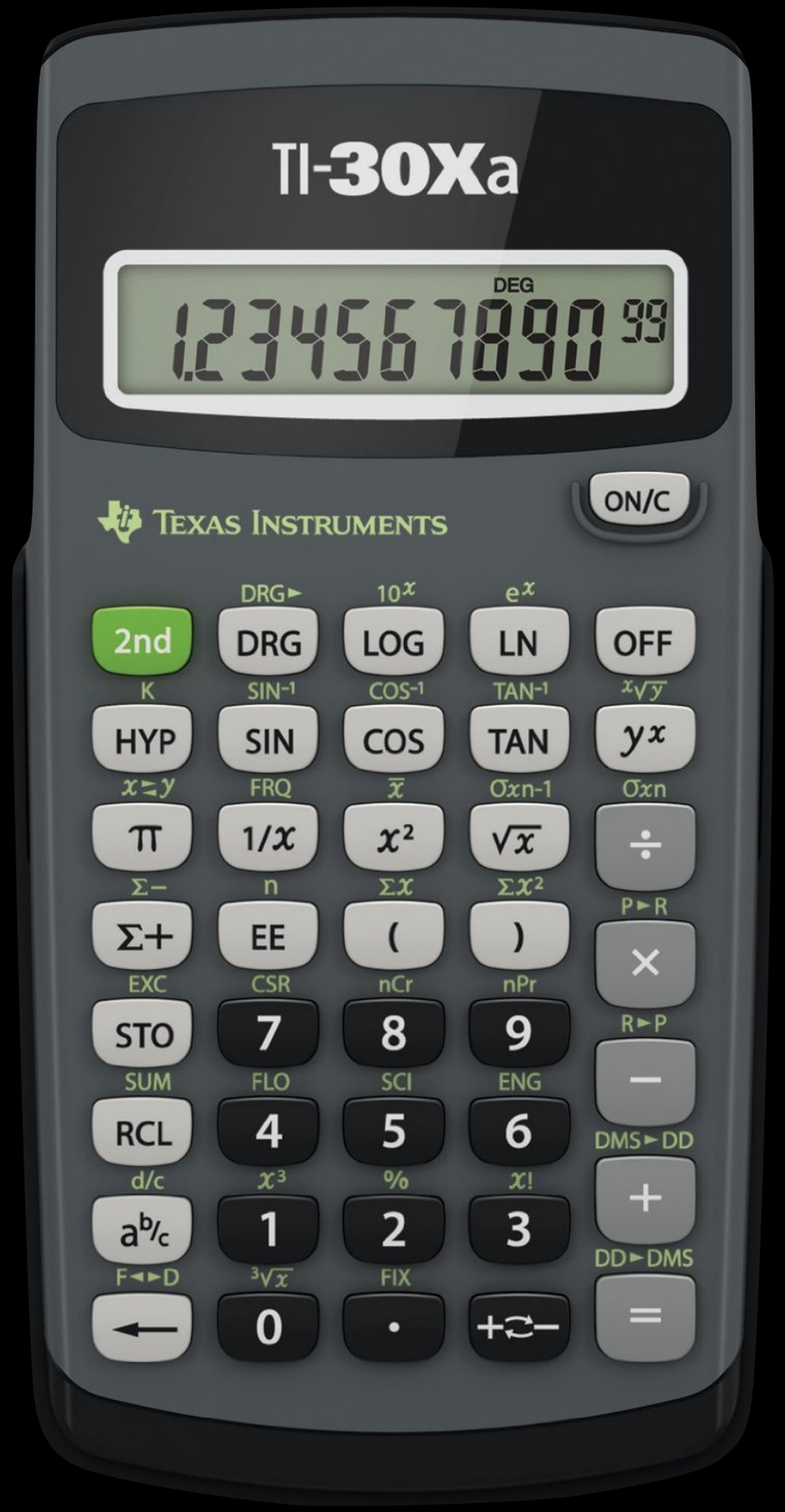
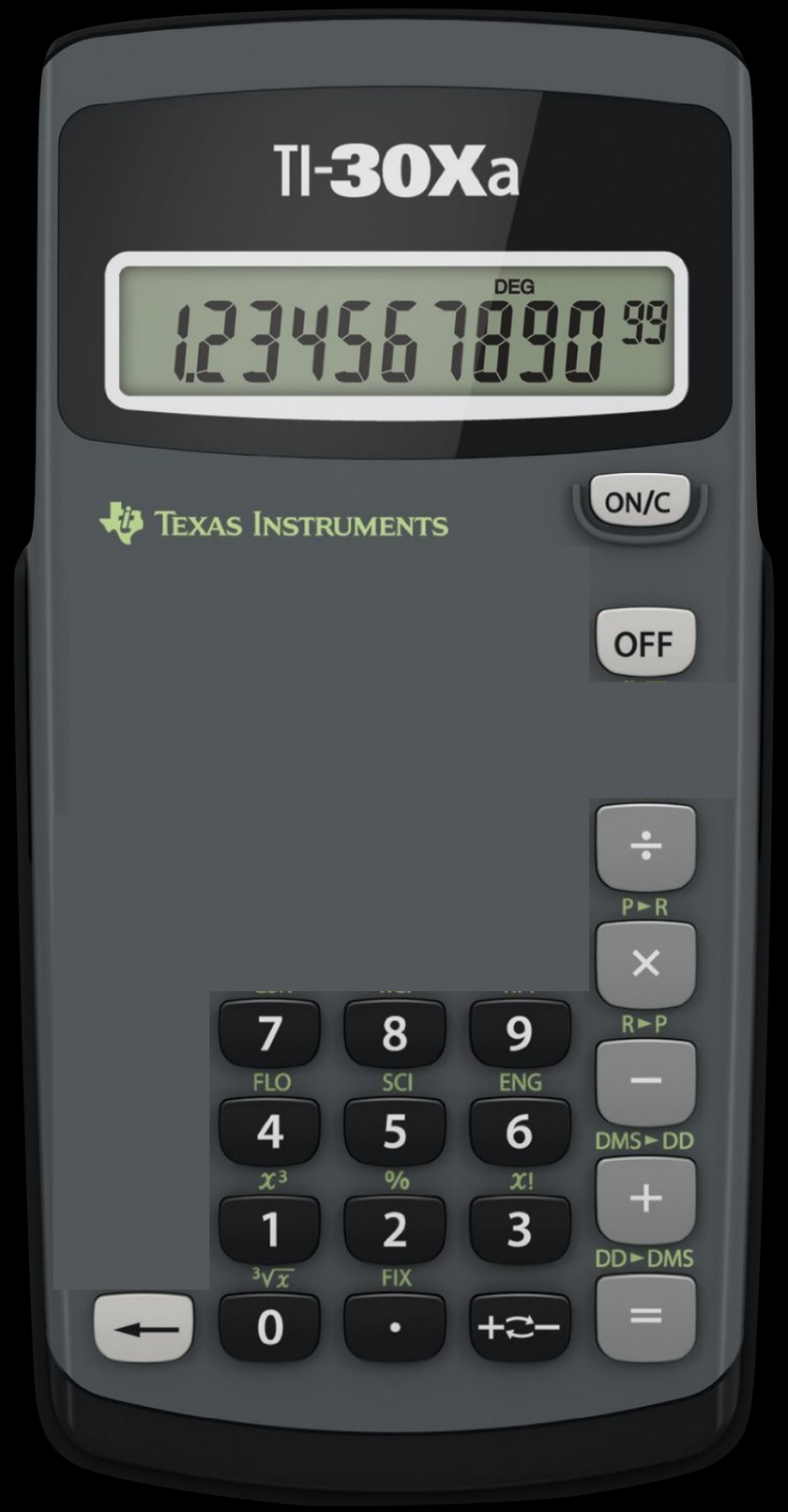
#8 Interpreters and Tail Calls

TA: Jerry Chen (jerry.c@berkeley.edu)

```
>>> (+ 1 (* 5 2))
  File "<stdin>", line 1
    (+ 1 (* 5 2))
               ^
SyntaxError: invalid syntax
>>> screw it, I'm going back to Scheme
  File "<stdin>", line 1
    screw it, I'm going back to Scheme
           ^
SyntaxError: invalid syntax
```


TI-30Xa
DEG
1.234567890 99
Texas Instruments
ON/C
DRG►
10^x
e^x
2nd
DRG
LOG
LN
OFF
K
SIN-1
COS-1
TAN-1
x√y
HYP
SIN
COS
TAN
y^x
x≷y
FRQ
x̄
σxn-1
σxn
π
1/x
x²
√x
÷
Σ−
n
Σx
Σx²
P►R
Σ+
EE
(
)
×
EXC
CSR
nCr
nPr
R►P
STO
7
8
9
−
SUM
FLO
SCI
ENG
DMS►DD
RCL
4
5
6
+
d/c
x³
%
x!
a b/c
1
2
3
DD►DMS
F◄►D
³√x
FIX
=
0
.
+/−


TI-30Xa
DEG
1.234567890 99
Texas Instruments
ON/C
OFF
÷
P►R
×
R►P
−
DMS►DD
+
DD►DMS
=
7
8
9
FLO
SCI
ENG
4
5
6
x³
%
x!
1
2
3
∛x
FIX
0
•
+/−

The humble Calculator language

Because algebra is all we need

Good ol' fashioned arithmetic

Our favorite Polish prefix notation

Short circuiting boolean expressions

The humble Calculator language

Transforming an expression

```
> (+ (* 1 2) (- 3 4))
1
```

The humble Calculator language

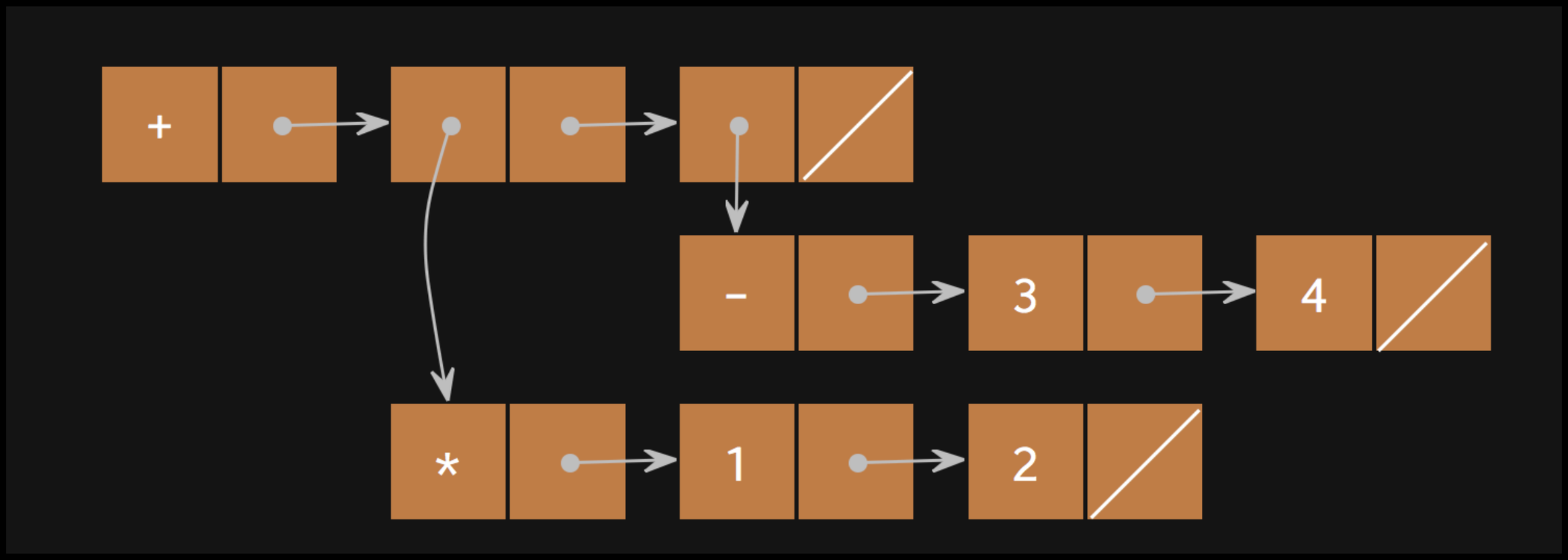
Transforming an expression

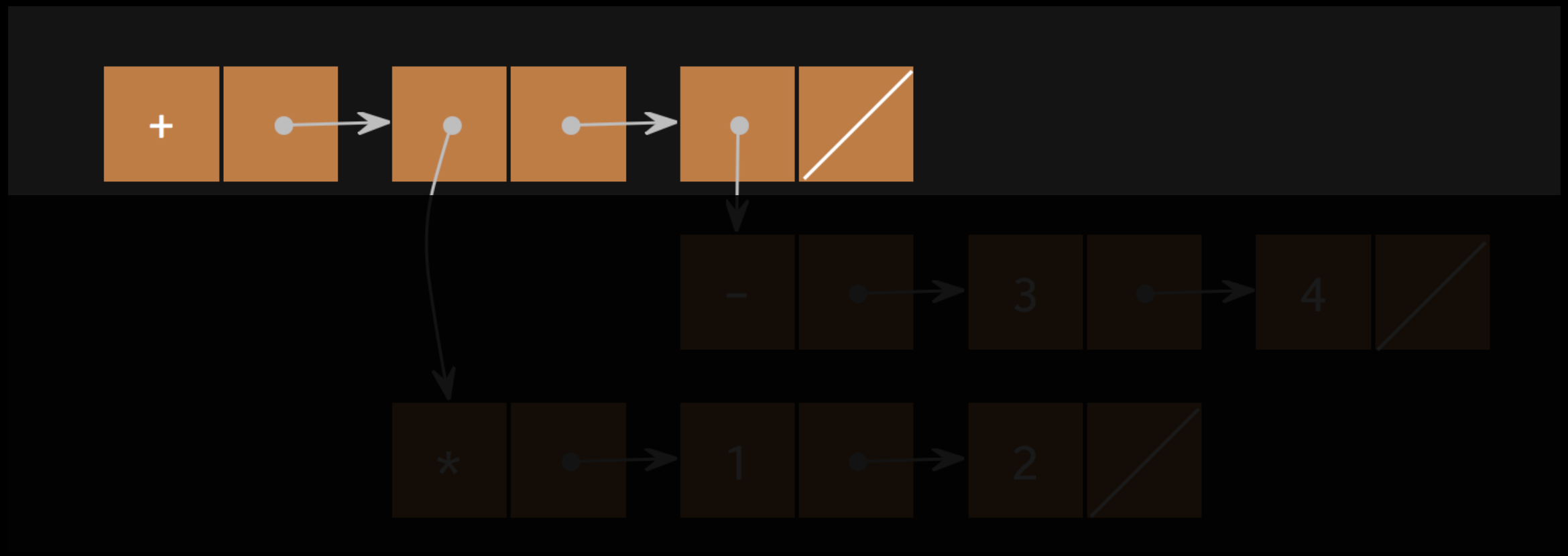
```
> (+ (* 1 2) (- 3 4))
1
```

```
Pair('+',
    Pair(Pair('*', Pair(1, Pair(2))),
        Pair(Pair('-', Pair(3, Pair(4))), nil)))
```

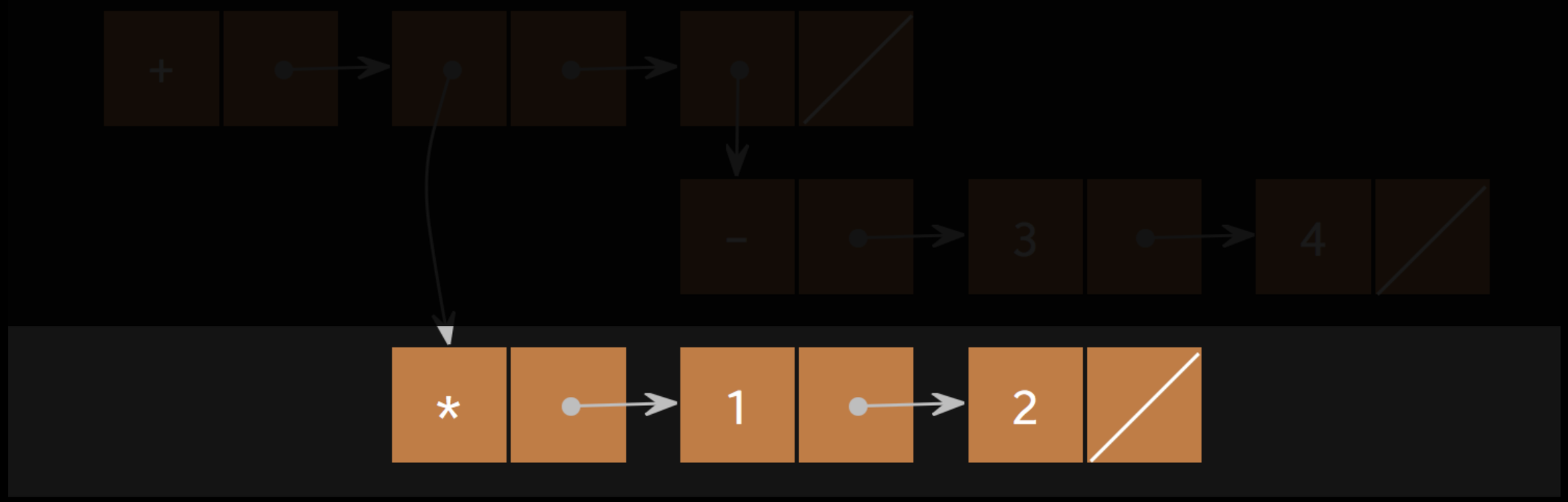
```
(+ (* 1 2) (- 3 4))
```

```
(+ (* 1 2) (- 3 4))
```

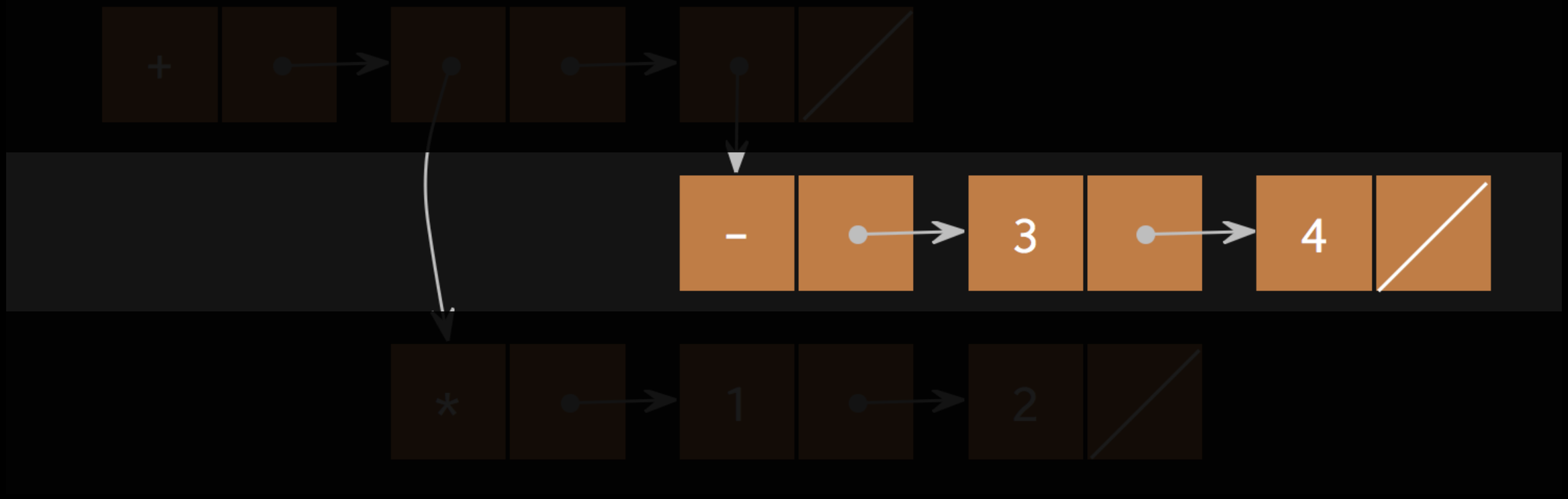




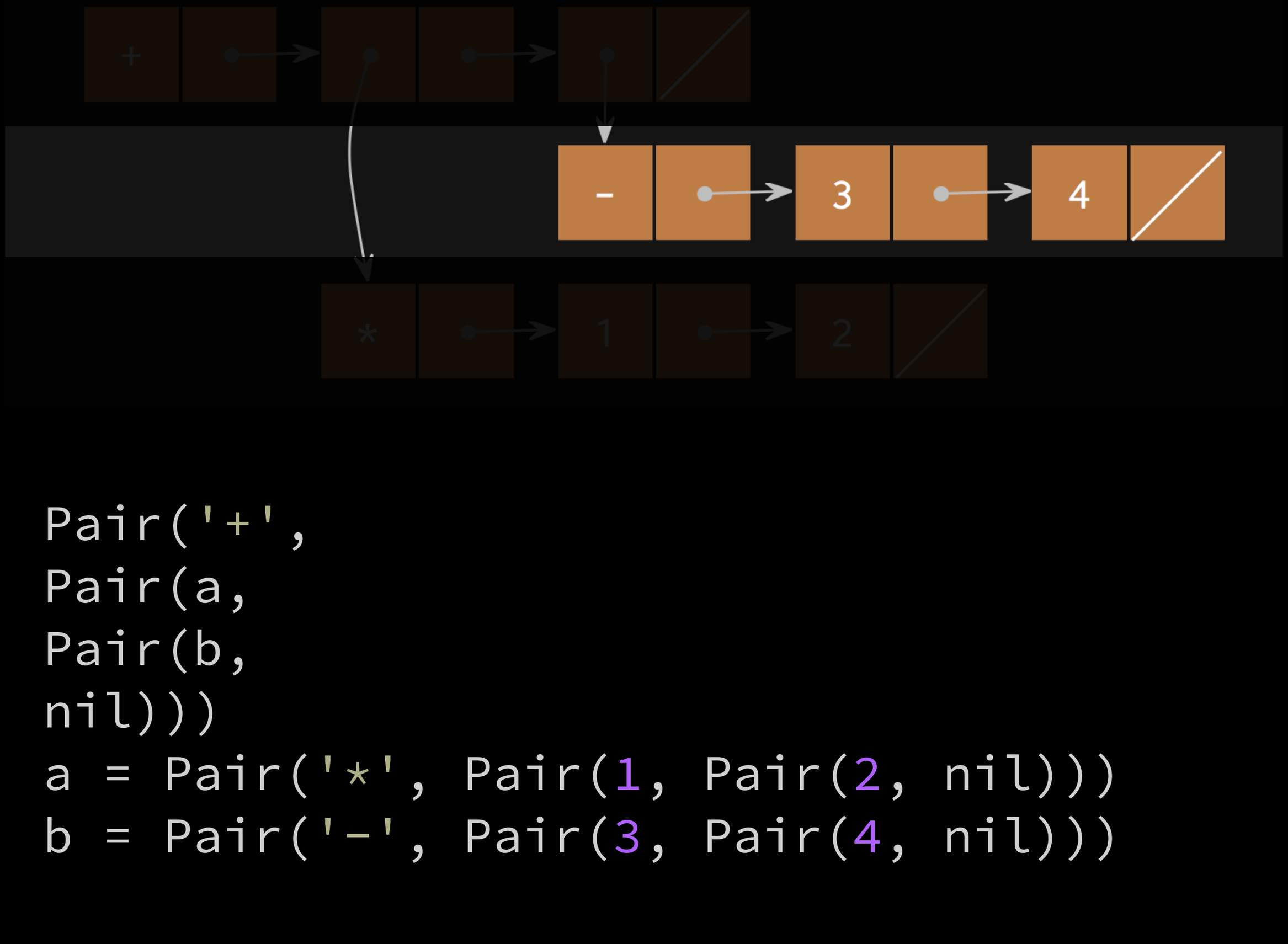

```
Pair('+', Pair(a, Pair(b, nil)))
```



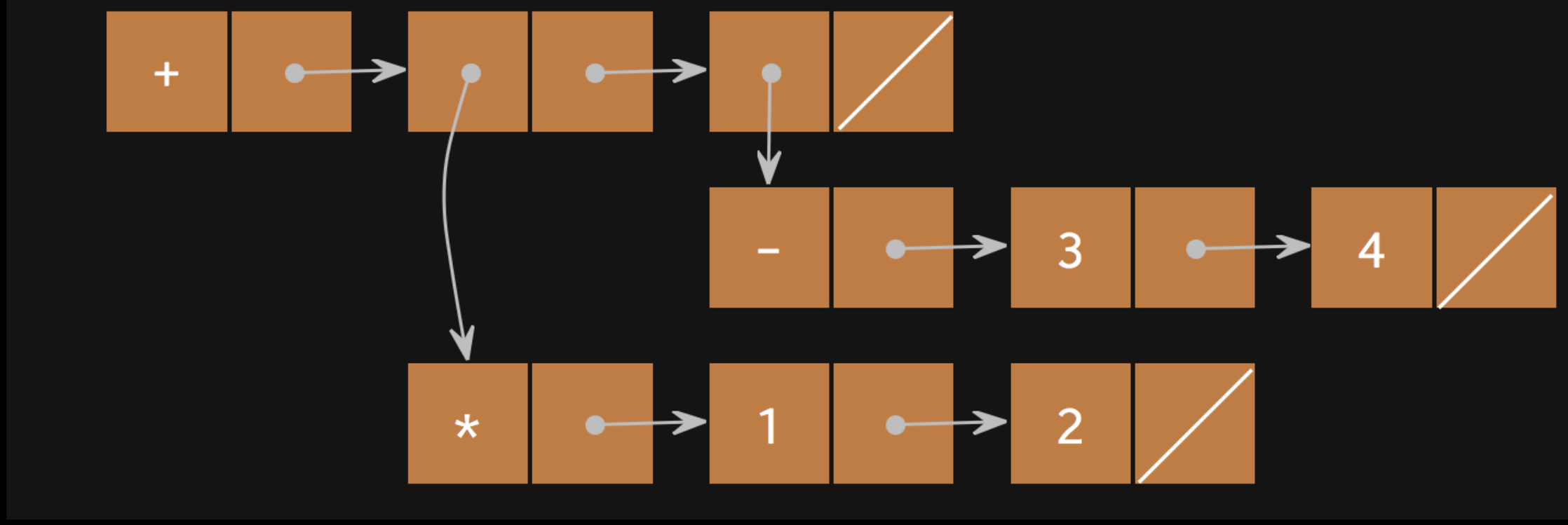

```
Pair('+', Pair(a, Pair(b, nil)))
a = Pair('*', Pair(1, Pair(2, nil)))
```




```
Pair('+', Pair(a, Pair(b, nil)))
a = Pair('*', Pair(1, Pair(2, nil)))
b = Pair('-', Pair(3, Pair(4, nil)))
```




```
Pair('+',
Pair(a,
Pair(b,
nil)))
a = Pair('*', Pair(1, Pair(2, nil)))
b = Pair('-', Pair(3, Pair(4, nil)))
```




```
Pair('+',
Pair(Pair('*', Pair(1, Pair(2, nil))),
Pair(Pair('-', Pair(3, Pair(4, nil))),
nil)))
a =
b =
```

Calculator Evaluation

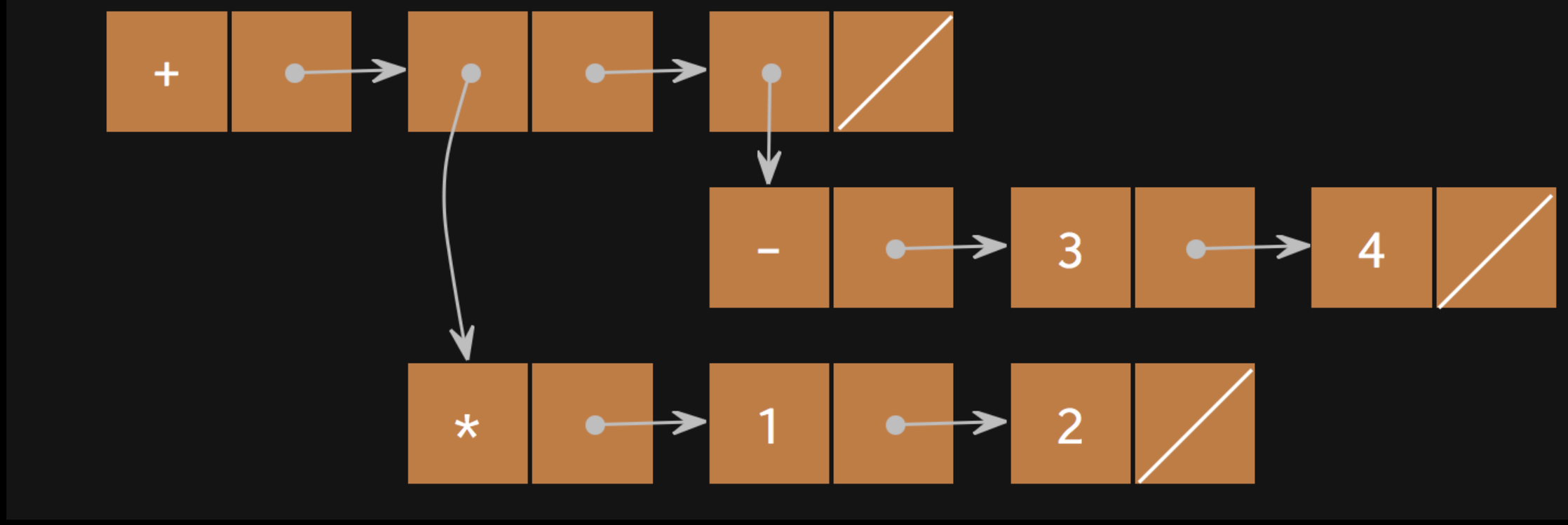
Putting our `Pair` to good use

- **Evaluate** the operator
- **Evaluate** the operands
- **Apply** the operator to the operands

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(
                  calc_eval(exp.first),
                  list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Primitive expression
        return exp
```

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(
                  calc_eval(exp.first),
                  list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Primitive expression
        return exp
```

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(
                calc_eval(exp.first),
                list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Primitive expression
        return exp
```




```
Pair('+',
Pair(Pair('*', Pair(1, Pair(2, nil))),
Pair(Pair('-', Pair(3, Pair(4, nil))),
nil)))
```

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(
                calc_eval(exp.first),
                list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Primitive expression
        return exp
```

```
1 def calc_eval(exp):
2     if isinstance(exp, Pair):
3         return calc_apply(
4                 calc_eval(exp.first),
5                 list(exp.second.map(calc_eval)))
6     elif exp in OPERATORS:
7         return OPERATORS[exp]
8     else: # Primitive expression
9         return exp
```

Evaluate the operator

```
1 def calc_eval(exp):
2     if isinstance(exp, Pair):
3         return calc_apply(
4                    calc_eval(exp.first),
5                    list(exp.second.map(calc_eval)))
6     elif exp in OPERATORS:
7         return OPERATORS[exp]
8     else: # Primitive expression
9         return exp
```

Evaluate the operands

```
1 def calc_eval(exp):
2     if isinstance(exp, Pair):
3         return calc_apply(
4                 calc_eval(exp.first),
5                 list(exp.second.map(calc_eval)))
6     elif exp in OPERATORS:
7         return OPERATORS[exp]
8     else: # Primitive expression
9         return exp
```

Apply the operator to the operands

```
1 def calc_eval(exp):
2     if isinstance(exp, Pair):
3         return calc_apply(
4                 calc_eval(exp.first),
5                 list(exp.second.map(calc_eval)))
6     elif exp in OPERATORS:
7         return OPERATORS[exp]
8     else: # Primitive expression
9         return exp
```

Operators (like '+') and primitives (like 3.1416)

How did we get here?

Started with an expression:

```
> (+ (* 1 2) (- 3 4))
```

Converted to a pair representation:

```
Pair('+',
    Pair(Pair('*', Pair(1, Pair(2))),
        Pair(Pair('-', Pair(3, Pair(4))), nil)))
```

Used evaluation rules to obtain result:

1

Tail Recursion

- Life is about trade offs*
- Recursive calls => non constant space**
- Tail recursive calls => constant space***

* and disclaimers
** usually
*** only if you put in the work

Tail Recursion

Necessary conditions

- **Tail context** - the "last thing" you do in an expression
- **Tail call** - a recursive call in a tail context
- Constant number of frames if **all recursive calls are tail calls**
- If you depend on other **non tail-recursive functions,** this might not be sufficient

Tail Recursion

Valid tail contexts

```
1 (define (fact n)
2   (if (= n 0)
3       1
4       (* n (fact (- n 1)))))
```

Tail Recursion

Valid tail contexts

```
1 (define (fact n)
2   (if (= n 0)
3       1
4       (* n (fact (- n 1)))))
```

Tail Recursion

Valid tail contexts

```
1 (define (fact n)
2   (if (= n 0)
3       1
4       (* n (fact (- n 1)))))
```

Last thing we do is this multiplication

Tail Recursion

More space efficient fact

```
1 (define (fact n)
2   (define (fact-tail n result)
3     (if (= n 0)
4         result
5         (fact-tail (- n 1) (* n result))))
6   (fact-tail n 1))
```

Tail Recursion

More space efficient fact

```
1 (define (fact n)
2   (define (fact-tail n result)
3     (if (= n 0)
4         result
5         (fact-tail (- n 1) (* n result))))
6   (fact-tail n 1))
```

Tail Recursion

More space efficient fact

```
1 (define (fact n)
2   (define (fact-tail n result)
3     (if (= n 0)
4         result
5         (fact-tail (- n 1) (* n result))))
6   (fact-tail n 1))
```

```scheme
(define (fact n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1)
                   (* n result))))
  (fact-tail n 1))
```

```scheme
(define (fact n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1)
                   (* n result))))
  (fact-tail n 1))
```

```python
def fact(n):
    result = 1
    while n > 0:
        n, result = n - 1, result * n
    return result
```

Thanks to Kavi Gupta for this visualization idea

```scheme
(define (fact n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1)
                   (* n result))))
  (fact-tail n 1))
```

```python
def fact(n):
    result = 1
    while n > 0:
        n, result = n - 1, result * n
    return result
```

Thanks to Kavi Gupta for this visualization idea