call()
  }
end
```

```
$ ruby -I. closures_as_objects.rb
re=1.00, im=2.00
re=3.00, im=4.00
re=4.00, im=6.00
```

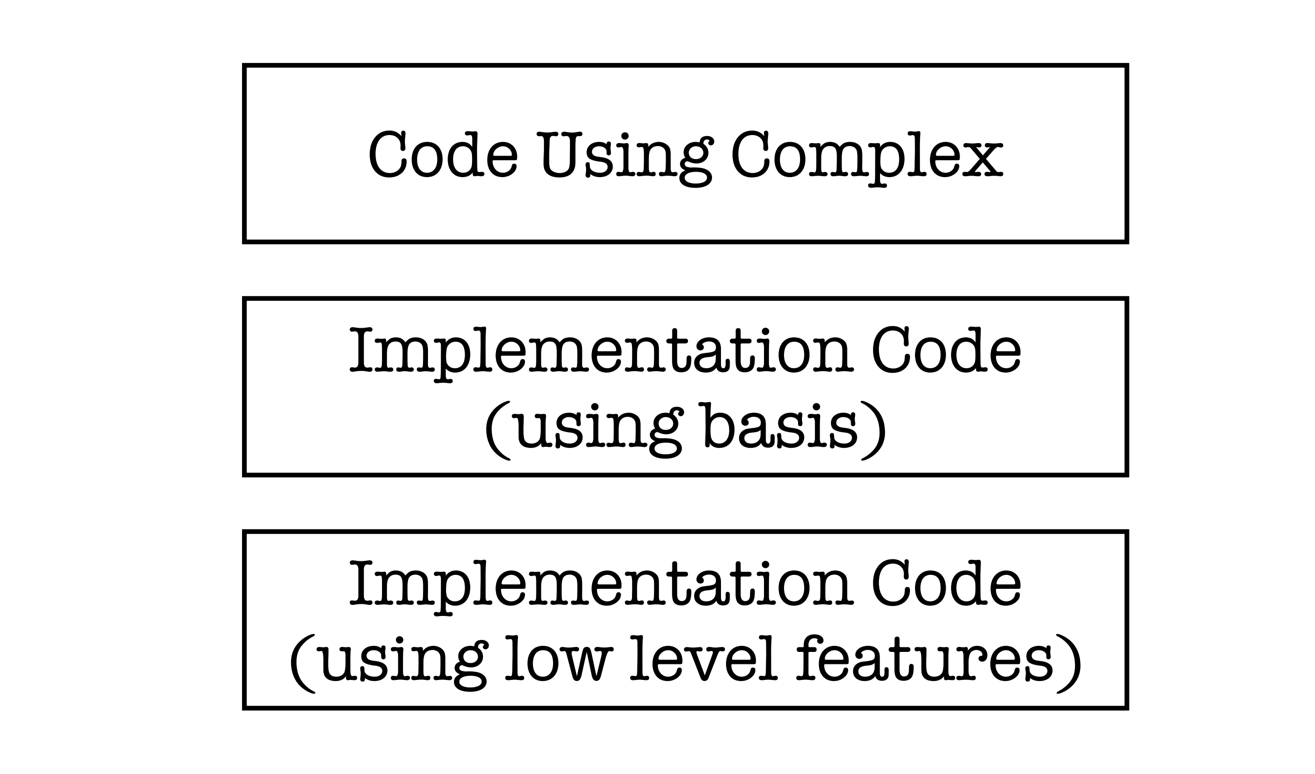
Levels of Abstraction



Code Using Complex

Implementation Code

Levels of Abstraction



Code Using Complex

Implementation Code (using basis)

Implementation Code (using low level features)

Object VS Closure Duality

Chapter 3?

Summary

Look For Non-Traditional Solutions

Find Good Abstractions

Embrace Duality

Resources

http://mitpress.mit.edu/sicp/

http://
groups.google.com/
group/
wizardbookstudy

http://github.com/jimweirich/presentation_parenthetically_speaking

The End

Questions?