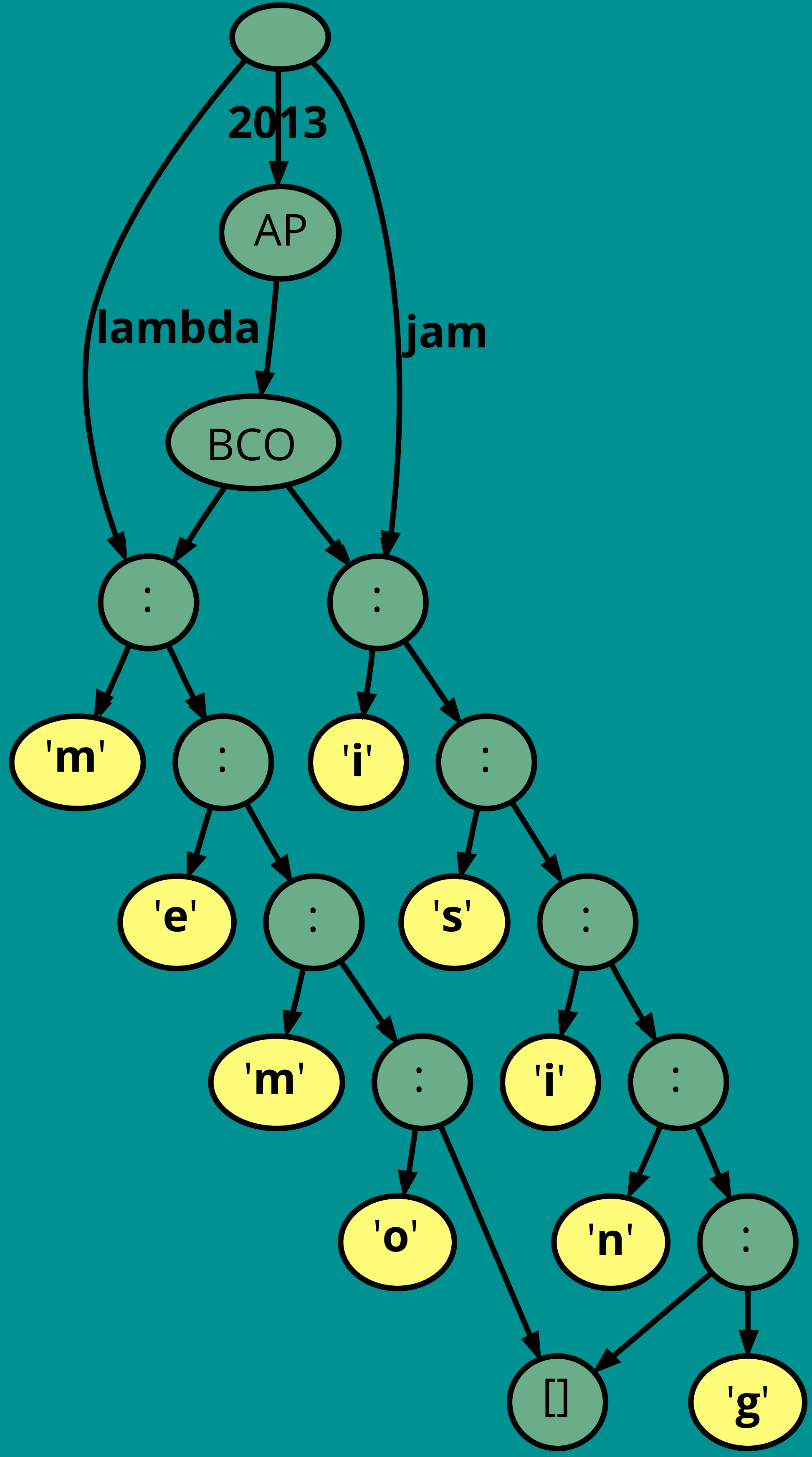


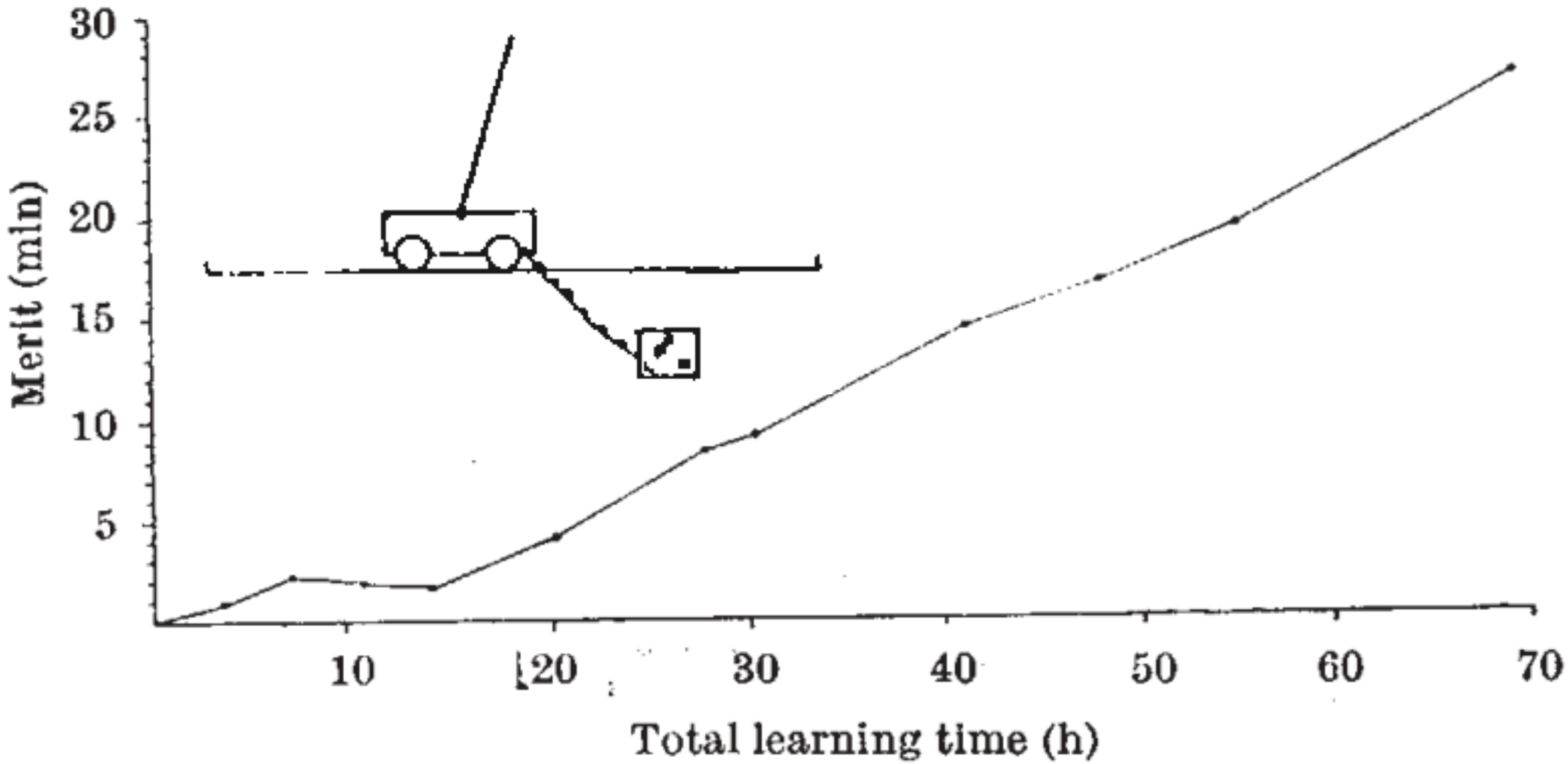
Algorithms

Andrew Newman

"It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution."

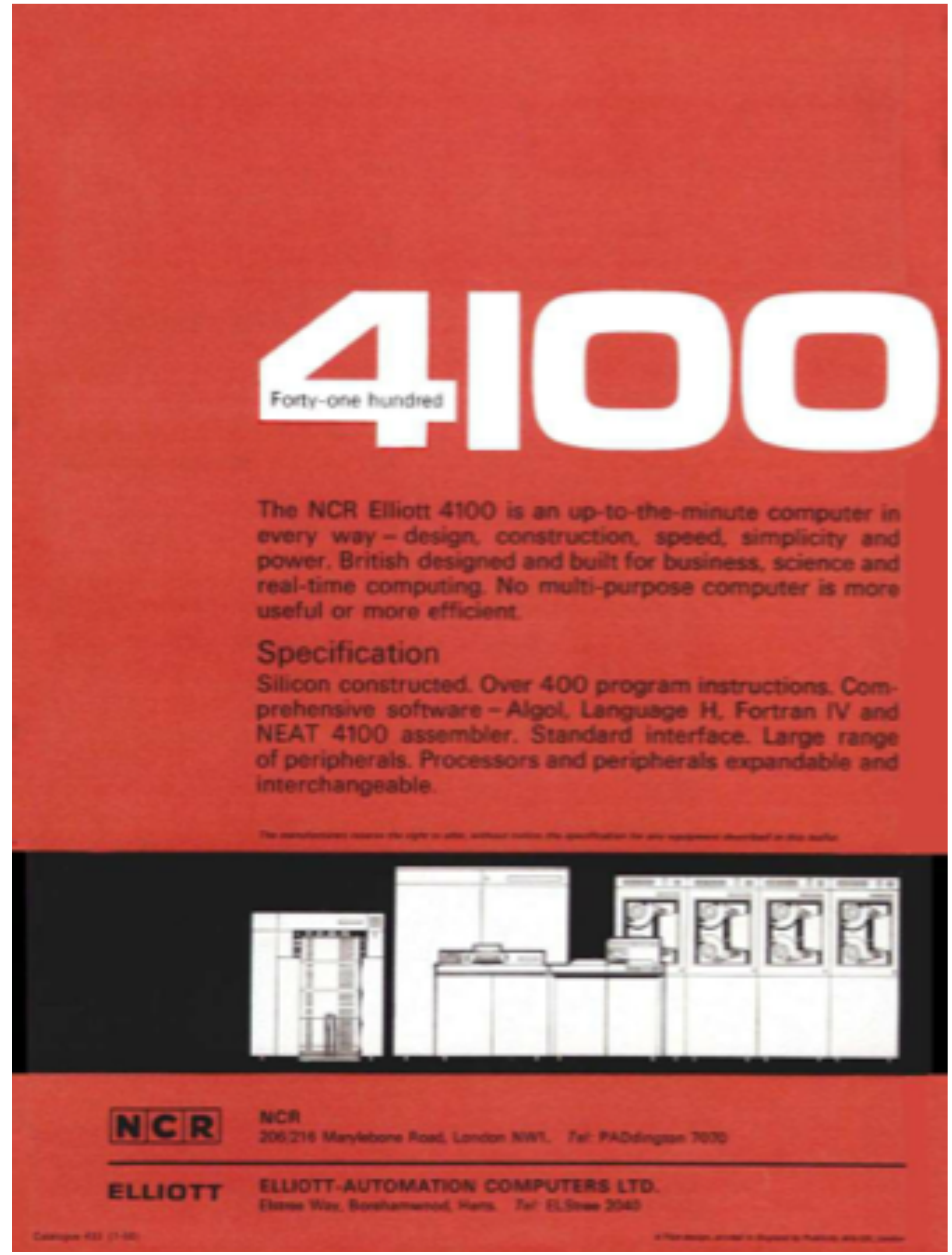
Donald Michie, Nature, 1968

“Memo” Functions



Elliott 4120

Algol, H, Fortran
Assembler
6μs read/write, 12μs add
24 bits, 64K words



Endowing “memo”

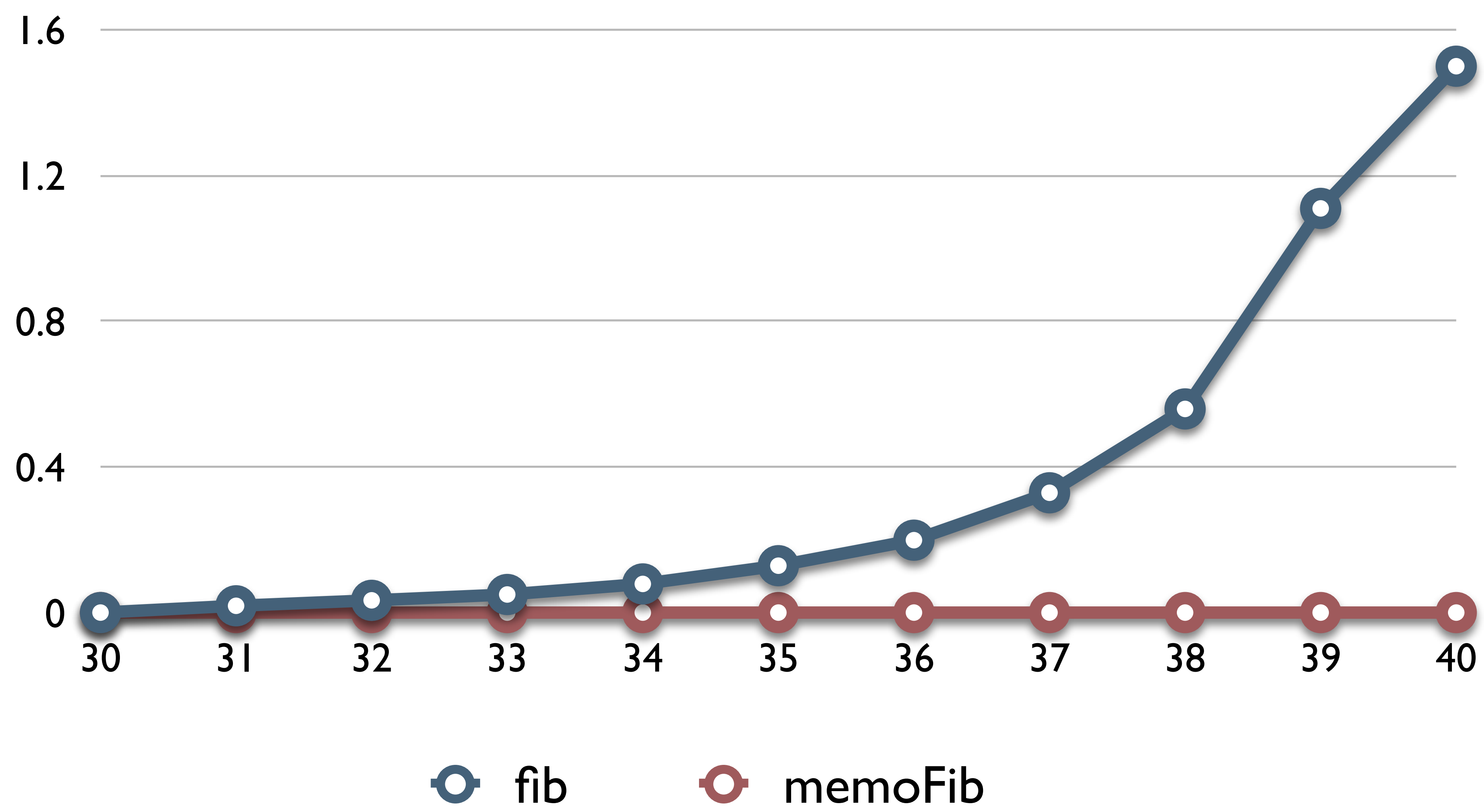
```
function fact(n);
  if n = 0 then 1 else n * fact (n - 1)
  endif
enddefine;

newmemo (fact , 20) -> fact;
```

Running “trace fact;”

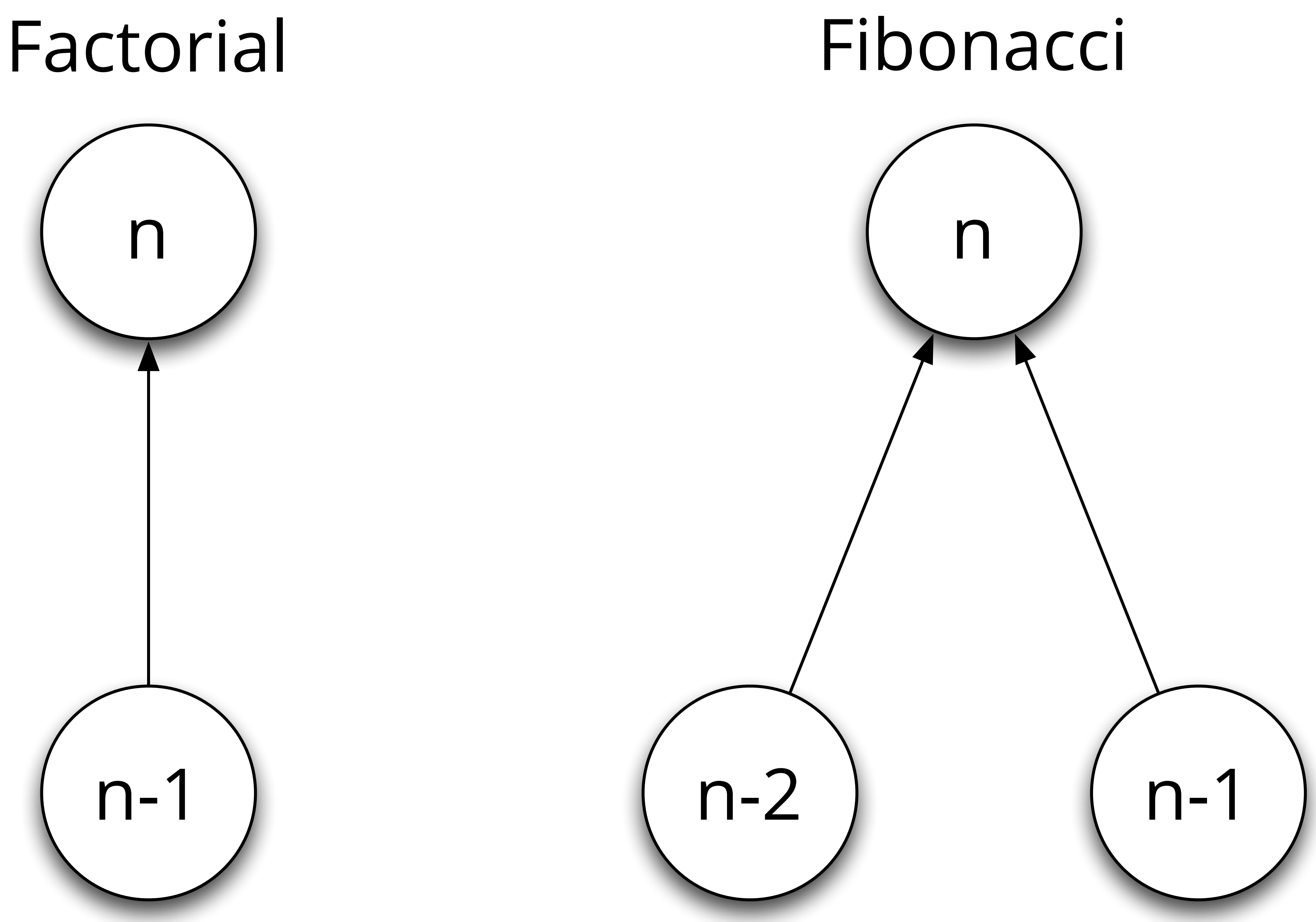
```
fact(2) =>
> fact 2
!> fact 1
!!> fact 0
!!< fact 1
!< fact 1
< fact 2
** 2
```

```
fact(4) =>
> fact 4
!> fact 3
!!> fact 2
!!< fact 2
!< fact 6
< fact 24
** 24
```

Space for Time



Bad Examples



comp.lang.lisp

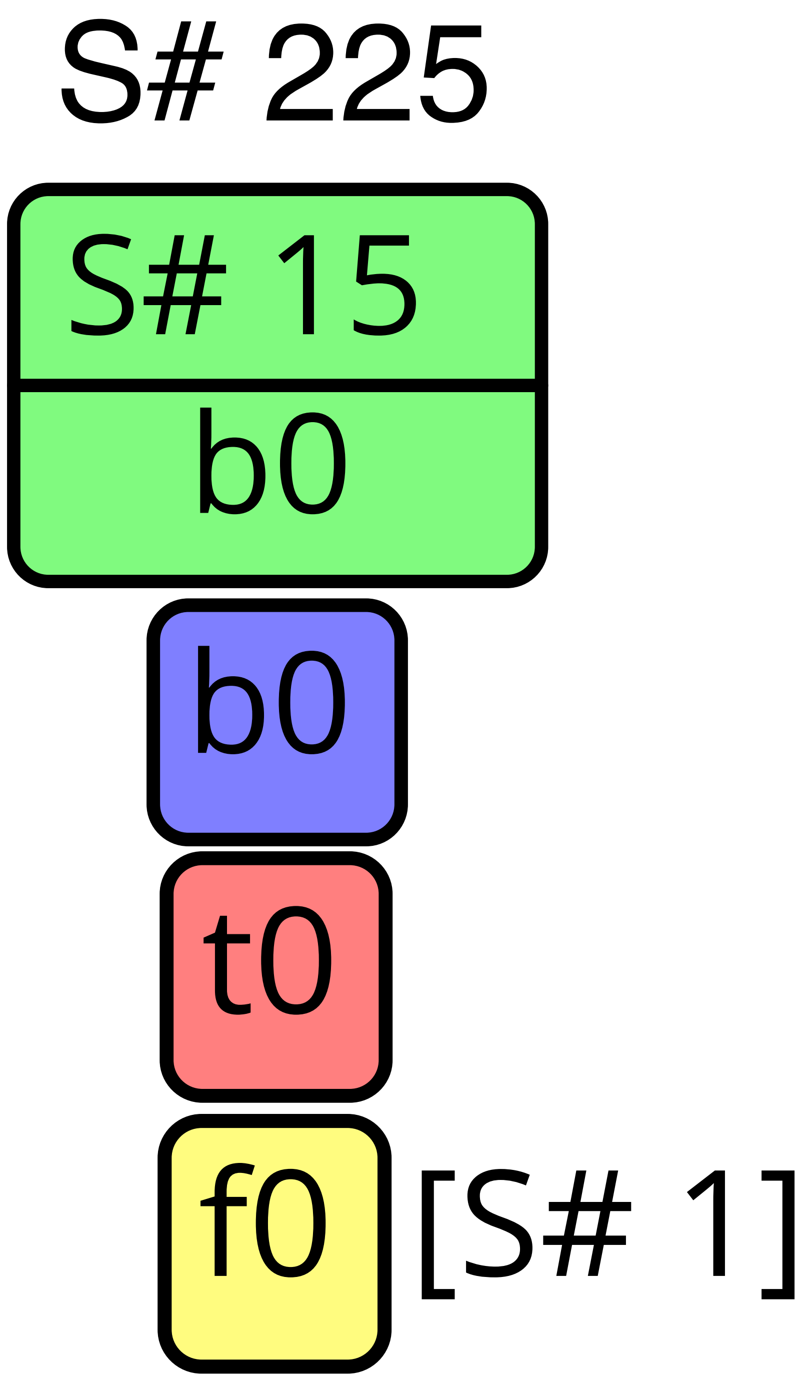
"I see memoisation as necessarily a system capability and not something a user can write...in a discussion in Glasgow with Simon Peyton-Jones and Cordelia Hall, it was pointed out to me that memoisation could affect semantics. This it clearly would do if you memoised a non-strict function, such as a constructor."

Robin Popplestone, 1998

Evaluation

Strict	Non-Strict
Applicative Order	Normal Order
Call by Value	Call by Name
Call by Reference	Call by Need

Heap Objects



CONSTR

NAMED

LINK

THUNK

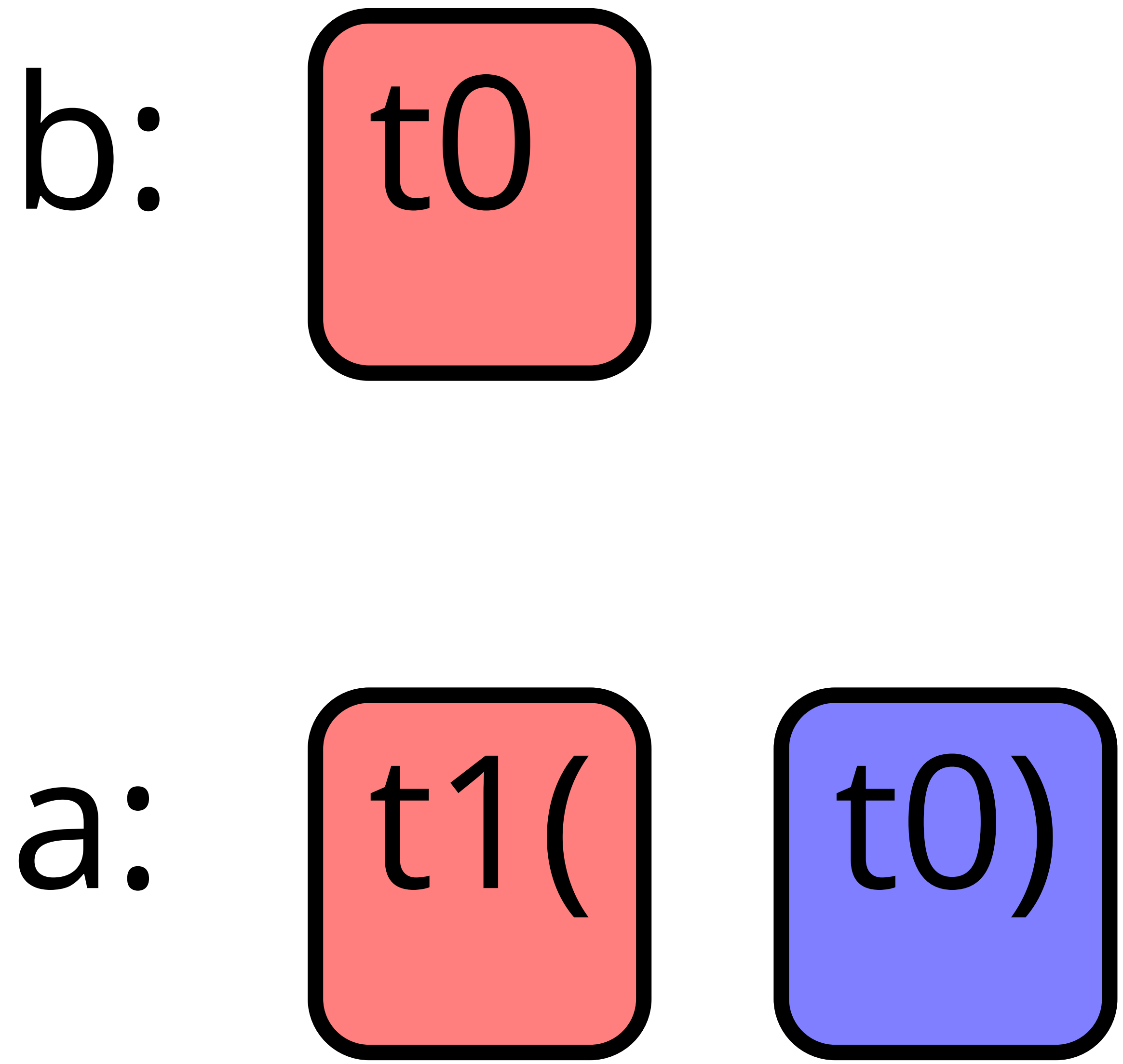
FUNCTION

Thunking

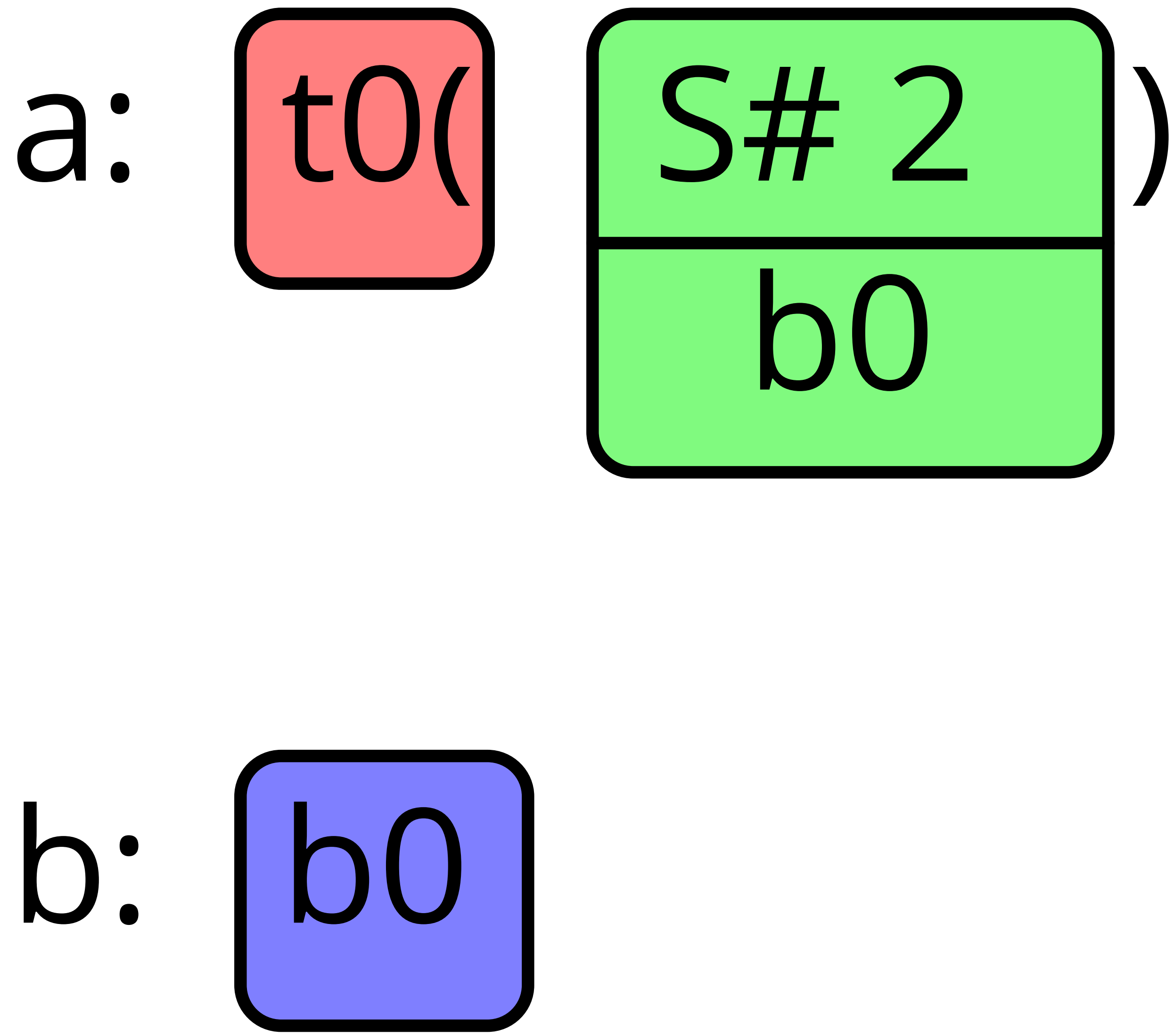
Values

```
> let b = 1 + 1
> let a = 1 + b
```

:view a, :view b



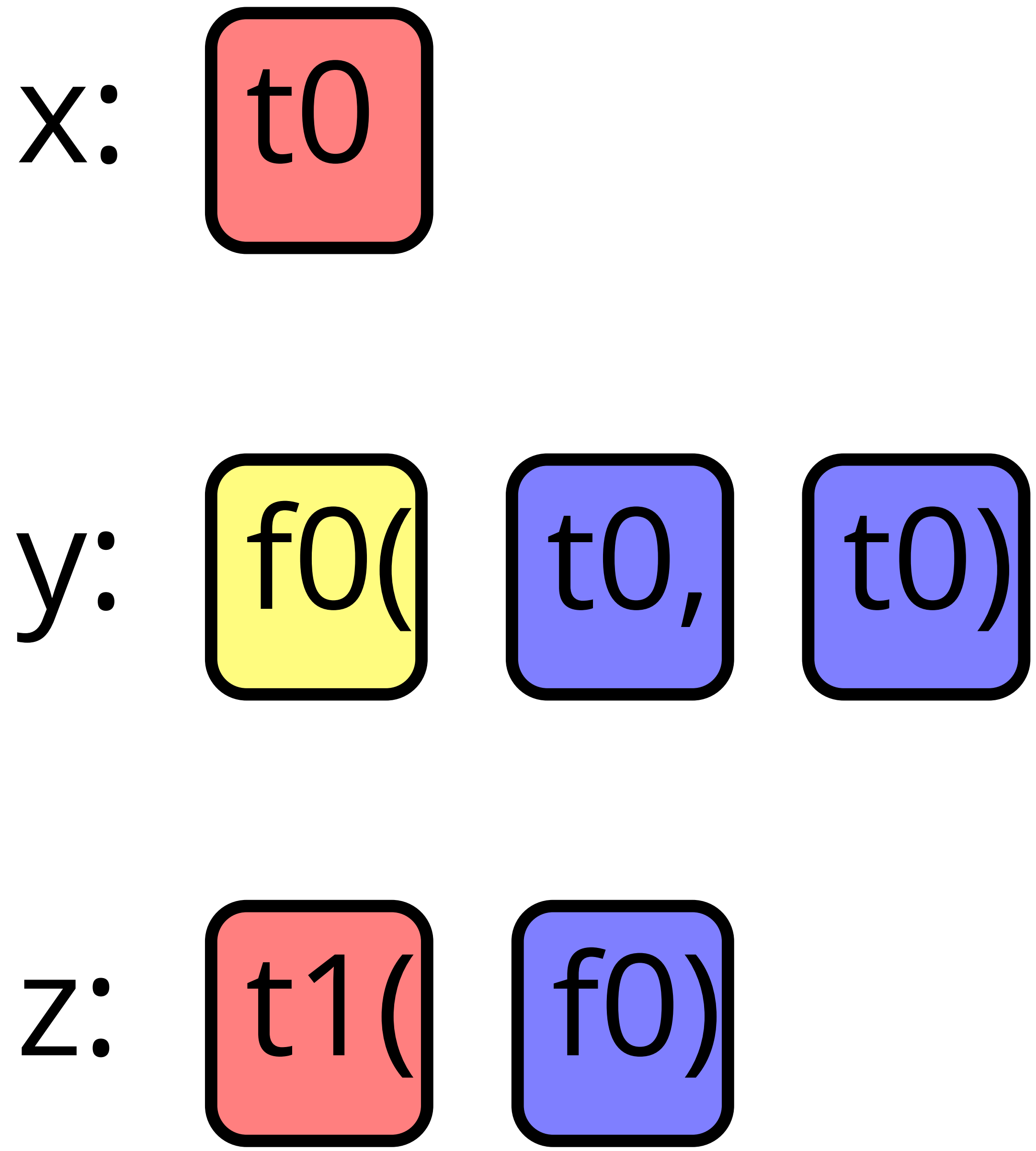
> b



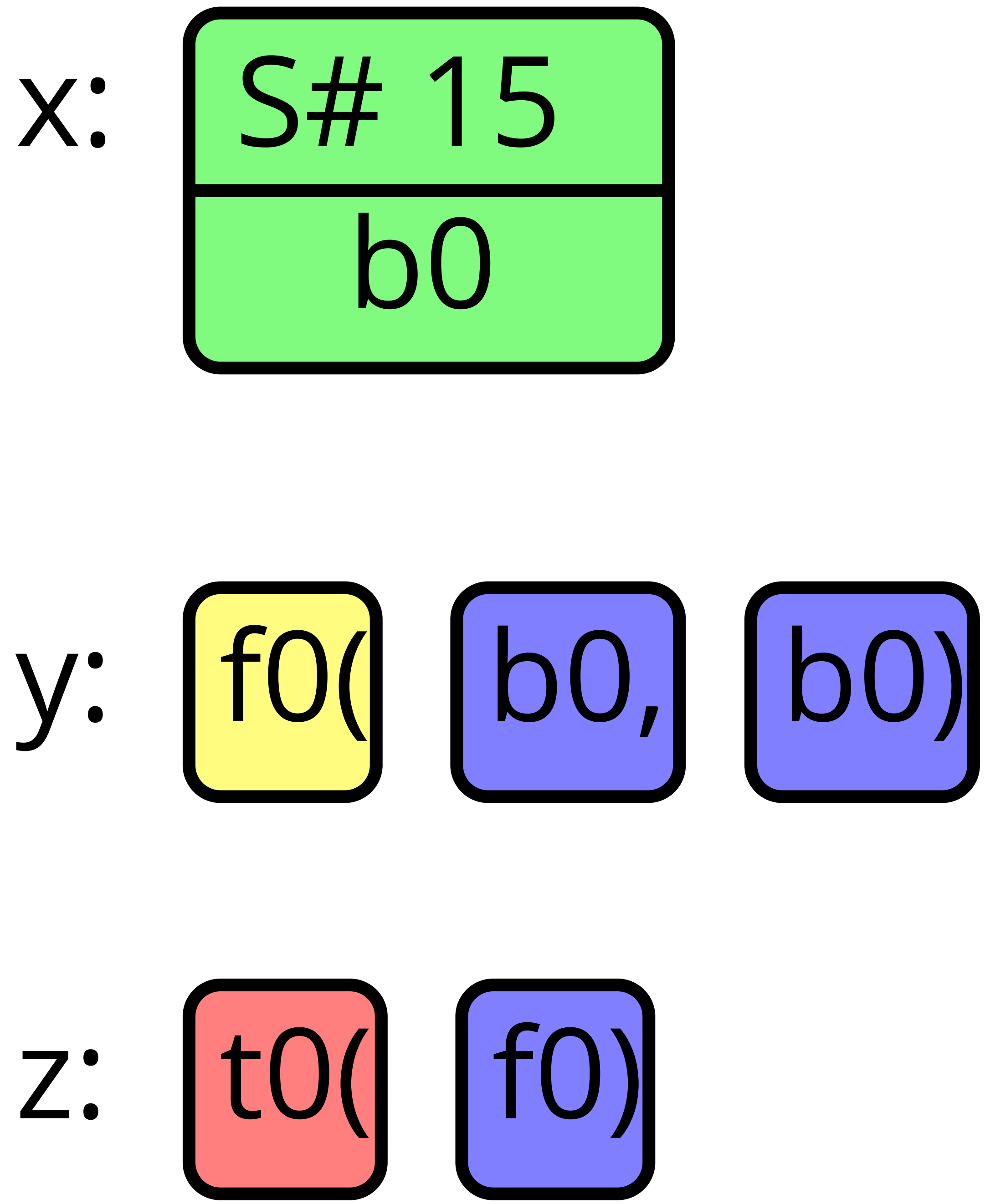
Functions

```
> let x = sum [1..5]
> let y f = x `f` x
> let z = y (*)
> z
225
```

:view x, :view y, :view z



:eval t0



:eval t0

x:

S# 15
b0

y: | f0(| b0, | b0) |

z: S# 225

Data Structures

```
> let x = Data.IntMap.fromList $
  zip [1,2] [1..]
> let y = Data.IntMap.fromList $
  zip [6,7] [6..]
> let z = Data.IntMap.union x y
```

:view x, :view y

x: t0

y: t1

Data.IntMap.union x y: t2(t1, t0)

:eval t0

x: Bin 0 2 (Tip 1 (S# 1)) (Tip 2 (S# 2))

b0

y: t0

Data.IntMap.union x y: t1(t0, b0)

:eval t0

x:

Bin 0 2 (Tip 1 (S# 1)) (Tip 2 (S# 2))
b0

y:

Bin 6 1 (Tip 6 (S# 6)) (Tip 7 (S# 7))
b1

Data.IntMap.union x y:

t0(	b1,	b0)

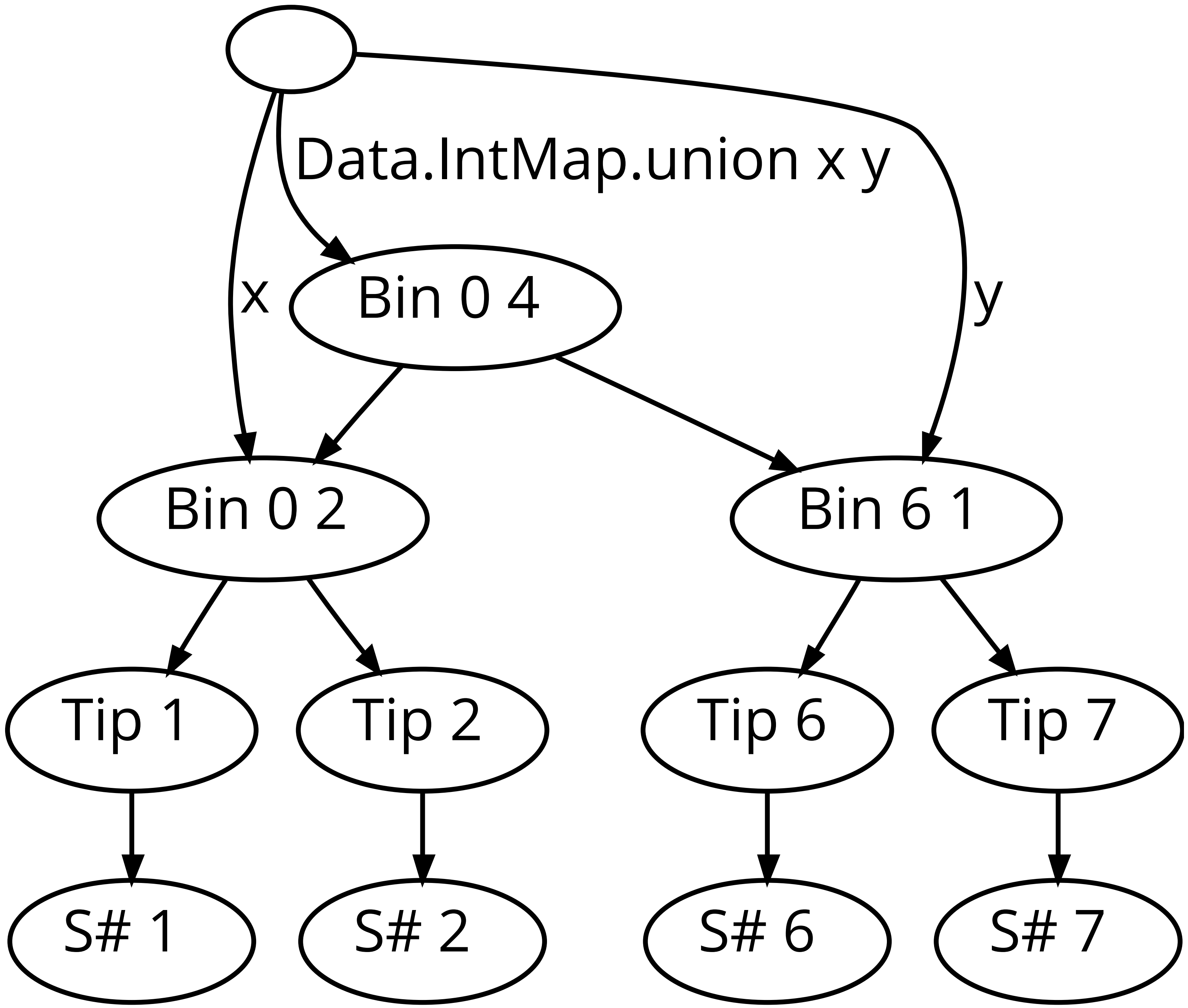
:eval t0

x: Bin 0 2 (Tip 1 (S# 1)) (Tip 2 (S# 2))
b0

y: Bin 6 1 (Tip 6 (S# 6)) (Tip 7 (S# 7))
b1

Data.IntMap.union x y: Bin 0 4 b0 b1

:switch



Data.IntMap.union x y
x
y
Bin 0 4
Bin 0 2
Bin 6 1
Tip 1
Tip 2
Tip 6
Tip 7
S# 1
S# 2
S# 6
S# 7

Letters and Numbers

Based on the English “Countdown” and the original French series “Des chiffres et des lettres”.

Given a list of numbers: 2, 5, 8, 10, 11, 17, 24, 50

Target: 53280

Use: +, *, /, -

Answer:11 + 5 * 8 - 17 * 24 * 10 * 2

Ingredients

```
subseqs [x] = [[x]]
subseqs (x:xs) = xss ++ [x] : map (x:) xss
  where xss = subseqs xs

> subseqs [1,2]
[[1],[2],[1,2]]

> subseqs [1,2,3]
[[1],[2],[3],[1,2],[2,3],[1,3],[1,2,3]]
```

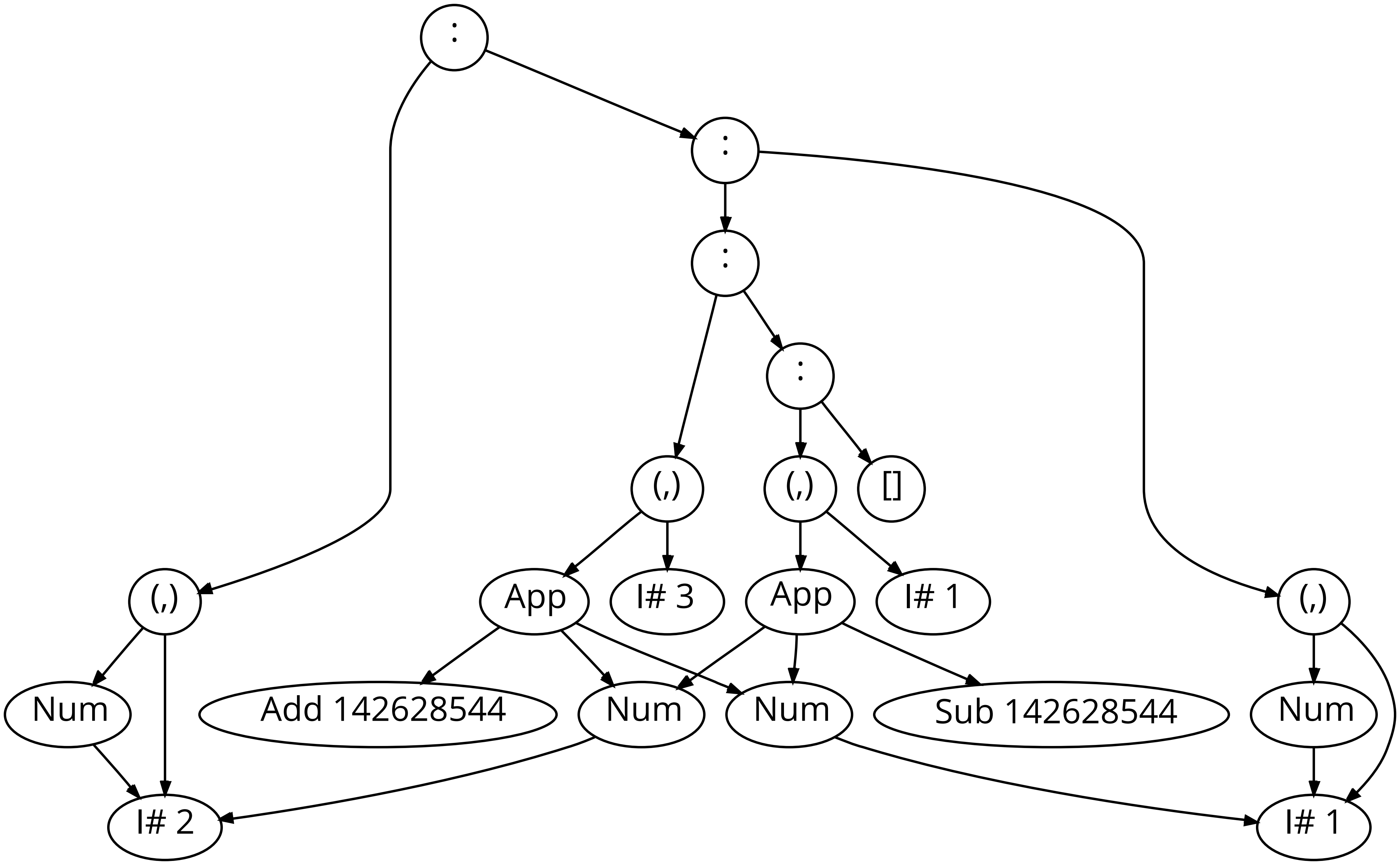
Ingredients

```
mkExprs :: [Int] -> [(Expr, Value)]
mkExprs [x] = [(Num x, x)]
mkExprs xs = [ev | (ys, zs) <- unmerges xs,
              ev1 <- mkExprs ys,
              ev2 <- mkExprs zs,
              ev <- combine ev1 ev2]

> mkExprs [1,2]
[(App Add (Num 1) (Num 2),3),(App Sub (Num 2) (Num
1),1)]
```

Ingredients

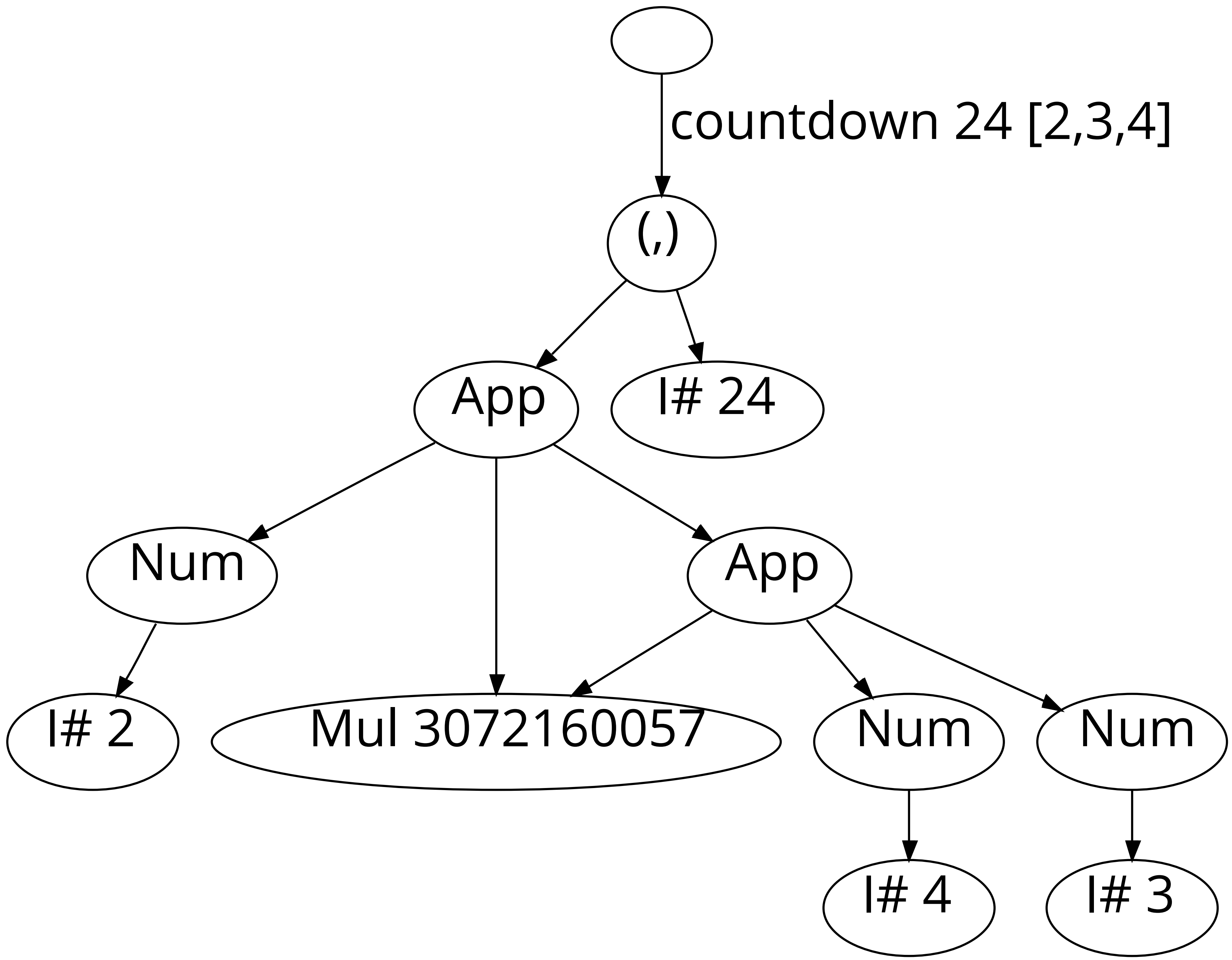
```
data Op = Add | Sub | Mul | Div
legal Add v1 v2 = True
legal Sub v1 v2 = (v2 < v1)
legal Mul v1 v2 = True
legal Div v1 v2 = (v1 `mod` v2 == 0)

countdown n = nearest n . concatMap
  mkExprs . subseqs
```

concatMap mkExprs . subseqs [1,2]

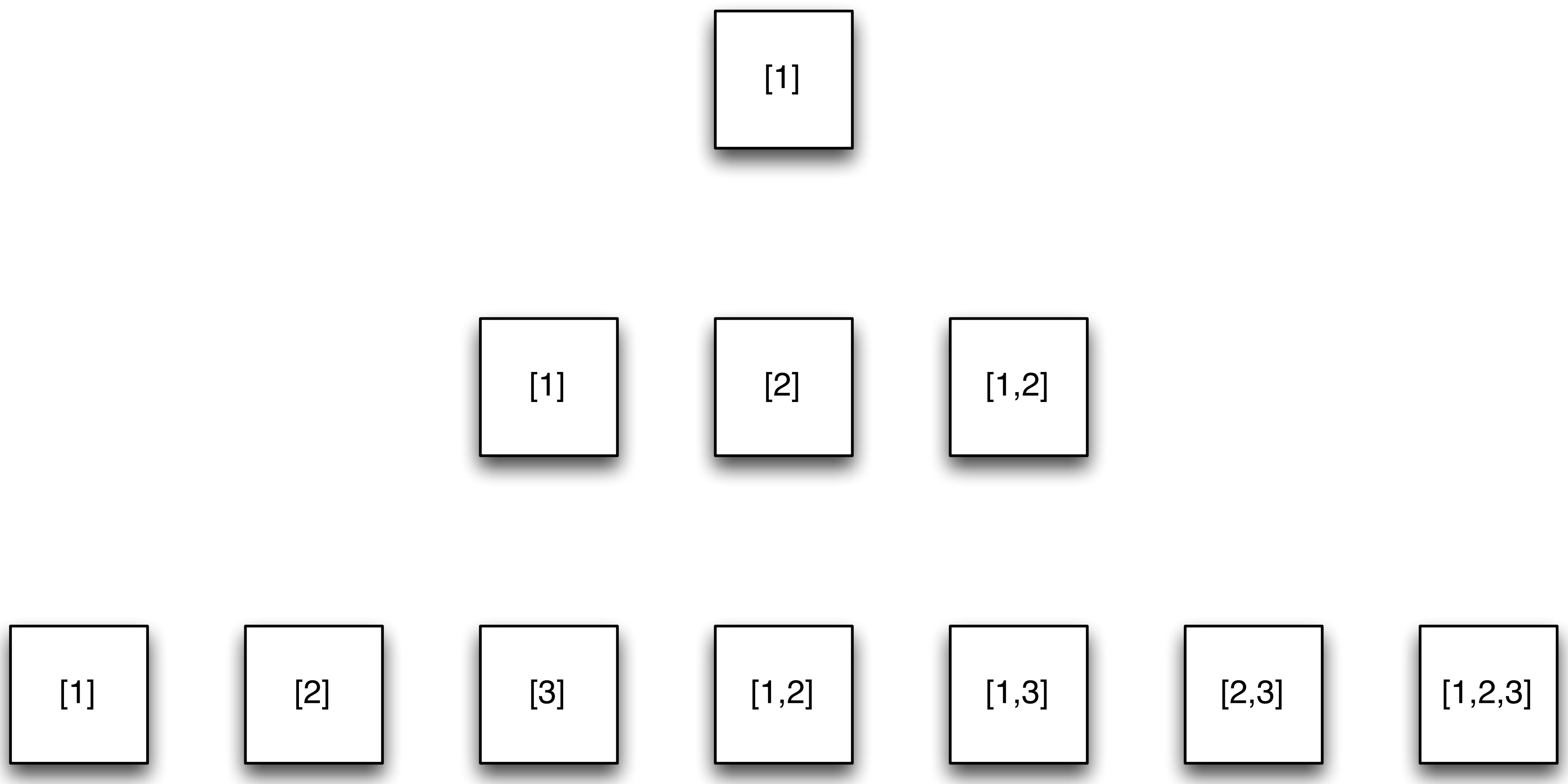


[(Num 2,2),(Num 1,1),(App Add (Num 1) (Num 2),3),
(App Sub (Num 2) (Num 1),1)]

countdown 24 [2,3,4]

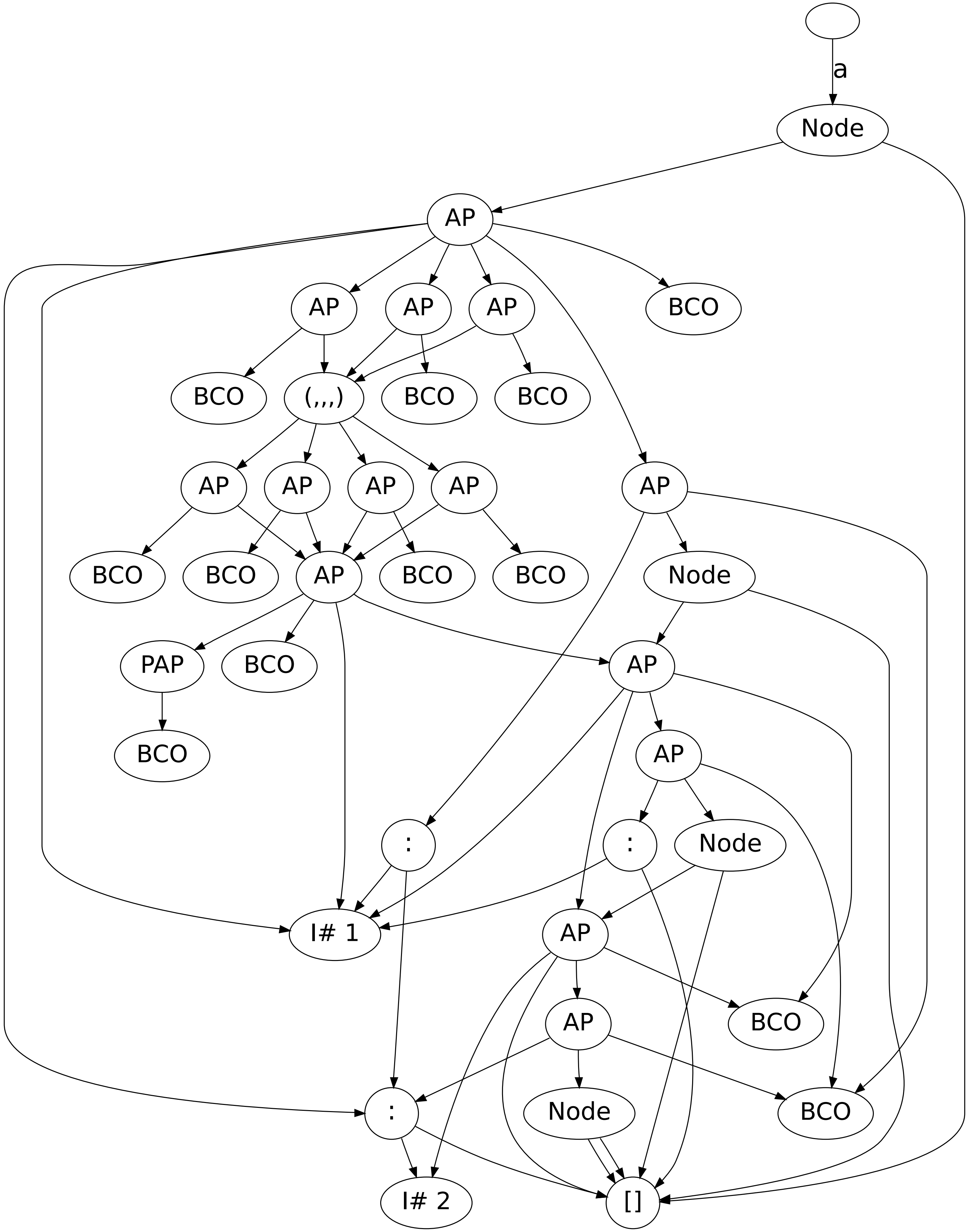


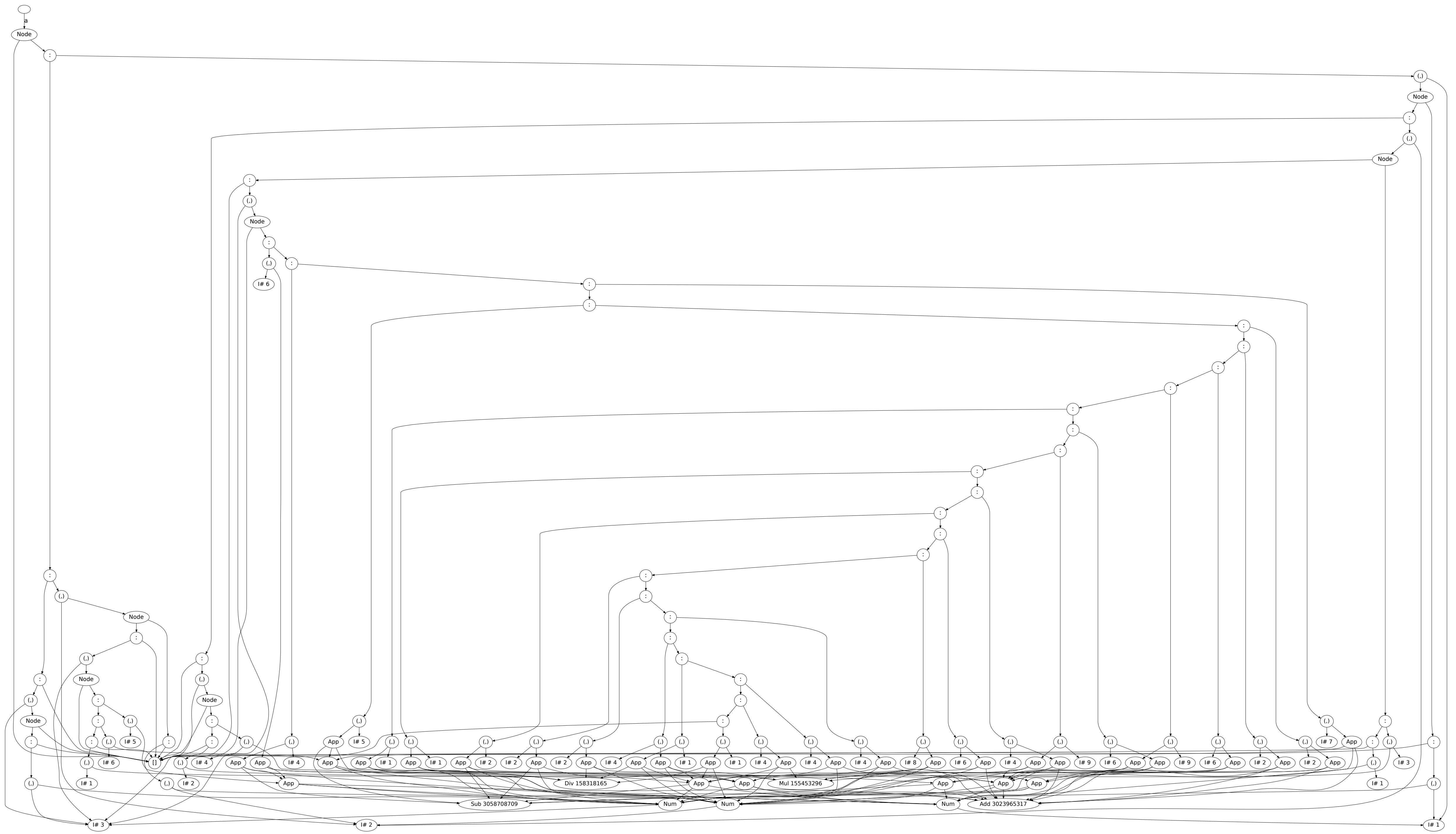
Redundancy



Trie

```
data Trie a = Node a [(Int, Trie a)]

type Memo = Trie [(Expr, Value)]
fetch :: Memo -> [Int] -> [(Expr, Value)]
store :: [Int] -> [(Expr, Value)] ->
  Memo -> Memo

countdown n = nearest n . extract .
 memoise . subseqs
  subseqs
```


a
Node
AP
AP
AP
AP
BCO
BCO
(,,,)
BCO
BCO
AP
AP
AP
AP
AP
BCO
BCO
AP
BCO
BCO
Node
PAP
BCO
AP
BCO
AP
:
:
Node
I# 1
AP
AP
BCO
:
Node
BCO
I# 2
[]

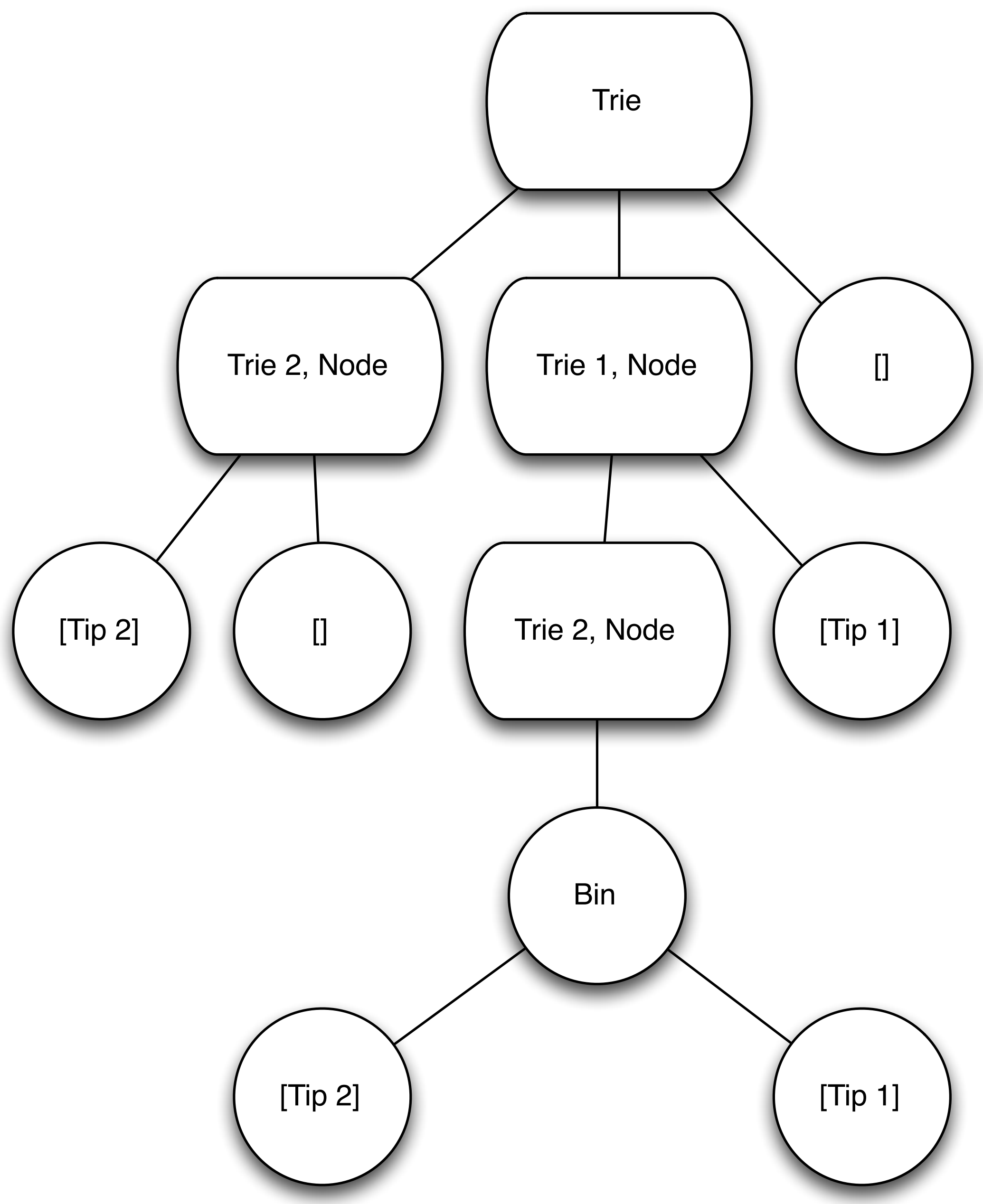

a
Node
:
(,)
[]
App
Num
I# 1
I# 2
I# 3
I# 4
I# 5
I# 6
I# 7
I# 8
I# 9
Sub 3058708709
Div 158318165
Mul 155453296
Add 3023965317

Skeleton Tree

```
data Trie a = Node a [(Int, Trie a)]
type Memo = Trie [Tree]
data Tree = Tip Int | Bin Tree Tree
fetch :: Memo -> [Int] -> [Tree]
store :: [Int] -> [Tree] -> Memo -> Memo

countdown n = nearest n .
  concatMap toExprs . extract . memoise .
  subseqs
```

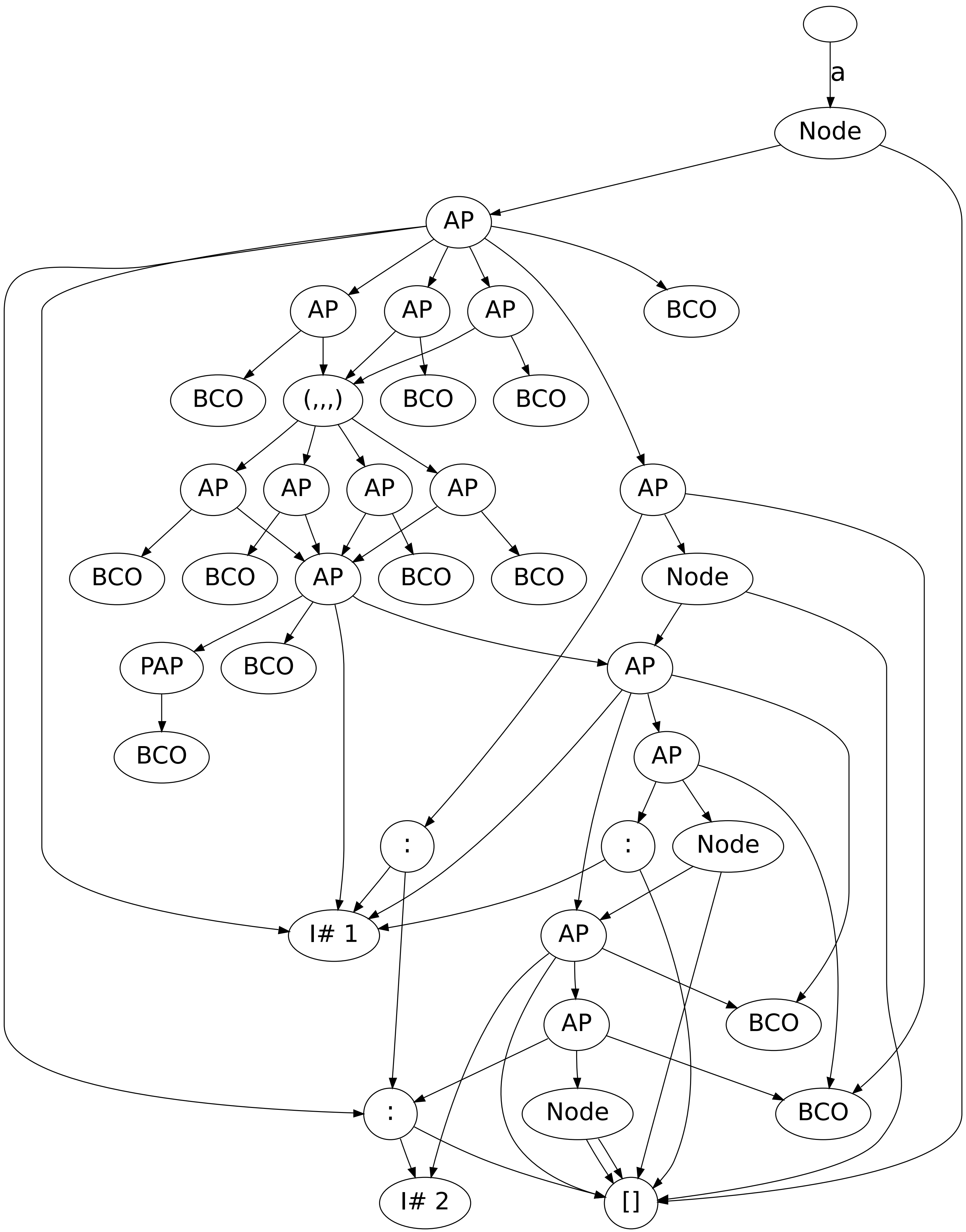
Ingredients

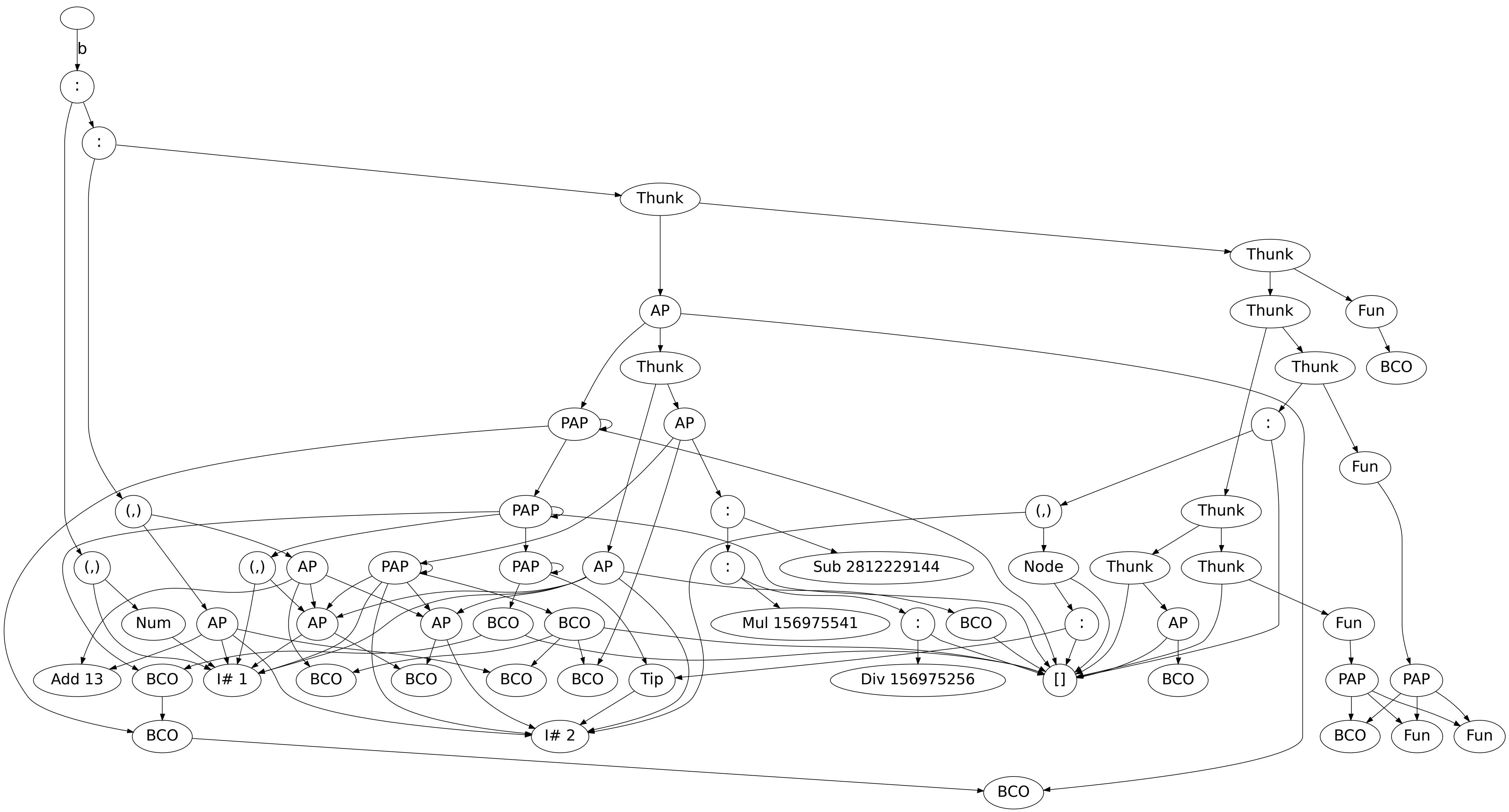
> memoise $ subseqs [1,2]

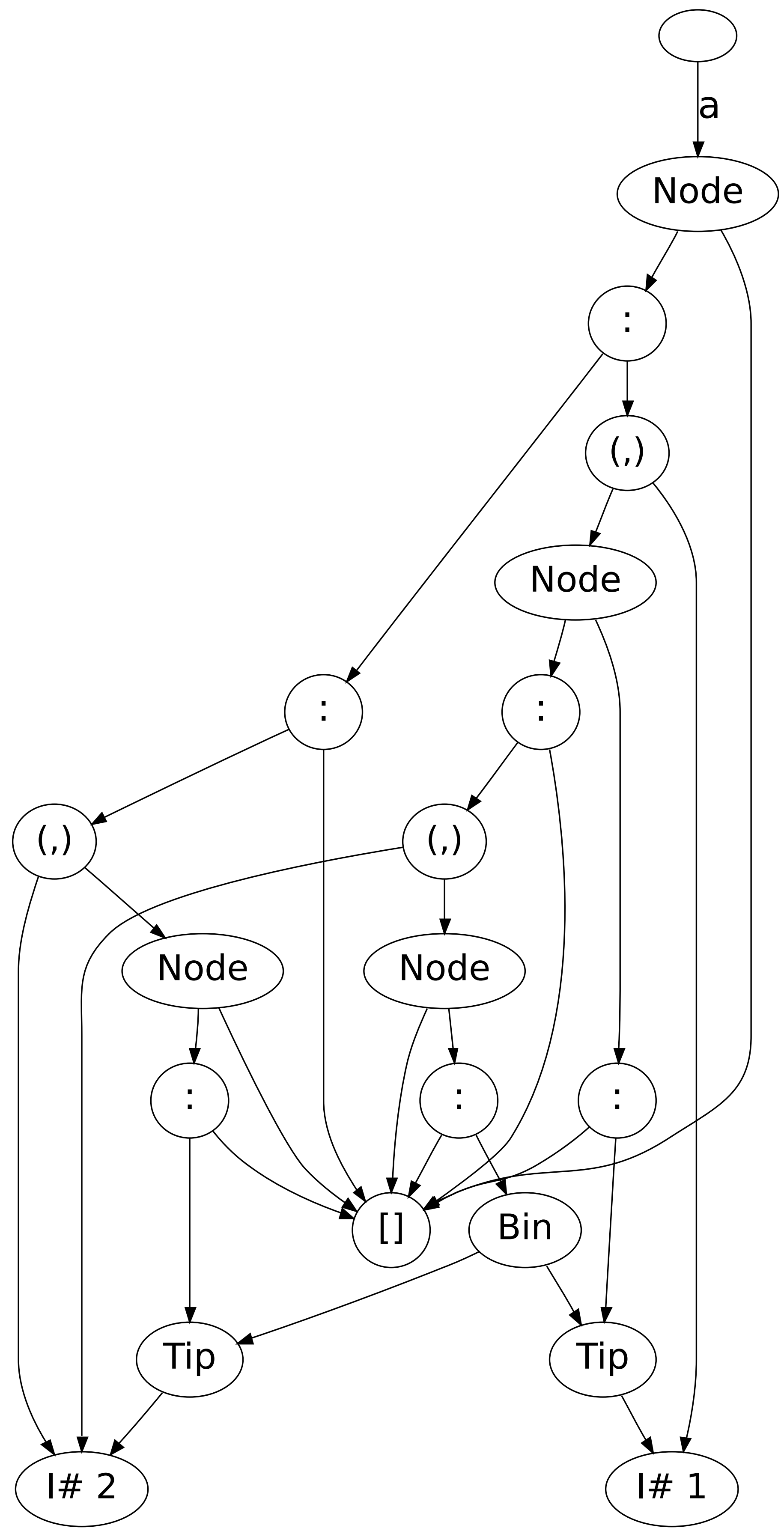


Ingredients

```
> extact $ memoise $ subseqs [1,2]
> [Tip 1,Bin (Tip 1) (Tip 2),Tip 2]
```


a
Node
AP
BCO
(,,,)
PAP
:
I# 1
I# 2
[]


b
:
:
Thunk
Thunk
AP
Thunk
Fun
Thunk
Thunk
BCO
PAP
AP
:
Fun
(,)
PAP
:
(,)
Thunk
(,)
(,)
AP
PAP
PAP
AP
:
Sub 2812229144
Node
Thunk
Thunk
Num
AP
AP
AP
BCO
BCO
Mul 156975541
:
BCO
:
AP
Fun
Add 13
BCO
I# 1
BCO
BCO
BCO
BCO
Tip
Div 156975256
[]
BCO
PAP
PAP
BCO
I# 2
BCO
Fun
Fun
BCO

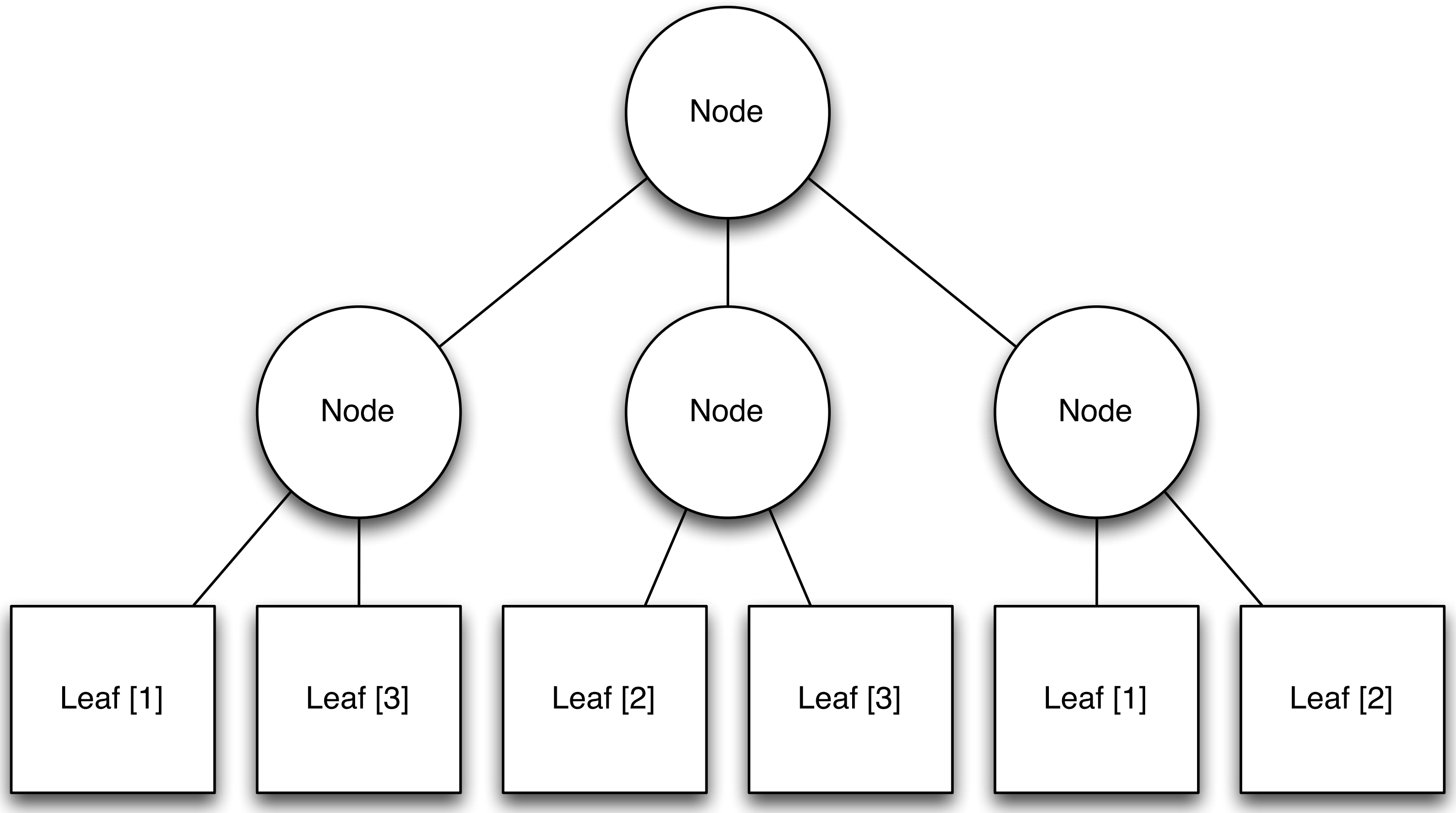

a
Node
:
(,)
Node
:
:
(,)
(,)
Node
Node
:
:
:
[]
Bin
Tip
Tip
I# 2
I# 1

Nexus

```
data Tree a = Leaf a | Node [Tree a]
data LTree a = LLeaf a |
  LNode a [LTree a]
treeToNexus :: Tree [a] -> LTree [ai]
treeToNexus = fill id recover
```

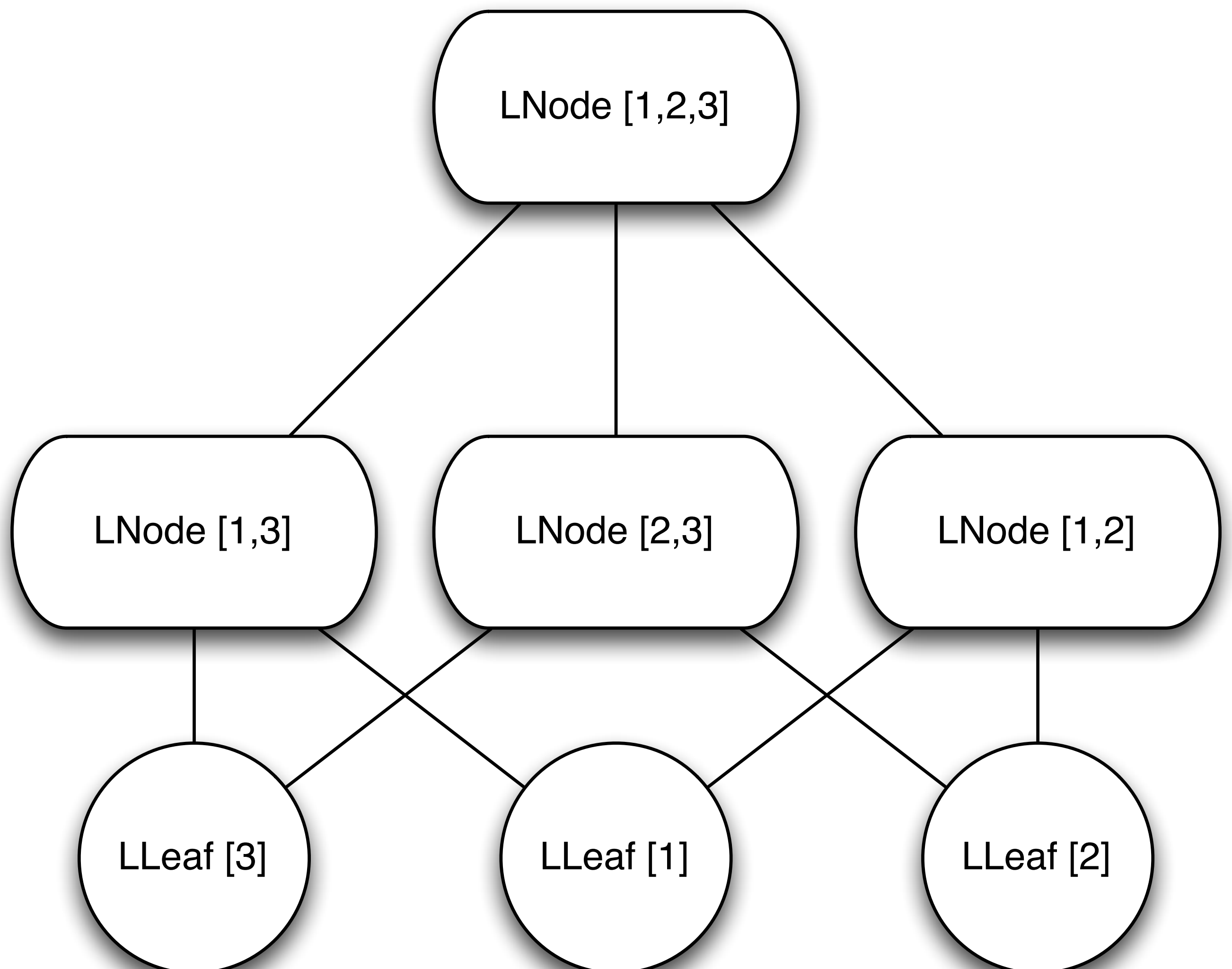
Ingredients

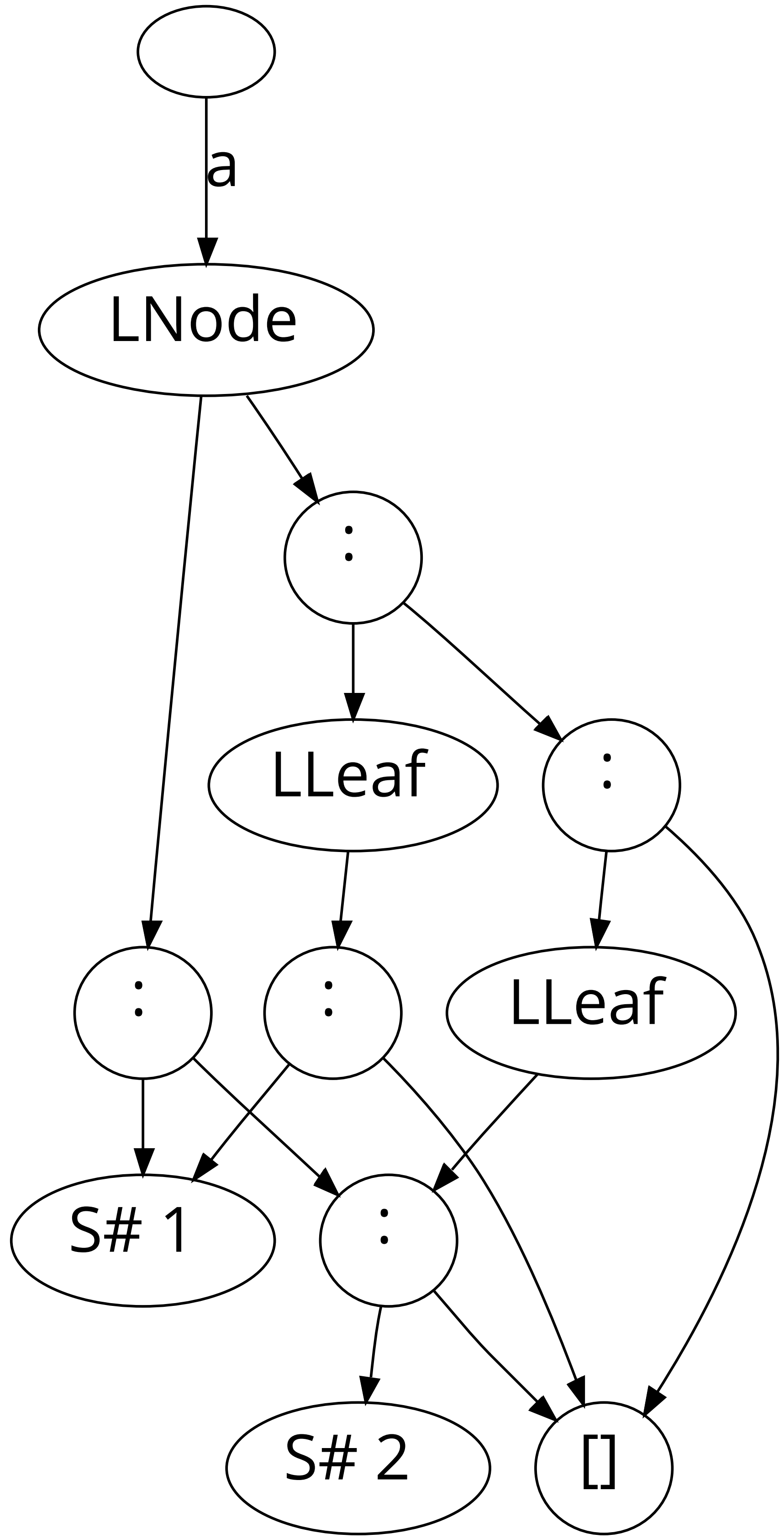
```
> minors [1,2,3]
[[1,2],[1,3],[2,3]]
> mkTree minors [1,2,3]
```



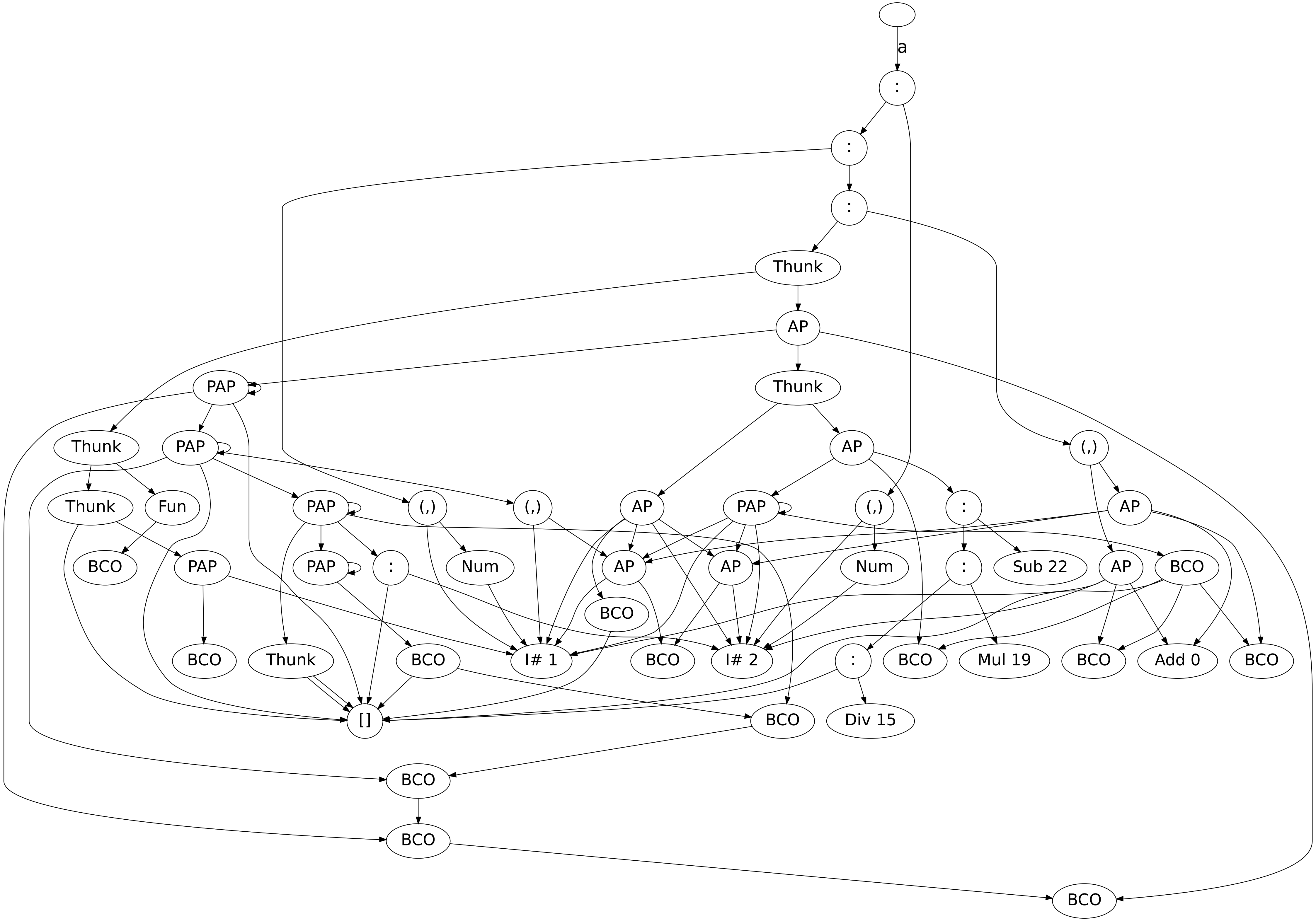
Ingredients

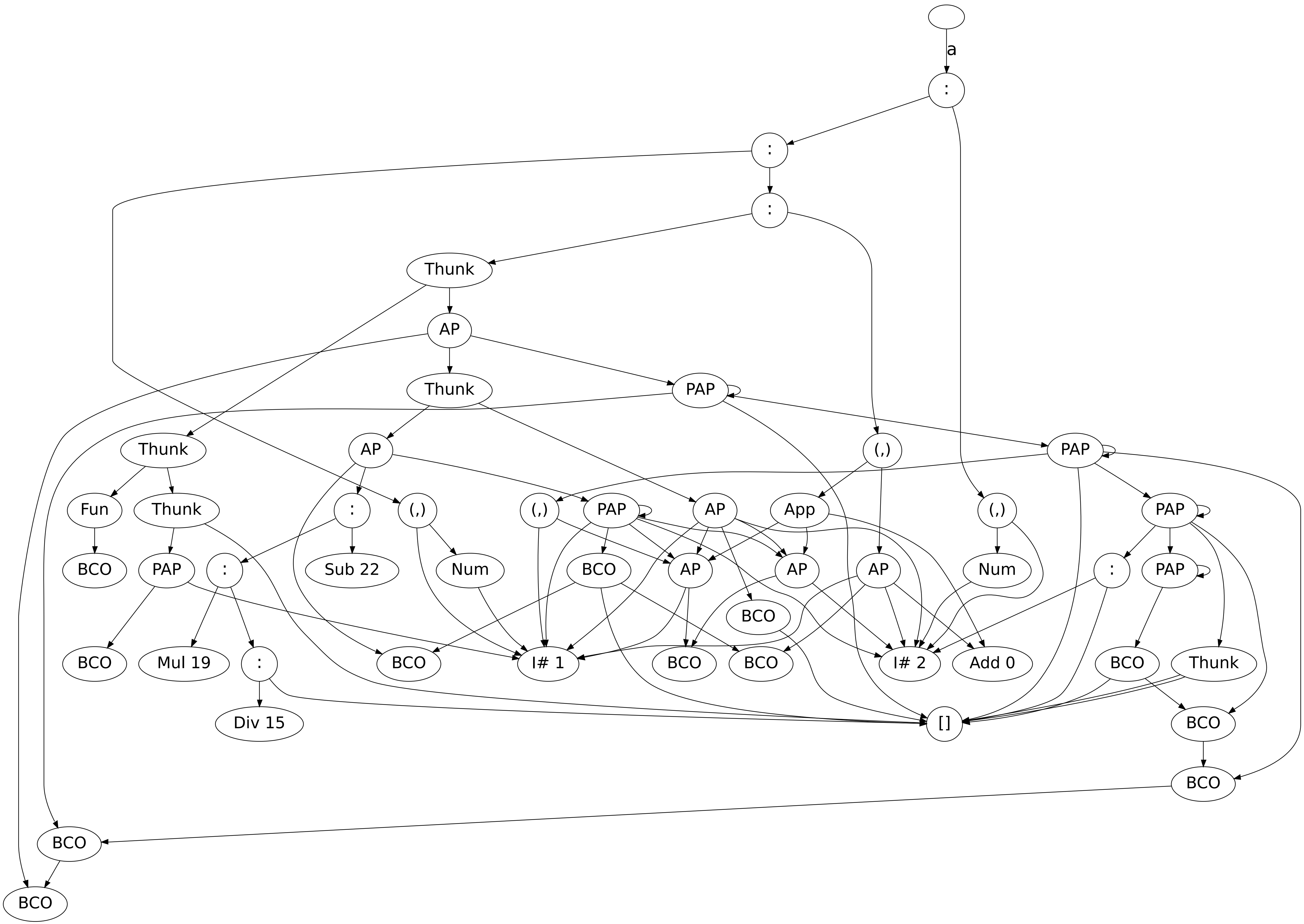
> treeToNexus $ mkTree minors [1,2,3]





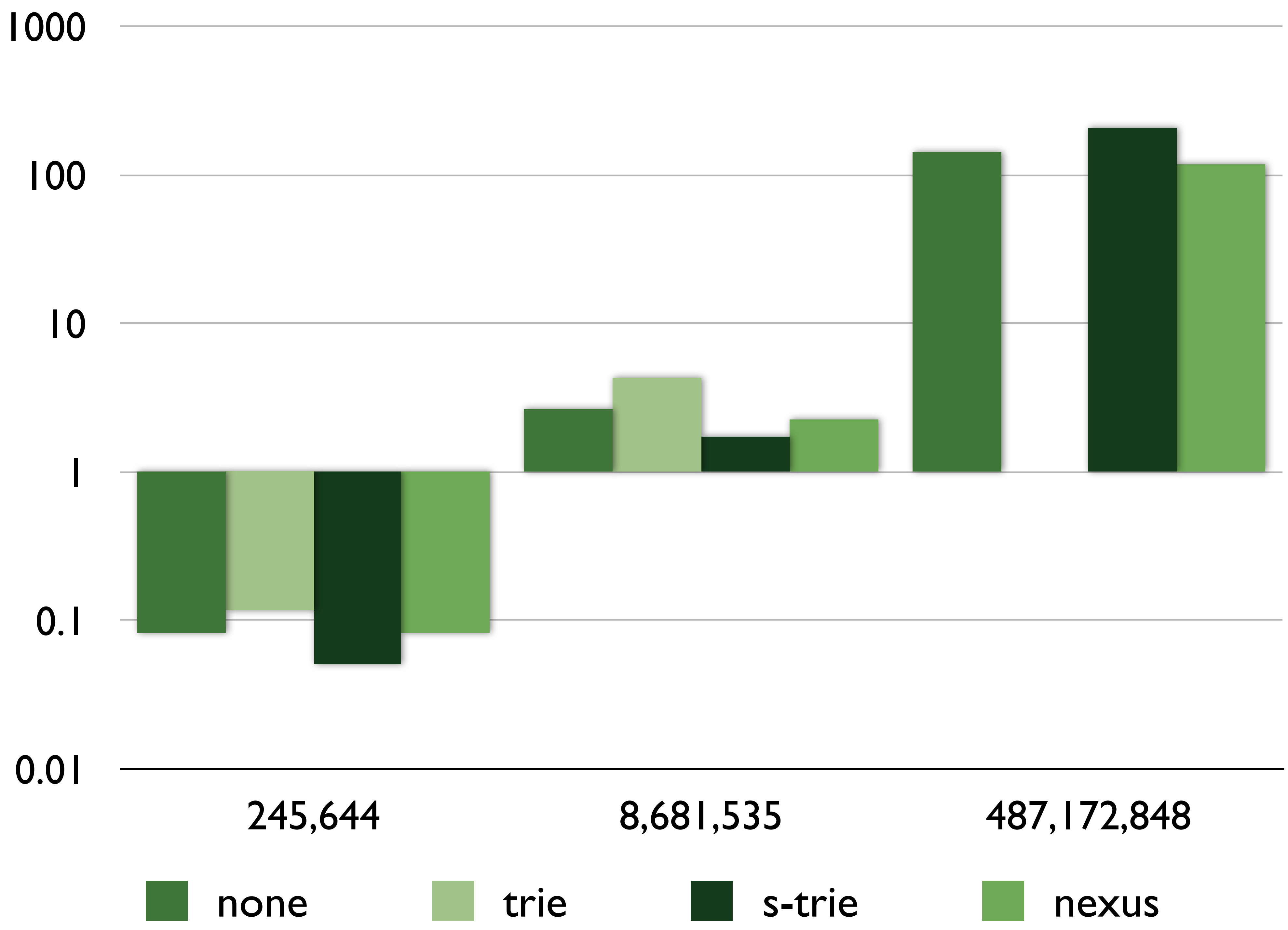
a
LNode
:
LLeaf
:
:
:
LLeaf
S# 1
:
S# 2
[]


a
:
:
:
Thunk
AP
PAP
Thunk
PAP
Thunk
AP
(,)
Thunk
Fun
PAP
(,)
(,)
AP
PAP
(,)
:
AP
BCO
PAP
PAP
:
Num
AP
AP
Num
:
Sub 22
AP
BCO
BCO
BCO
BCO
Thunk
BCO
I# 1
BCO
I# 2
:
BCO
Mul 19
BCO
Add 0
BCO
[]
BCO
Div 15
BCO
BCO
BCO



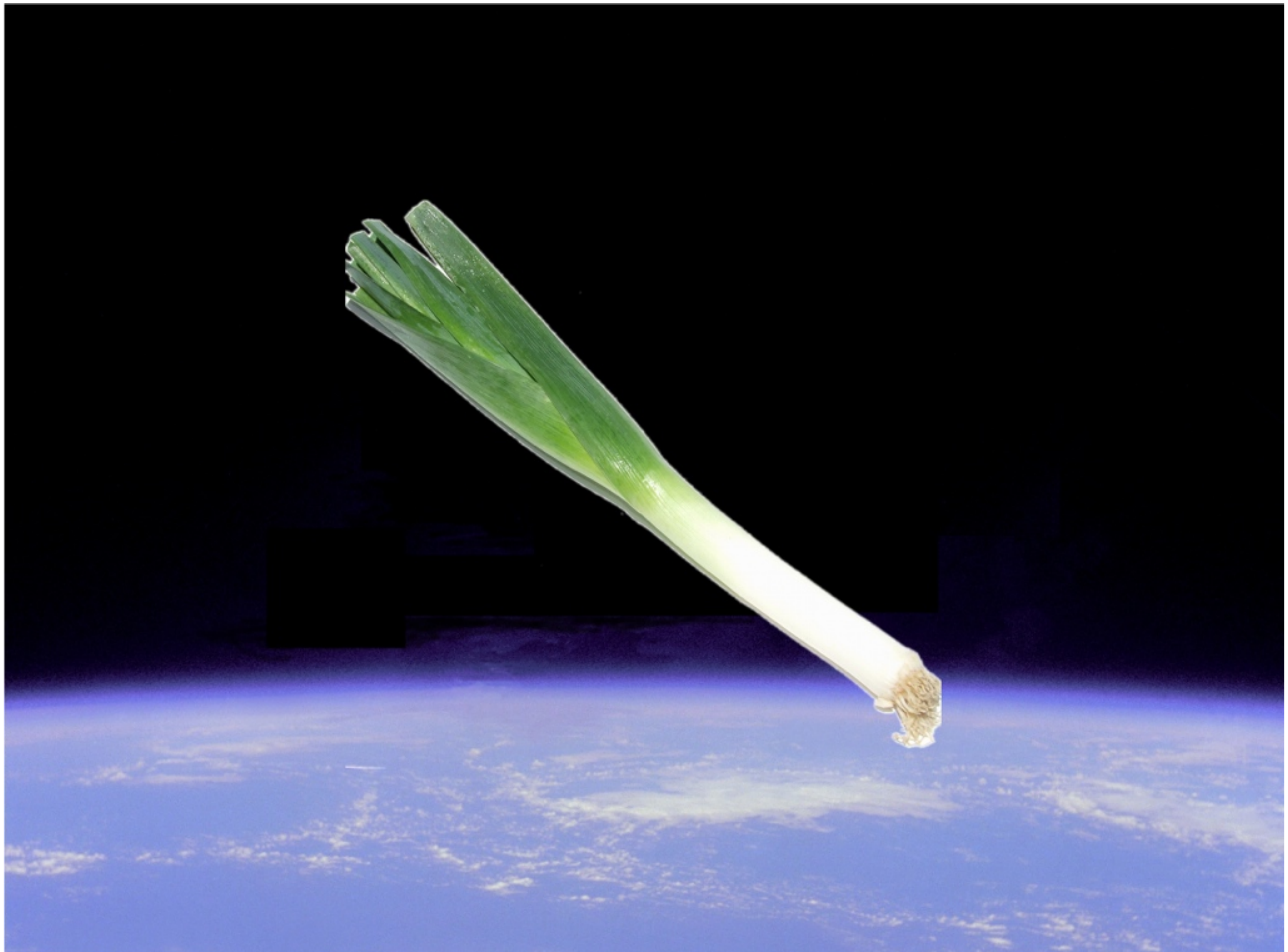
a
:
:
:
Thunk
AP
Thunk
PAP
Thunk
AP
(,)
PAP
Fun
Thunk
:
(,)
(,)
PAP
AP
App
(,)
PAP
BCO
PAP
:
Sub 22
Num
BCO
AP
AP
AP
Num
:
PAP
BCO
BCO
Mul 19
:
BCO
I# 1
BCO
BCO
I# 2
Add 0
BCO
Thunk
Div 15
[]
BCO
BCO
BCO
BCO

Results



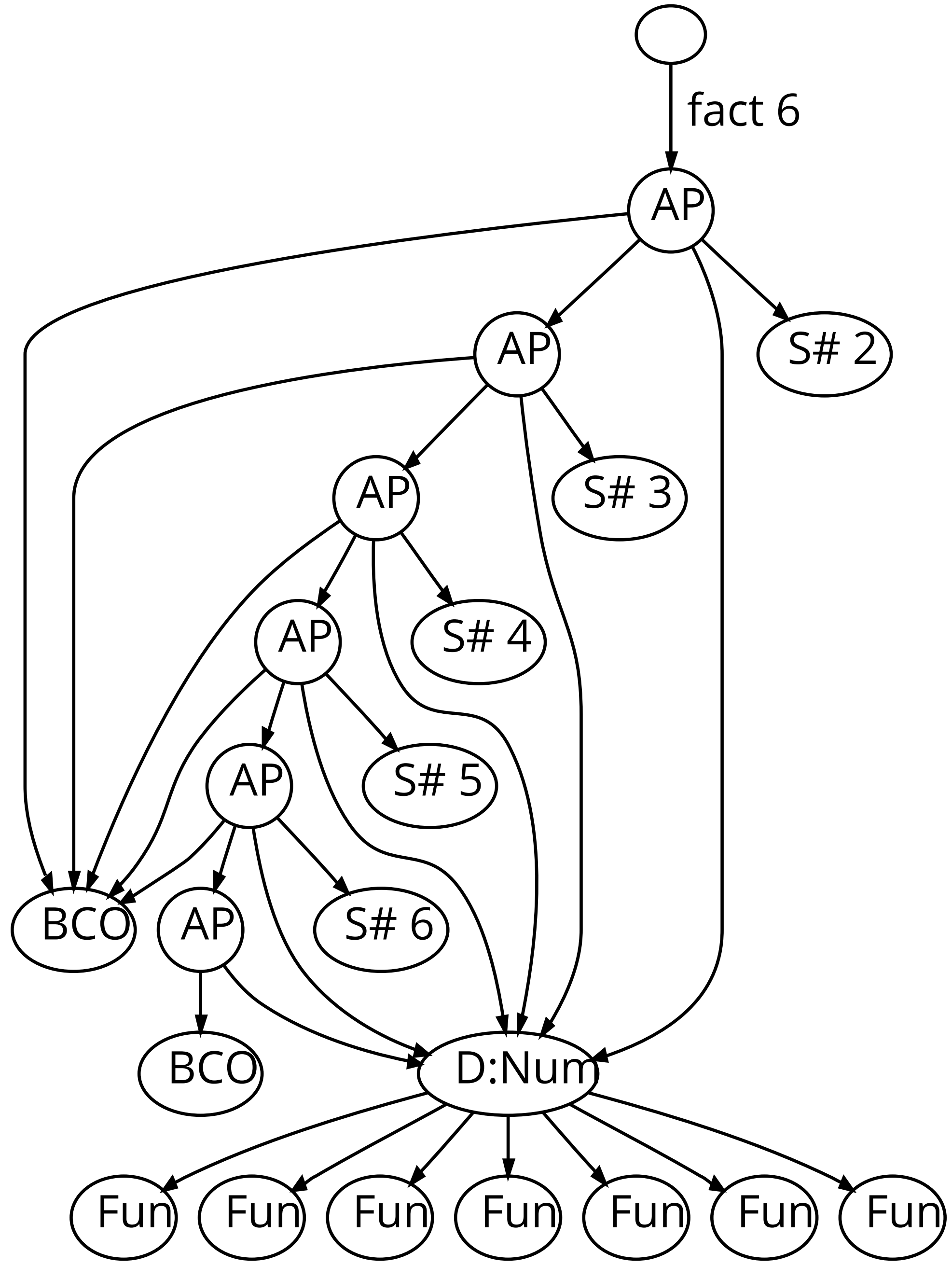
1000
100
10
1
0.1
0.01
245,644
8,681,535
487,172,848
none
trie
s-trie
nexus

Space Leak



Example

```
fact n = fact' 1 n
  where fact' a 0 = a
        fact' a n = fact' (a * n) (n - 1)

fac n = foldr (*) 1 [1..n]
```


fact 6
AP
AP
S# 2
AP
S# 3
AP
S# 4
AP
S# 5
BCO
AP
S# 6
BCO
D:Num
Fun
Fun
Fun
Fun
Fun
Fun
Fun

fac 6: TSO

Problems

How many values to keep?

When to throw them away?

How general is the “newmemo” function?

Is it always faster?

Is it working?

Applied to infinite or circular structures?

Libraries

Memo Trie

https://github.com/conal/MemoTrie

Monad Memo

https://code.google.com/p/monad-memo/

Memo Combinators

https://github.com/luqui/data-memocombinators

Visualising Haskell

ghc-vis

http://felsin9.de/nnis/ghc-vis/

Hood and GHood

http://www.ittc.ku.edu/csdl/fpg/software/hood.html

Vacuum

http://thoughtpolice.github.io/vacuum/

Helium (Haskell Subset)

http://www.cs.uu.nl/wiki/Helium

Links

Discussion on Donald Michie's Paper

https://groups.google.com/group/comp.lang.lisp/tree/browse_frm/month/1998-02/4f3f2074edd8d631

POP-11 Book

http://www.cs.bham.ac.uk/research/projects/poplog/popbook/popbook.html

Interview with Donald Michie

http://www.youtube.com/watch?v=6p3mhkNgRXs

Recursive programming and memoization

http://www.youtube.com/watch?v=HVGTKI2jtLs

Thank You