Modular Searching with Higher-Order Functions

Paulo Oliva
Queen Mary University of London

British Logic Colloquium
University of Sussex
14 September 2017

A Puzzle

A Puzzle

Using the numbers 1,2,…,10 fill in the empty cells below so that each row and column has the same sum

	X	X	X
	X	X	X

A Puzzle

Using the numbers 1,2,…,10 fill in the empty cells below so that each row and column has the same sum

1	X	X	X
2	**5**	**7**	**8**
9	**3**	**4**	**6**
10	X	X	X

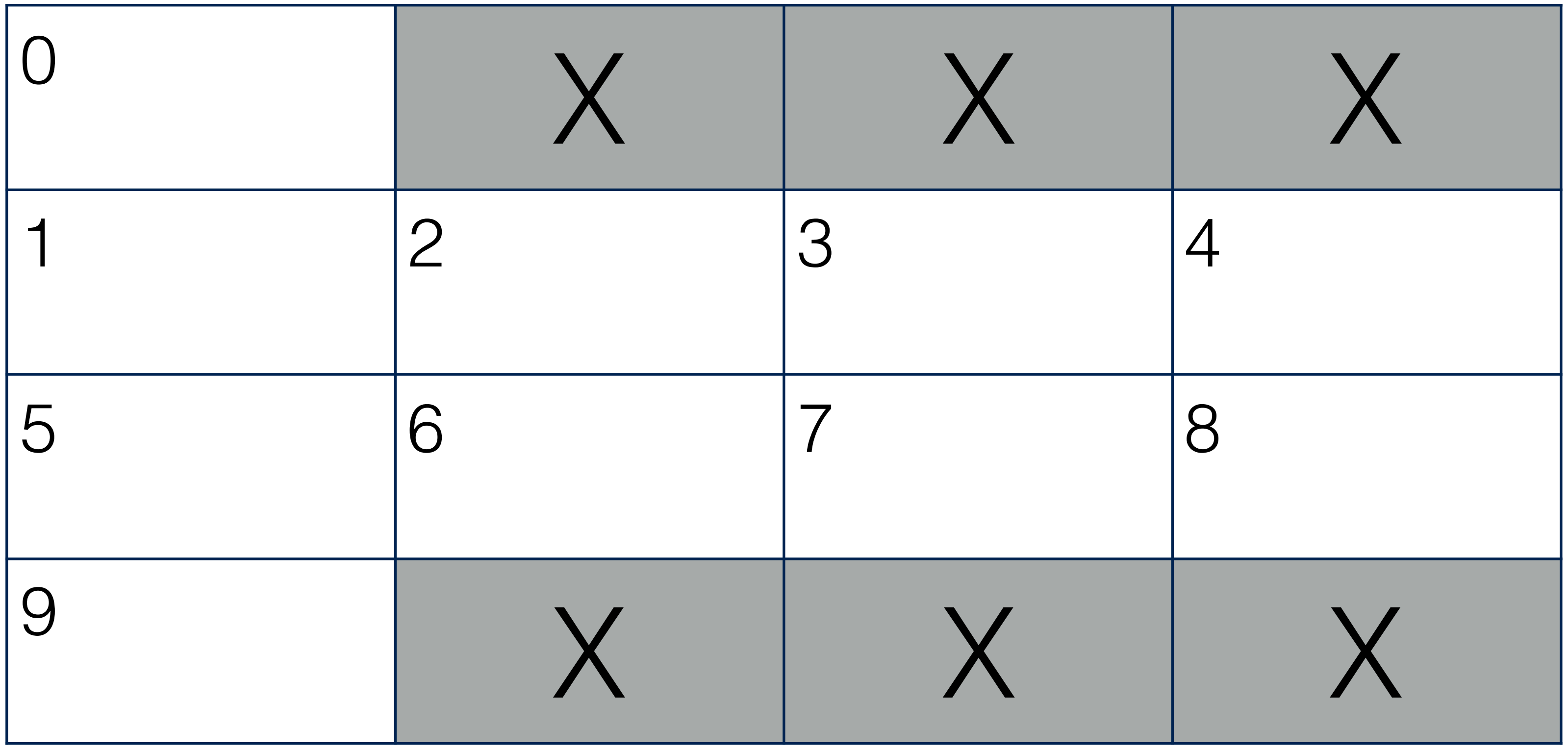
Searching for a Solution…

Searching for a Solution…

Order the cells:

Searching for a Solution…

Order the cells:



0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Searching for a Solution…

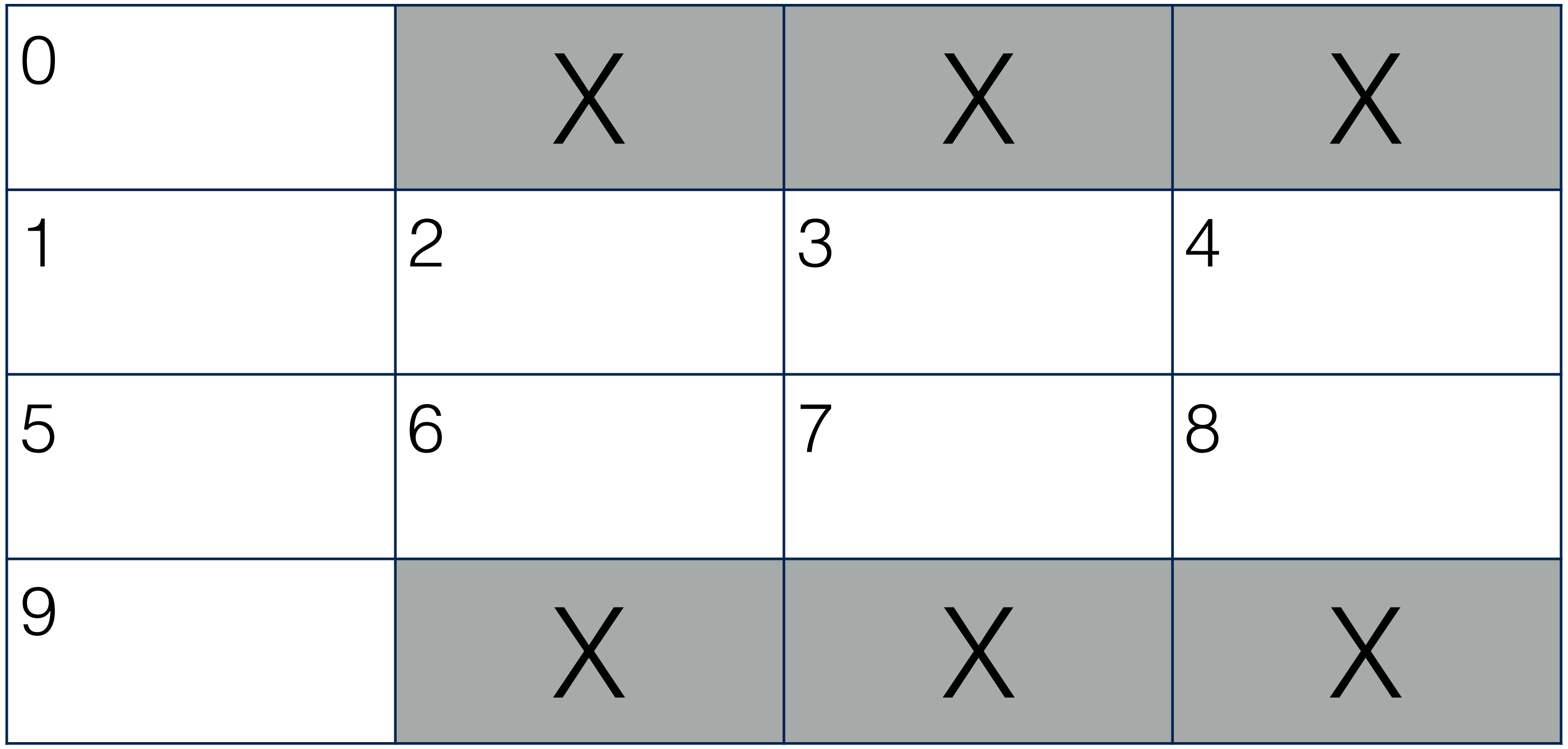
Order the cells:

0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Generate all arrays $[x_0,\dots,x_9]$, with x_i in $\{1,\dots,10\}$

Searching for a Solution…

Order the cells:

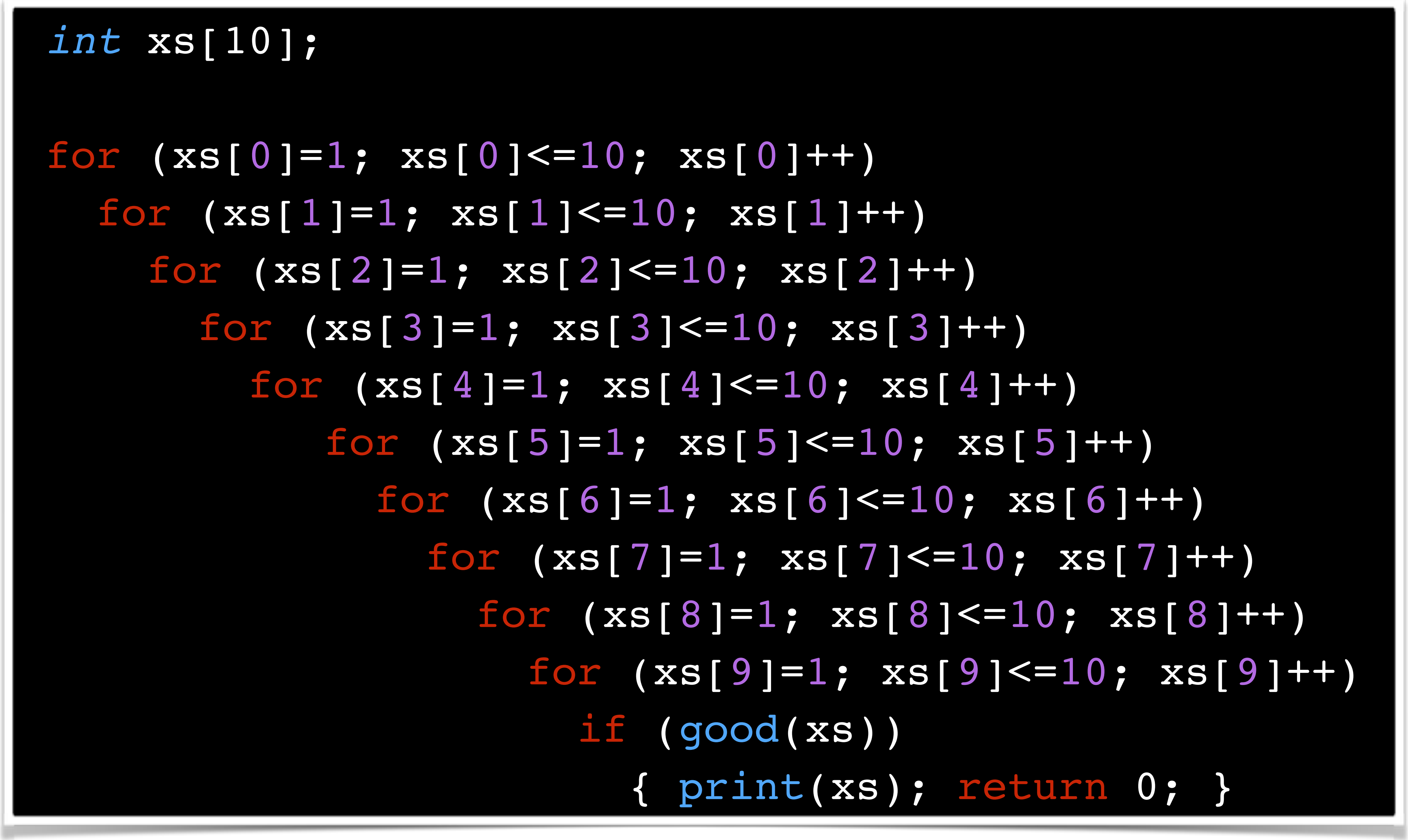


0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Generate all arrays [x_0,…,x_9], with x_i in {1,…,10}

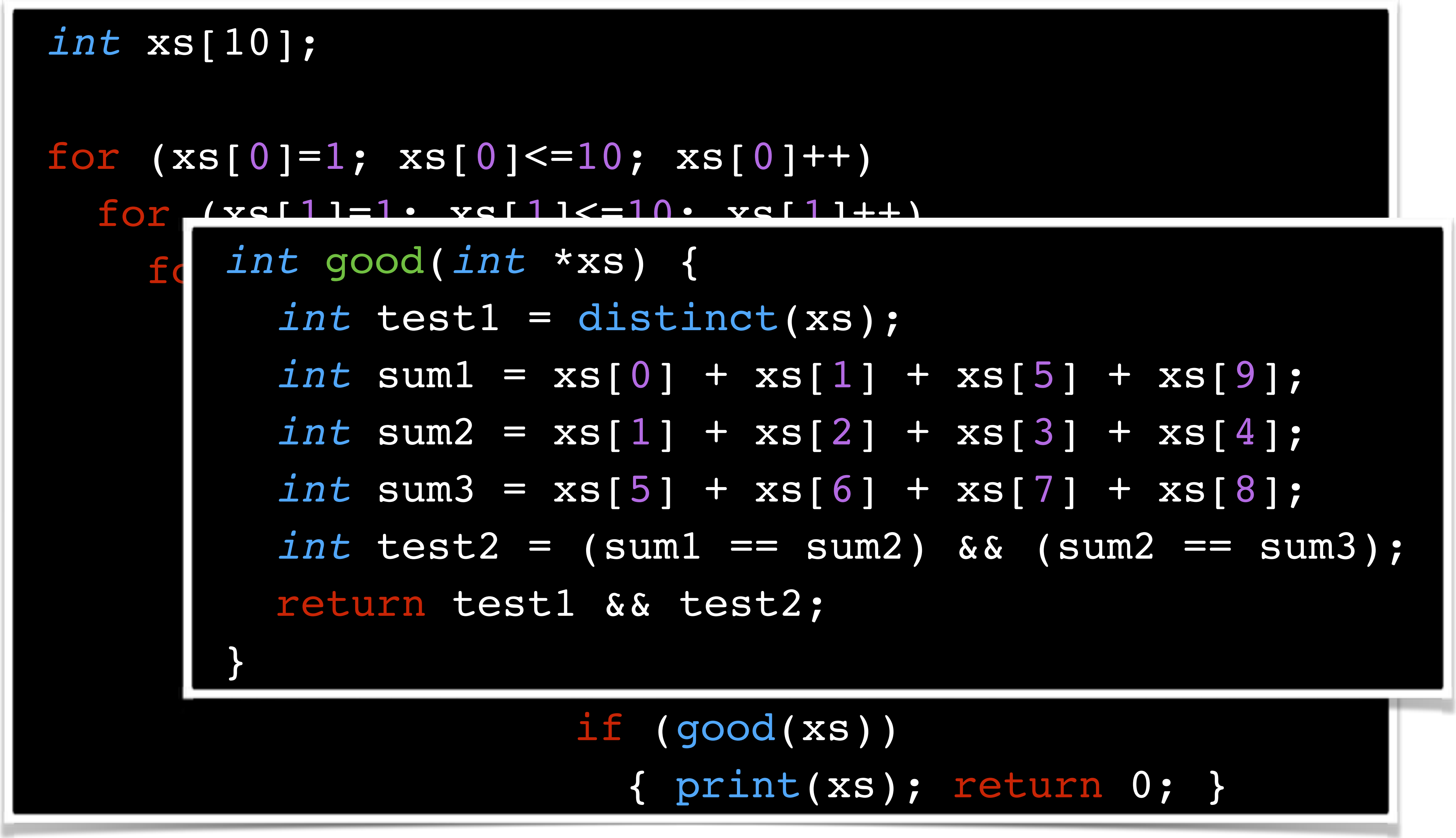
Until we find a “good” one

C Implementation



```
int xs[10];

for (xs[0]=1; xs[0]<=10; xs[0]++)
  for (xs[1]=1; xs[1]<=10; xs[1]++)
    for (xs[2]=1; xs[2]<=10; xs[2]++)
      for (xs[3]=1; xs[3]<=10; xs[3]++)
        for (xs[4]=1; xs[4]<=10; xs[4]++)
           for (xs[5]=1; xs[5]<=10; xs[5]++)
             for (xs[6]=1; xs[6]<=10; xs[6]++)
               for (xs[7]=1; xs[7]<=10; xs[7]++)
                 for (xs[8]=1; xs[8]<=10; xs[8]++)
                   for (xs[9]=1; xs[9]<=10; xs[9]++)
                     if (good(xs))
                       { print(xs); return 0; }
```

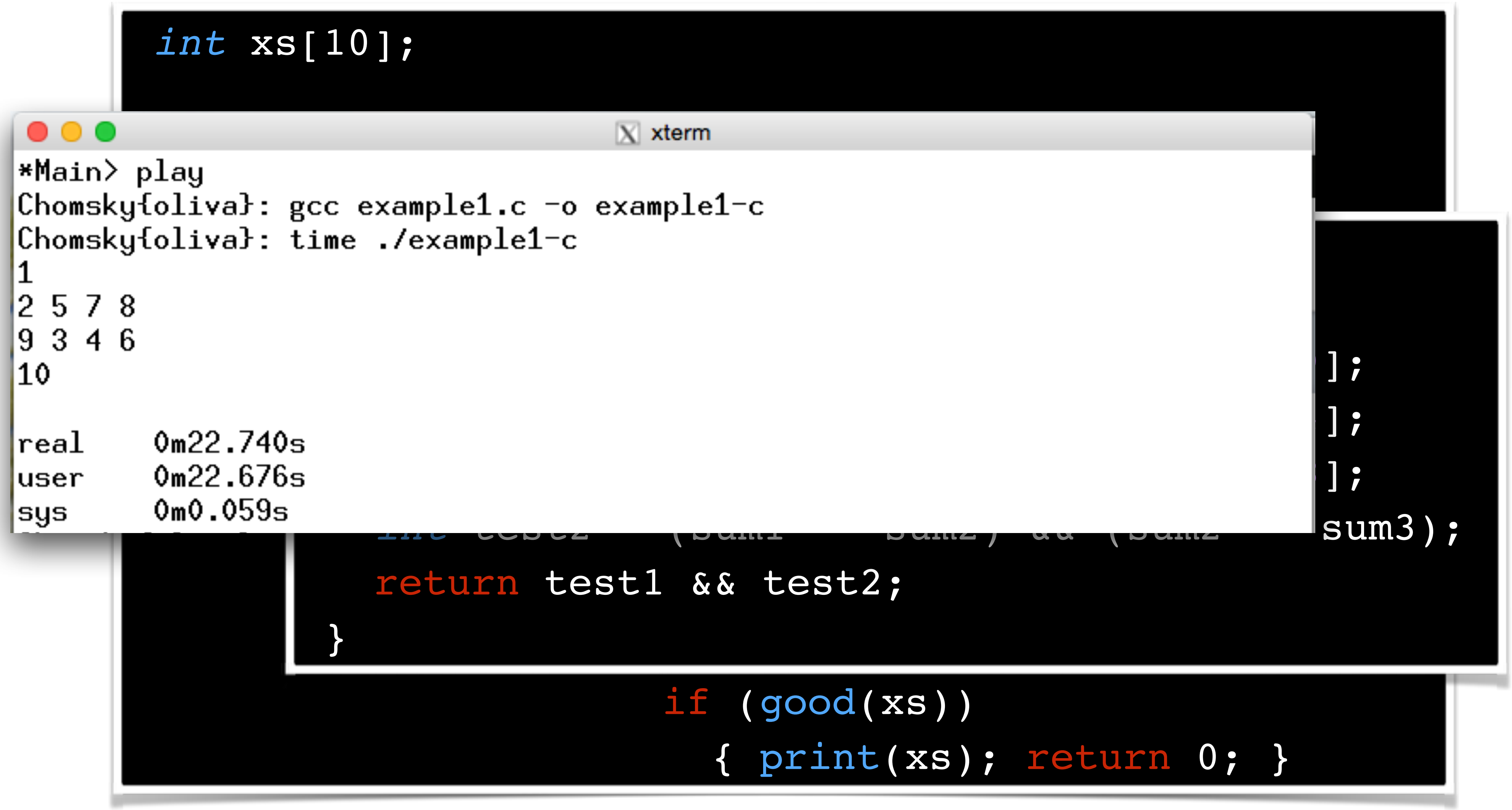
C Implementation



```
int xs[10];

for (xs[0]=1; xs[0]<=10; xs[0]++)
  for (xs[1]=1; xs[1]<=10; xs[1]++)
    fo

                    if (good(xs))
                      { print(xs); return 0; }
```

```
int good(int *xs) {
  int test1 = distinct(xs);
  int sum1 = xs[0] + xs[1] + xs[5] + xs[9];
  int sum2 = xs[1] + xs[2] + xs[3] + xs[4];
  int sum3 = xs[5] + xs[6] + xs[7] + xs[8];
  int test2 = (sum1 == sum2) && (sum2 == sum3);
  return test1 && test2;
}
```

C Implementation



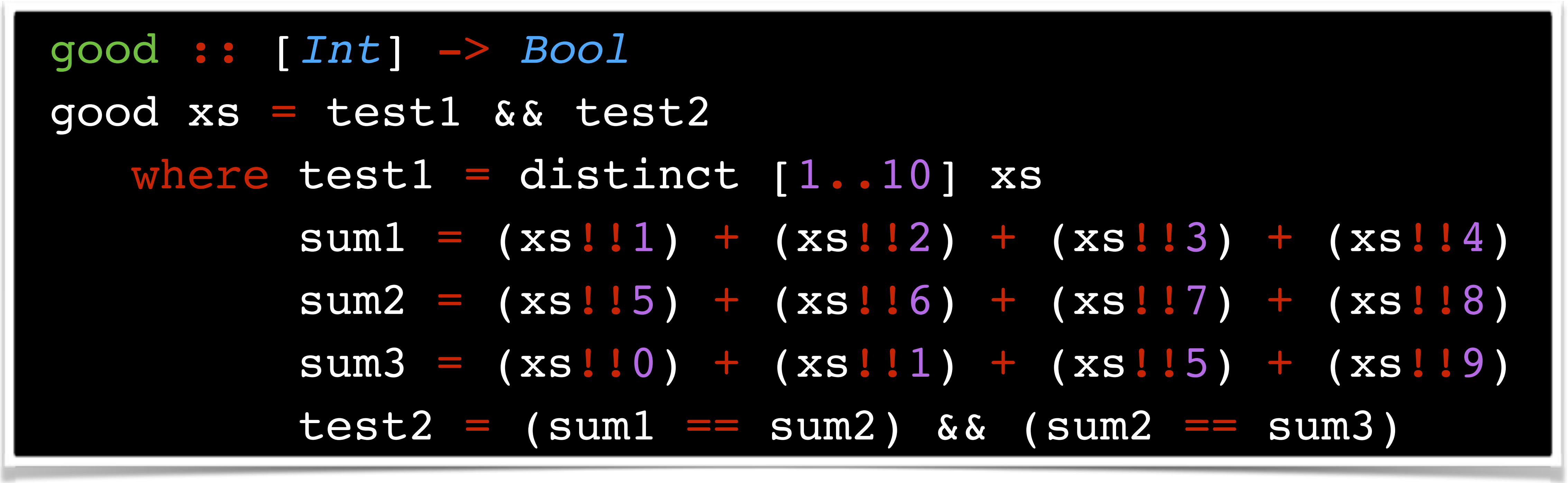
```
int xs[10];
```

```
*Main> play
Chomsky{oliva}: gcc example1.c -o example1-c
Chomsky{oliva}: time ./example1-c
1
2 5 7 8
9 3 4 6
10

real	0m22.740s
user	0m22.676s
sys	0m0.059s
```

```
];
];
];
sum3);
  return test1 && test2;
}
```

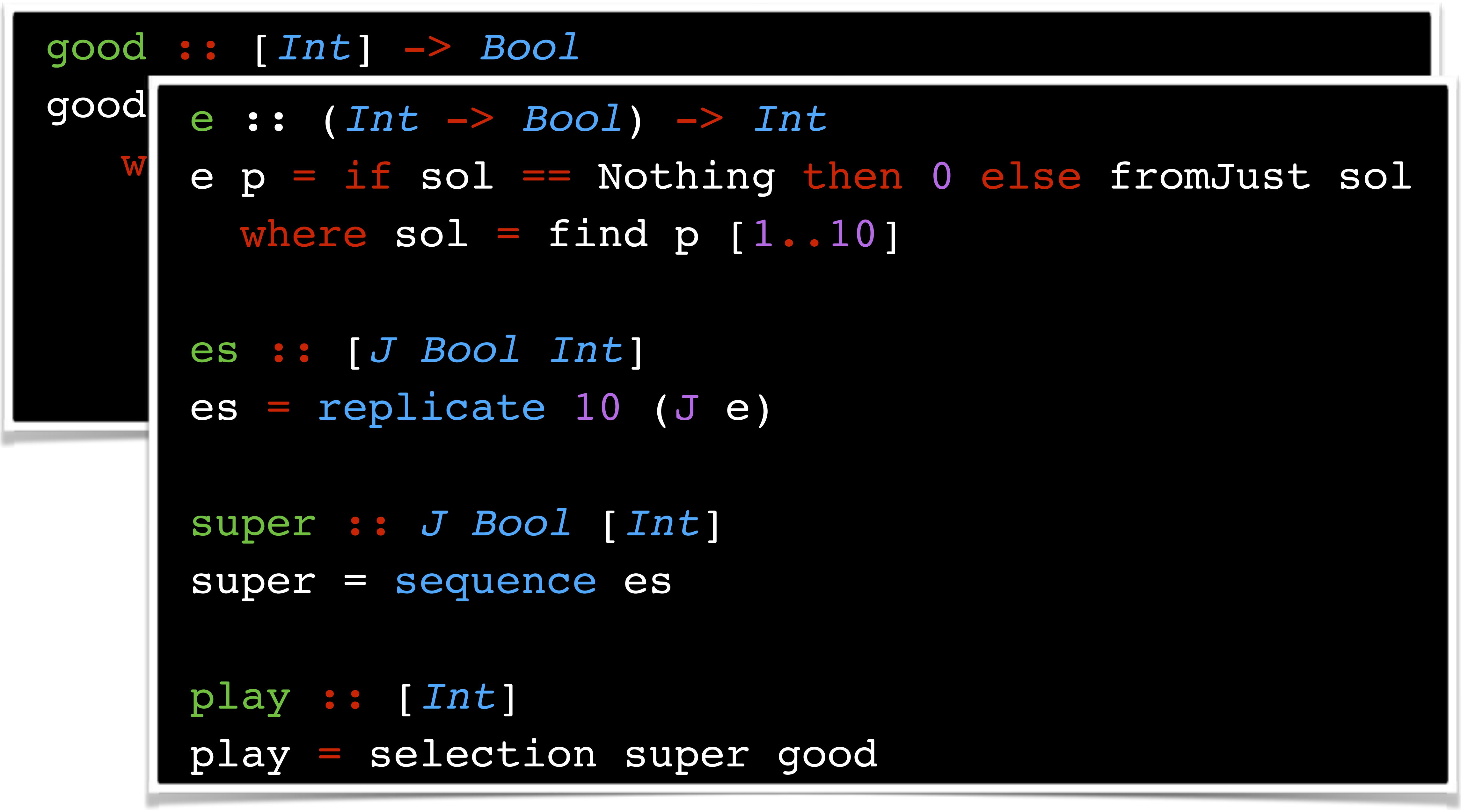
```
if (good(xs))
  { print(xs); return 0; }
```

Haskell Implementation



```haskell
good :: [Int] -> Bool
good xs = test1 && test2
   where test1 = distinct [1..10] xs
         sum1 = (xs!!1) + (xs!!2) + (xs!!3) + (xs!!4)
         sum2 = (xs!!5) + (xs!!6) + (xs!!7) + (xs!!8)
         sum3 = (xs!!0) + (xs!!1) + (xs!!5) + (xs!!9)
         test2 = (sum1 == sum2) && (sum2 == sum3)
```

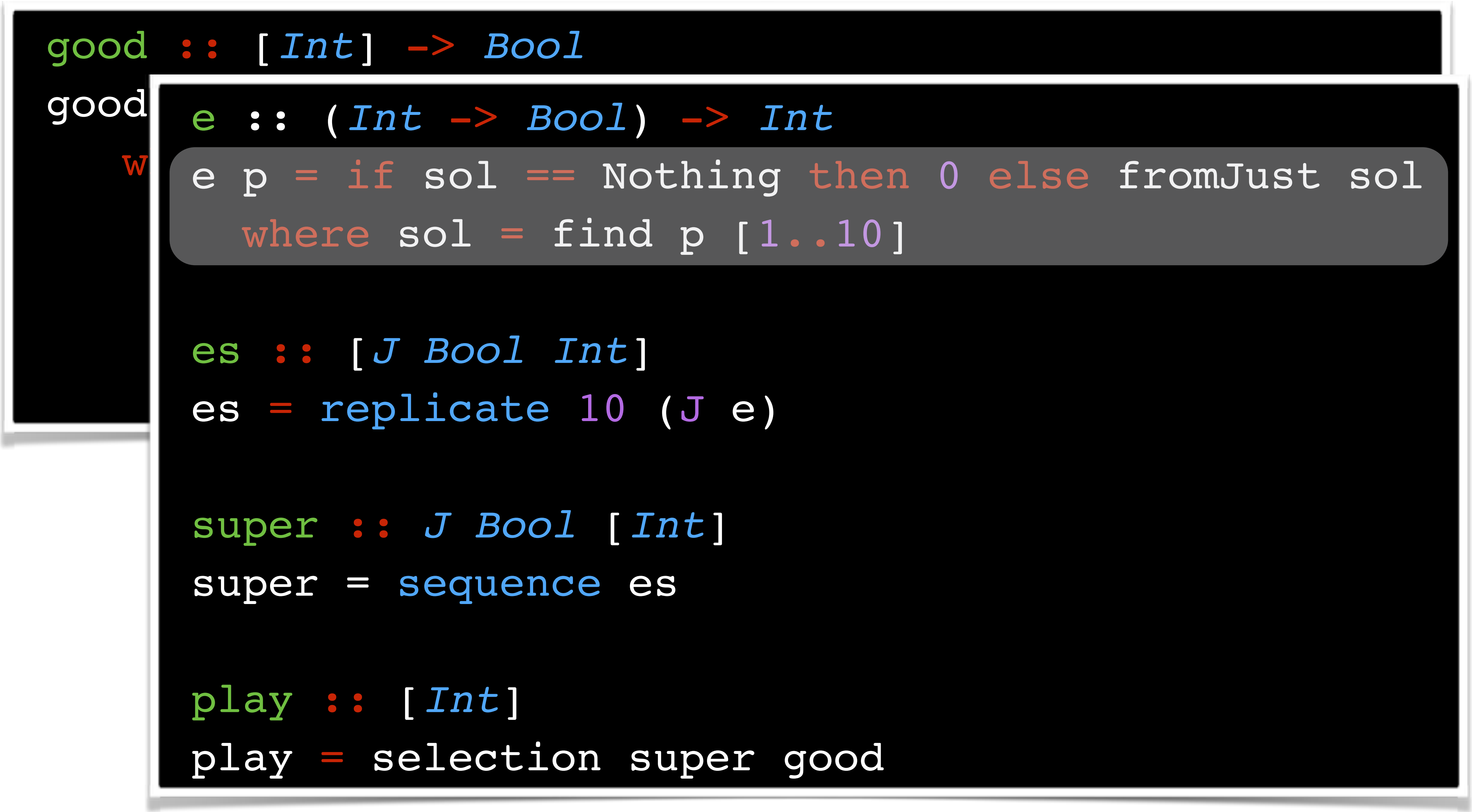
Haskell Implementation



```
good :: [Int] -> Bool
good
    w
```

```
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 0 else fromJust sol
  where sol = find p [1..10]

es :: [J Bool Int]
es = replicate 10 (J e)

super :: J Bool [Int]
super = sequence es

play :: [Int]
play = selection super good
```

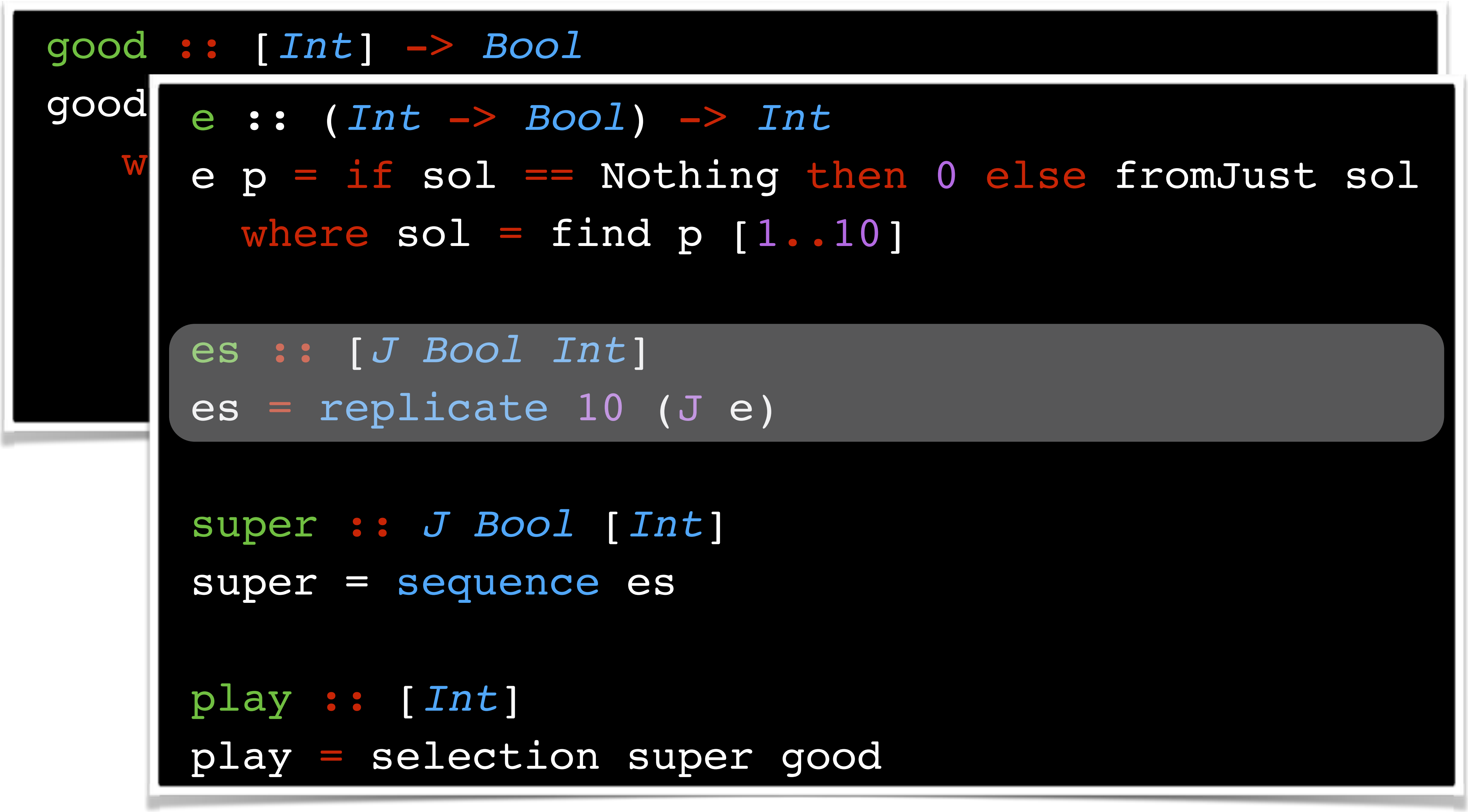
Haskell Implementation



```haskell
good :: [Int] -> Bool
good
    w
```

```haskell
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 0 else fromJust sol
  where sol = find p [1..10]

es :: [J Bool Int]
es = replicate 10 (J e)

super :: J Bool [Int]
super = sequence es

play :: [Int]
play = selection super good
```

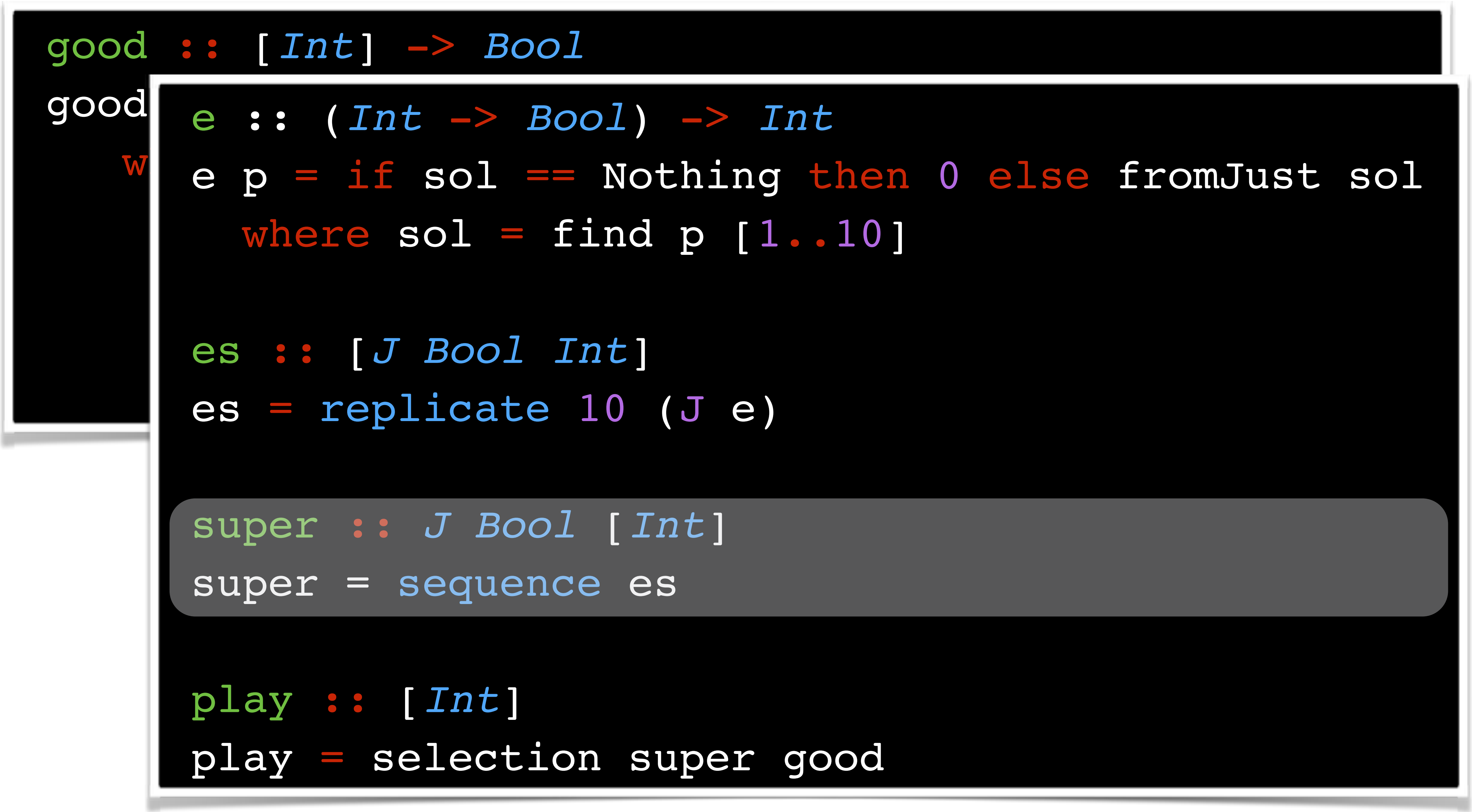
Haskell Implementation



```
good :: [Int] -> Bool
good
    w
```

```
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 0 else fromJust sol
  where sol = find p [1..10]

es :: [J Bool Int]
es = replicate 10 (J e)

super :: J Bool [Int]
super = sequence es

play :: [Int]
play = selection super good
```

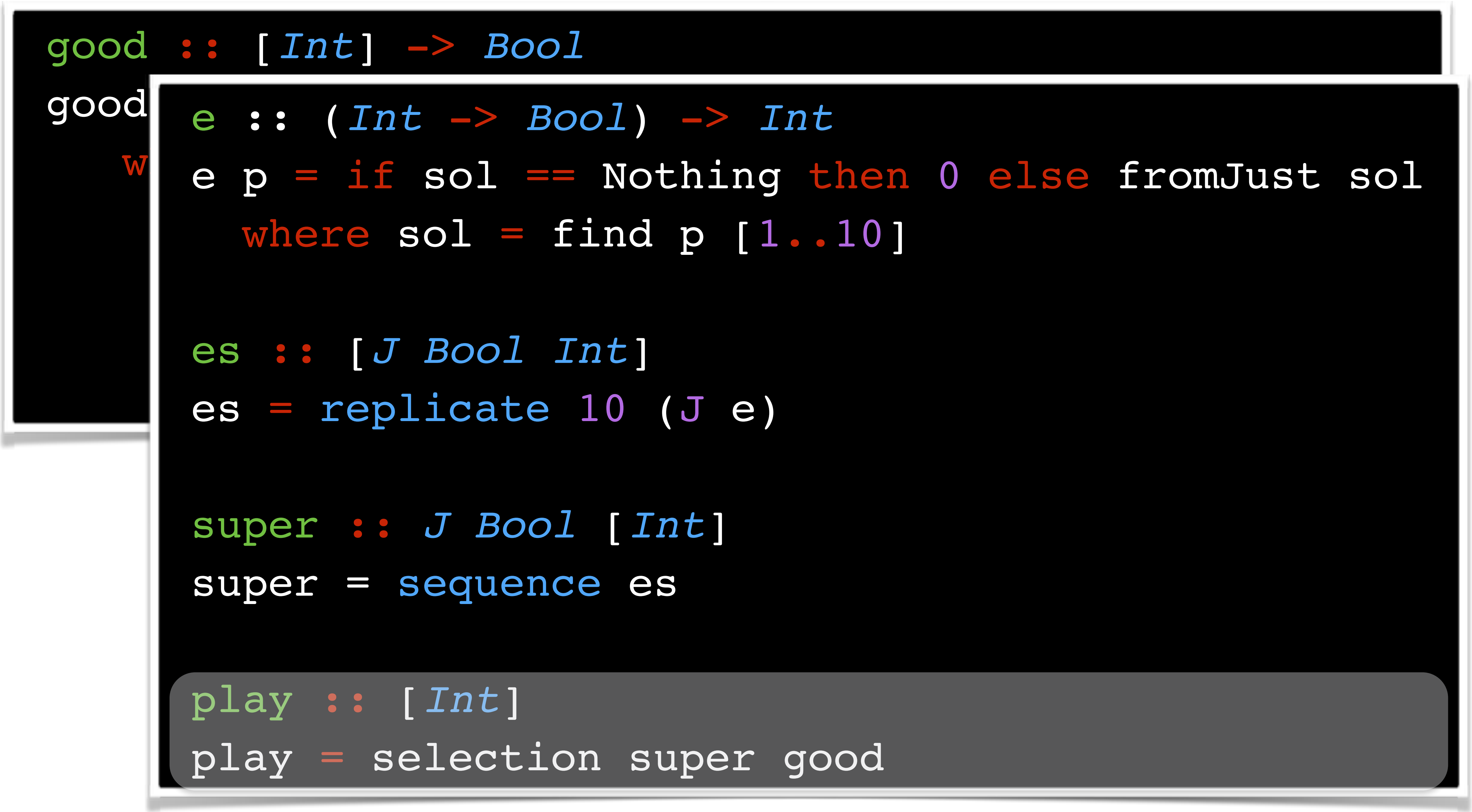
Haskell Implementation



```
good :: [Int] -> Bool
good
    w
```

```
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 0 else fromJust sol
  where sol = find p [1..10]

es :: [J Bool Int]
es = replicate 10 (J e)

super :: J Bool [Int]
super = sequence es

play :: [Int]
play = selection super good
```

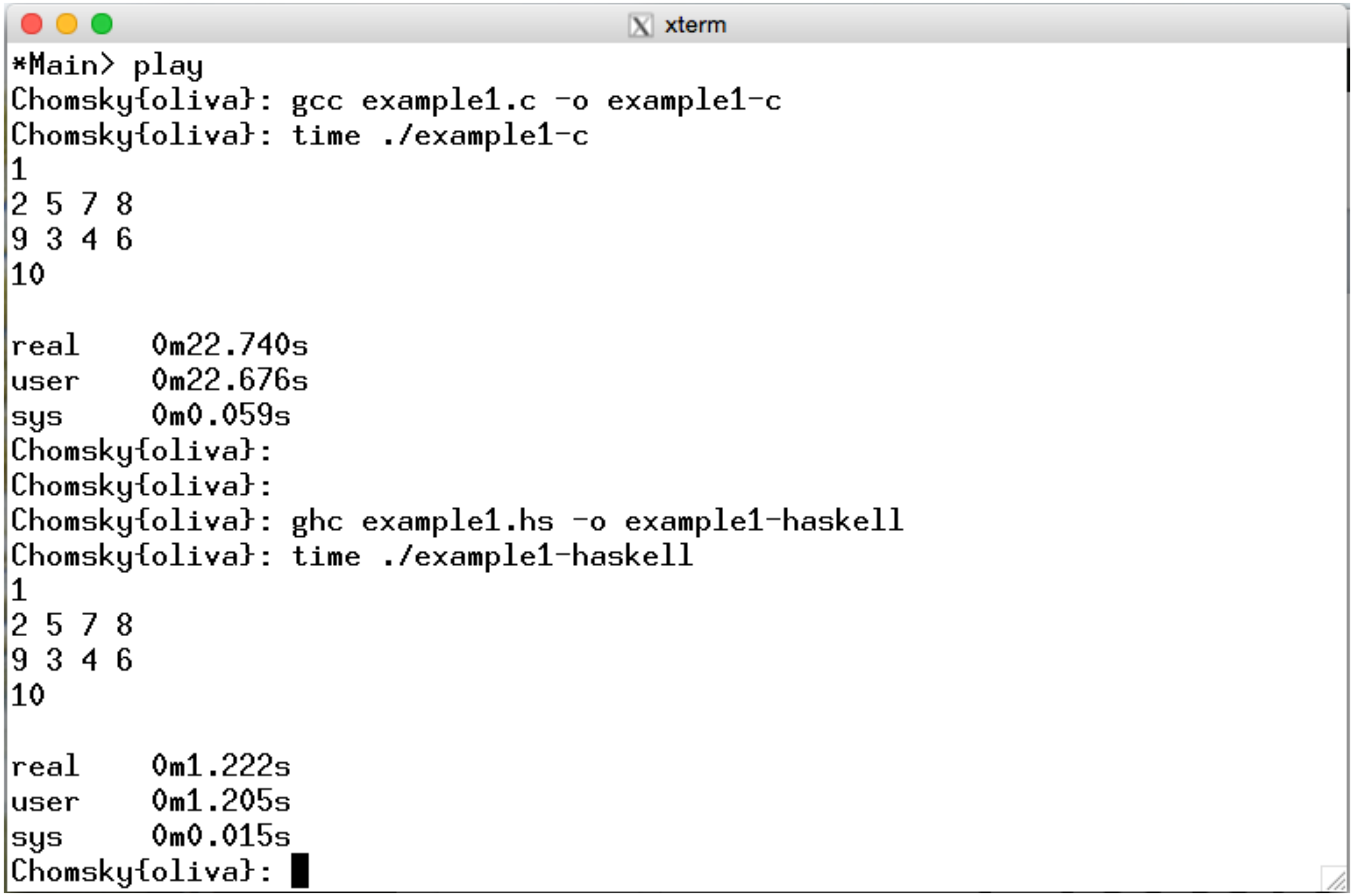
Haskell Implementation



```
good :: [Int] -> Bool
good
    w
```

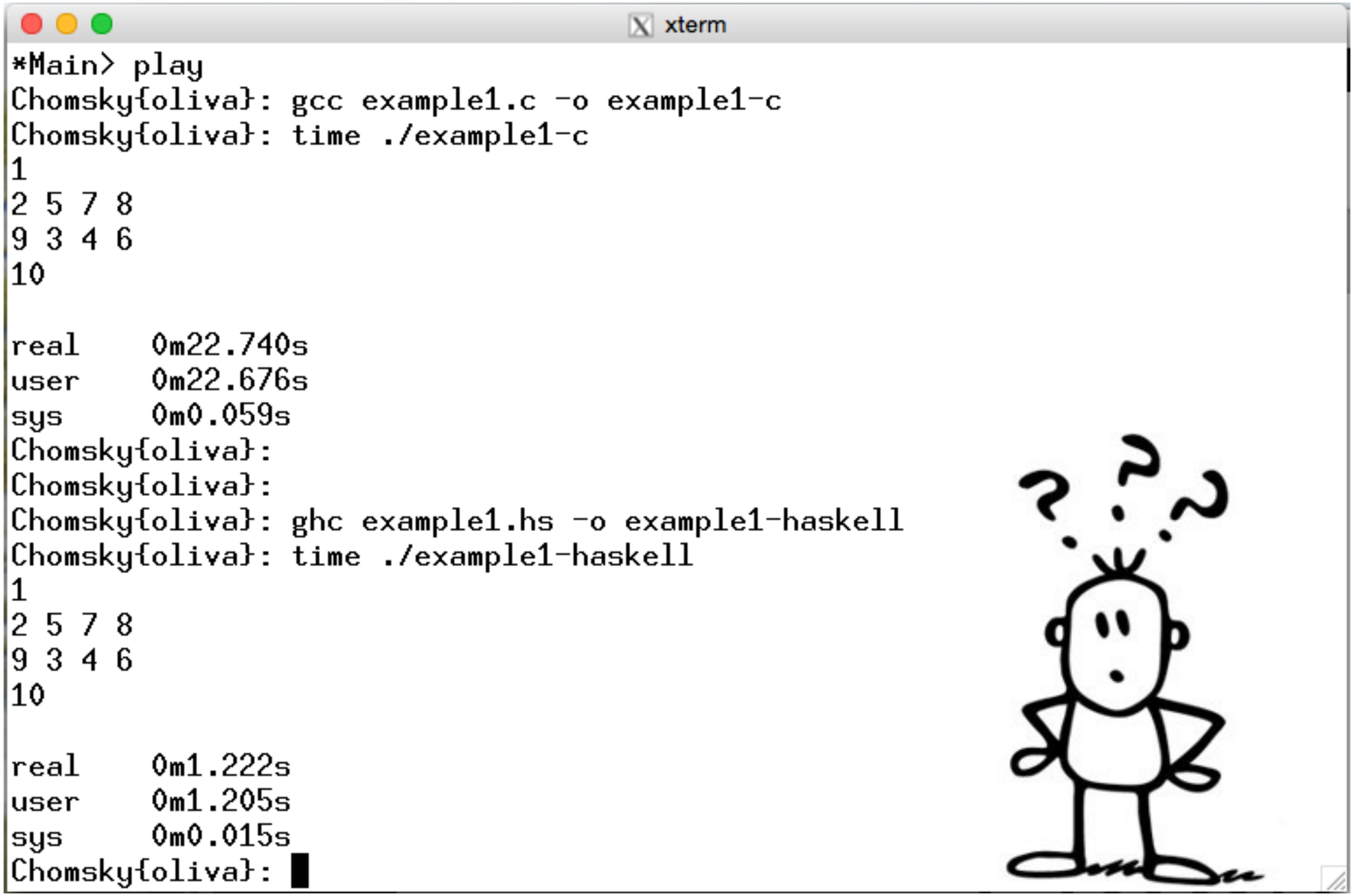
```
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 0 else fromJust sol
  where sol = find p [1..10]

es :: [J Bool Int]
es = replicate 10 (J e)

super :: J Bool [Int]
super = sequence es

play :: [Int]
play = selection super good
```

Haskell 20x faster than C



```
*Main> play
Chomsky{oliva}: gcc example1.c -o example1-c
Chomsky{oliva}: time ./example1-c
1
2 5 7 8
9 3 4 6
10

real    0m22.740s
user    0m22.676s
sys     0m0.059s
Chomsky{oliva}:
Chomsky{oliva}:
Chomsky{oliva}: ghc example1.hs -o example1-haskell
Chomsky{oliva}: time ./example1-haskell
1
2 5 7 8
9 3 4 6
10

real    0m1.222s
user    0m1.205s
sys     0m0.015s
Chomsky{oliva}:
```

Haskell 20x faster than C



```
*Main> play
Chomsky{oliva}: gcc example1.c -o example1-c
Chomsky{oliva}: time ./example1-c
1
2 5 7 8
9 3 4 6
10

real    0m22.740s
user    0m22.676s
sys     0m0.059s
Chomsky{oliva}:
Chomsky{oliva}:
Chomsky{oliva}: ghc example1.hs -o example1-haskell
Chomsky{oliva}: time ./example1-haskell
1
2 5 7 8
9 3 4 6
10

real    0m1.222s
user    0m1.205s
sys     0m0.015s
Chomsky{oliva}: █
```

Selection and Continuation Monads

Selection Monad

- Fix R. The type mapping

$$J_R X = (X \to R) \to X$$

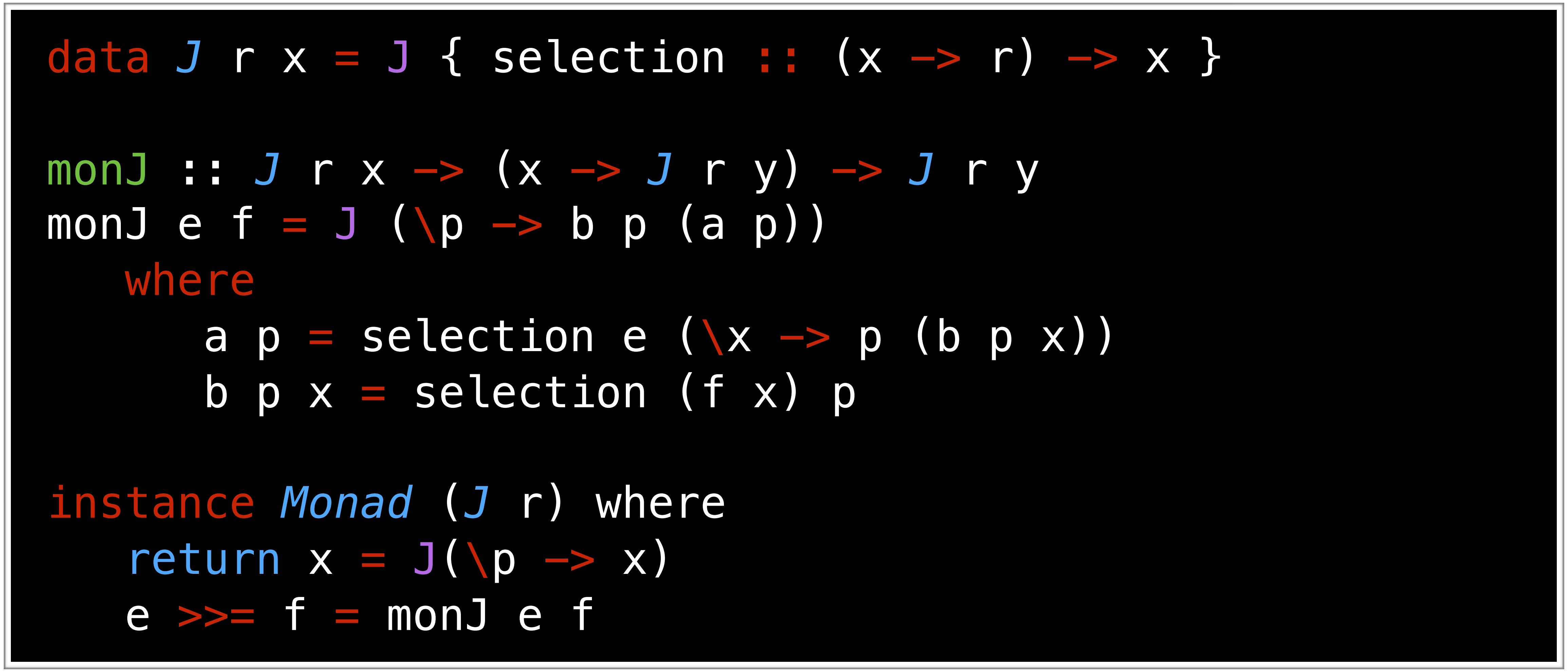
  is a **strong monad**

Selection Monad

- Fix R. The type mapping

$$J_R X = (X \rightarrow R) \rightarrow X$$

is a **strong monad**



```haskell
data J r x = J { selection :: (x -> r) -> x }

monJ :: J r x -> (x -> J r y) -> J r y
monJ e f = J (\p -> b p (a p))
   where
      a p = selection e (\x -> p (b p x))
      b p x = selection (f x) p

instance Monad (J r) where
   return x = J(\p -> x)
   e >>= f = monJ e f
```

Interpretation

$$J_R X = (X \to R) \to X$$

Interpretation

$$\boldsymbol{J}_R \boldsymbol{X} = \underbrace{(\boldsymbol{X} \to \boldsymbol{R})}_{\text{local problem}} \to \boldsymbol{X}$$

Interpretation

$$J_R X = \underbrace{(X \to R)}_{\text{local problem}} \to \overbrace{X}^{\text{local solution}}$$

Continuation Monad

- Fix R. The type mapping

$$K_R X = (X \to R) \to R$$

  is also a **strong monad**

Continuation Monad

- Fix R. The type mapping

$$K_R X = (X \to R) \to R$$

is also a **strong monad**

$$\boxed{J_R X = (X \to R) \to X}$$

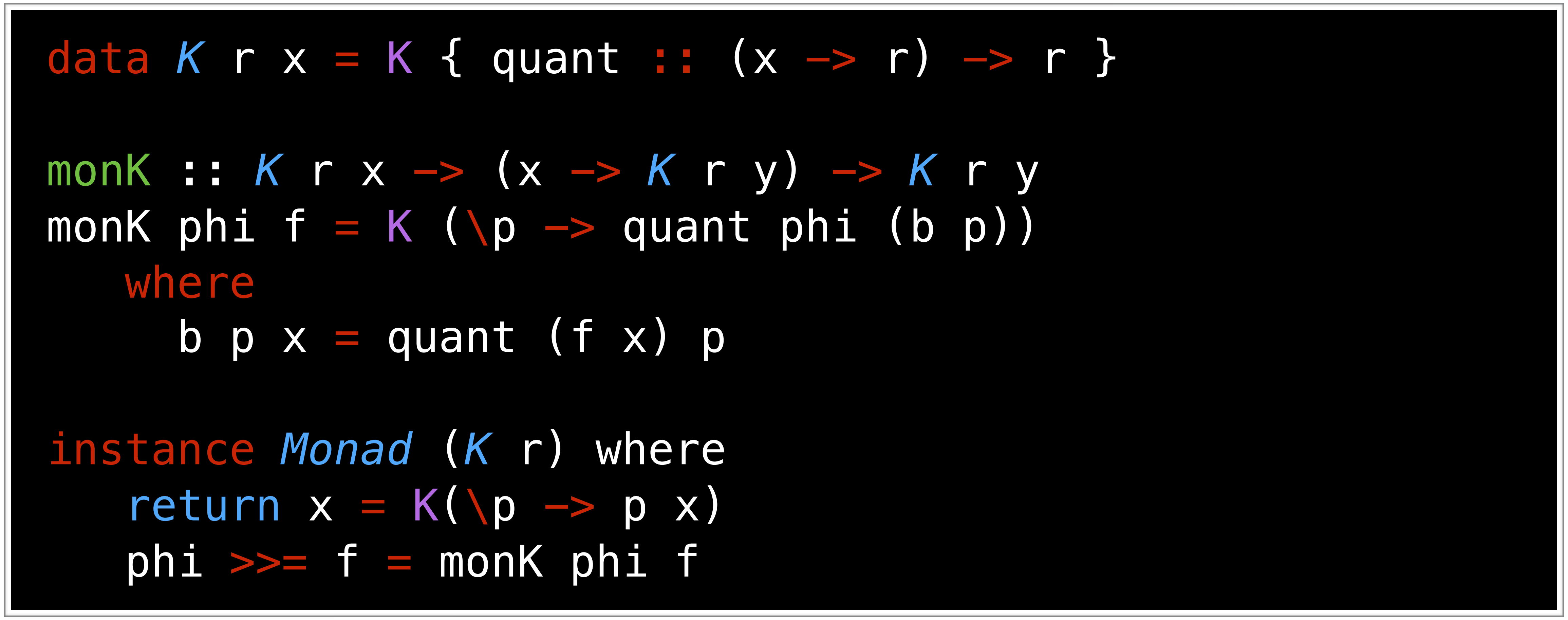
Continuation Monad

- Fix R. The type mapping

$$K_R X = (X \to R) \to R$$

is also a **strong monad**

$$\boxed{J_R X = (X \to R) \to X}$$



```
data K r x = K { quant :: (x -> r) -> r }

monK :: K r x -> (x -> K r y) -> K r y
monK phi f = K (\p -> quant phi (b p))
   where
     b p x = quant (f x) p

instance Monad (K r) where
   return x = K(\p -> p x)
   phi >>= f = monK phi f
```

Combining Local Searches

$$f \in \{K_R, J_R\}$$

Combining Local Searches

$$f \in \{K_R, J_R\}$$

- **Applicative functors** support the operation

Combining Local Searches

$$f \in \{K_R, J_R\}$$

- **Applicative functors** support the operation

$$\text{sequence} \; :: \; \Pi_i(f \; x_i) \to f(\Pi_i x_i)$$

Combining Local Searches

$$f \in \{K_R, J_R\}$$

- **Applicative functors** support the operation

$$\text{sequence} \ :: \ \Pi_i (f \ x_i) \to f(\Pi_i x_i)$$

- **Monads** support

Combining Local Searches

$$f \in \{K_R, J_R\}$$

- **Applicative functors** support the operation

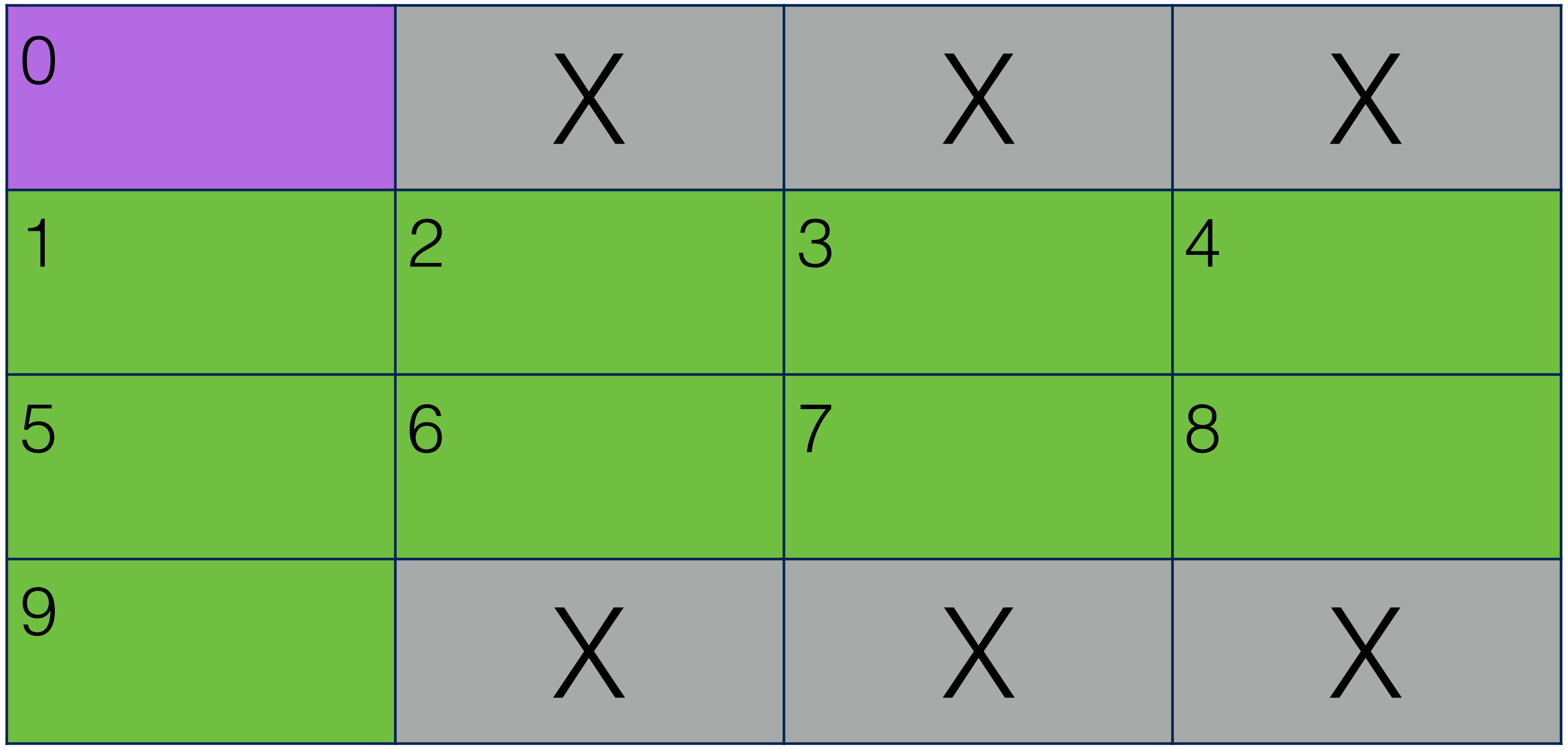
$$\text{sequence} \; :: \; \Pi_i (f \; x_i) \to f (\Pi_i x_i)$$

- **Monads** support

$$\text{depSequence} \; :: \; \Pi_i (\Pi_{j<i} x_j \to f \; x_i) \to f (\Pi_i x_i)$$

From Puzzle to Game…

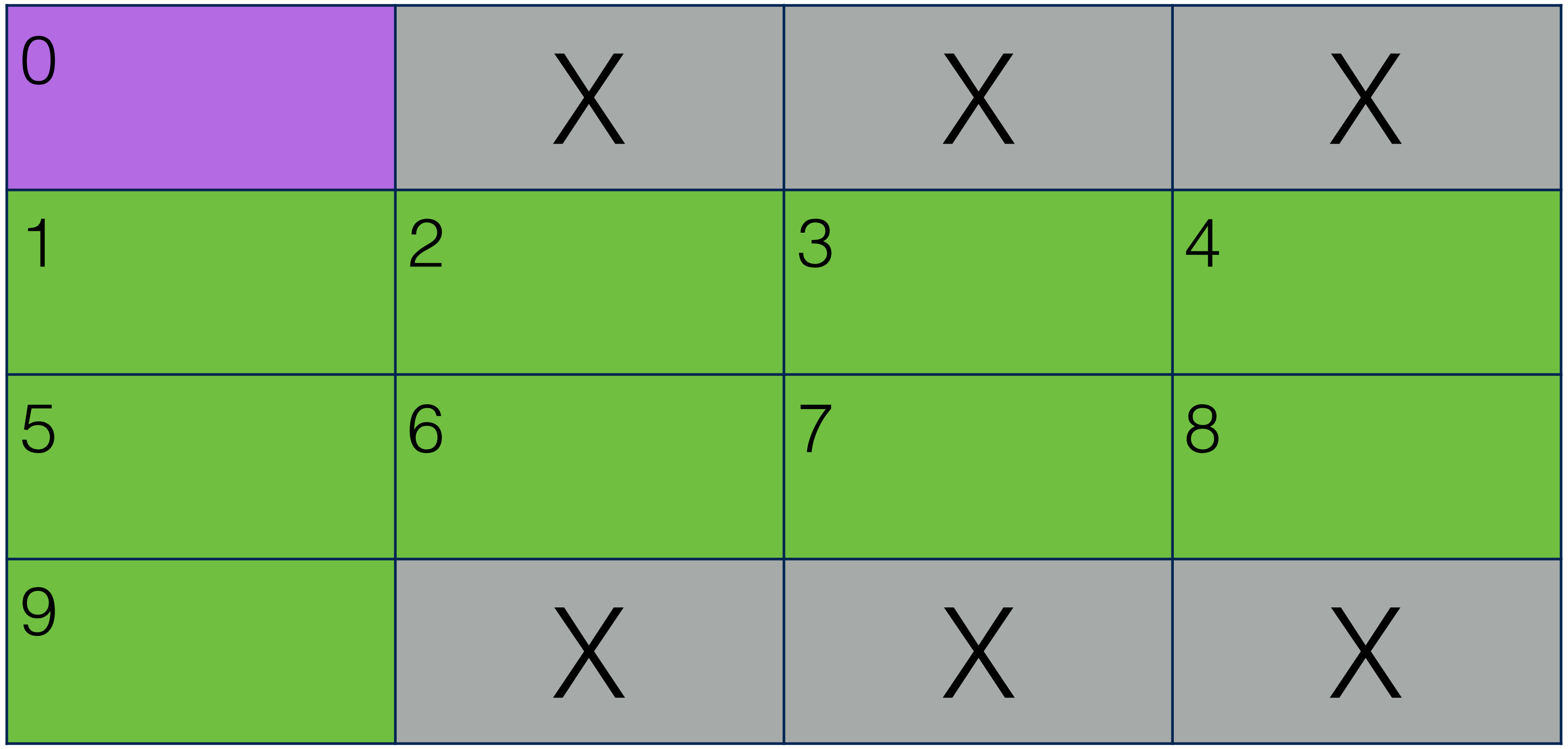
Purple player starts, Green players continues



0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

From Puzzle to Game…

Purple player starts, Green players continues

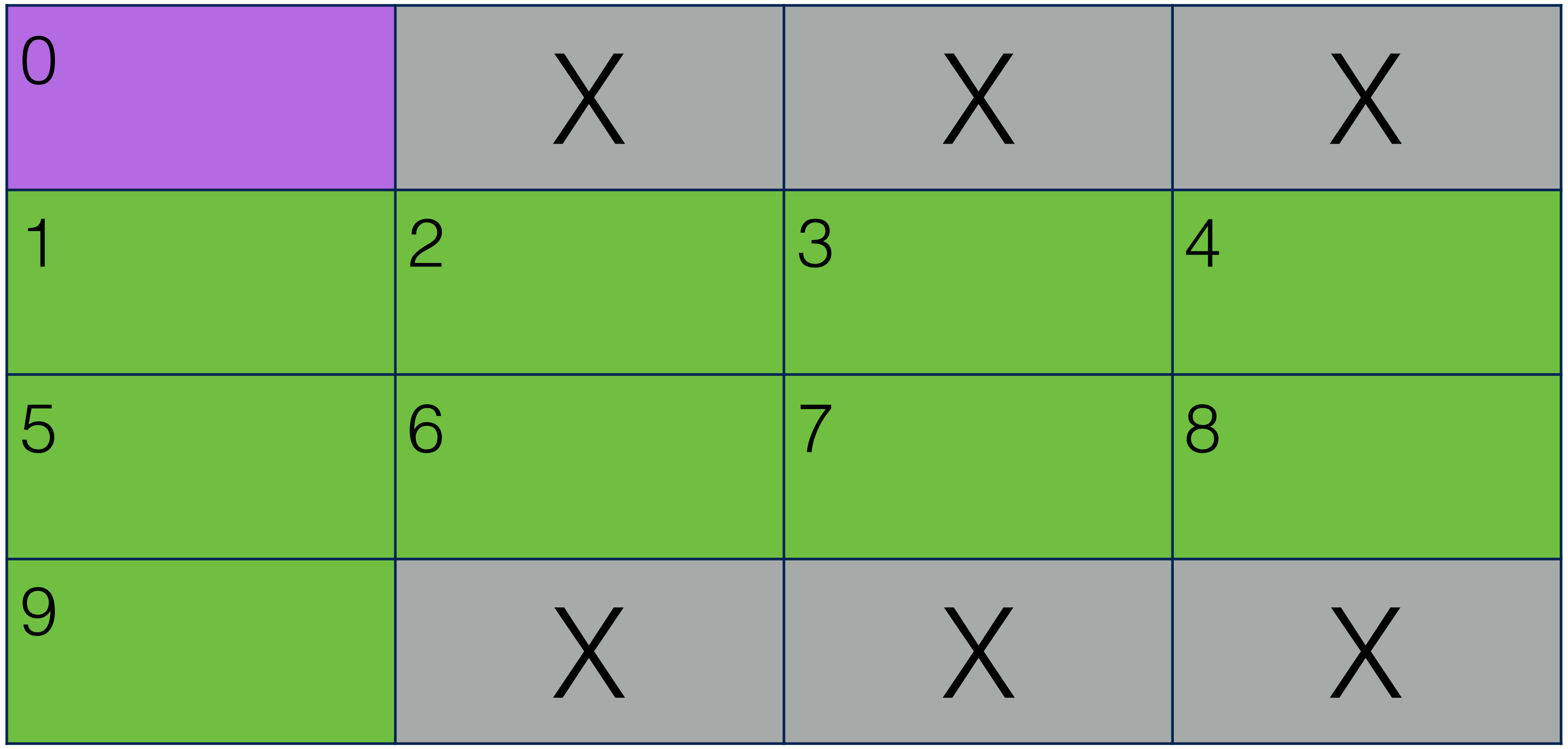


0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Green wins if a solution is achieved

From Puzzle to Game…

Purple player starts, Green players continues

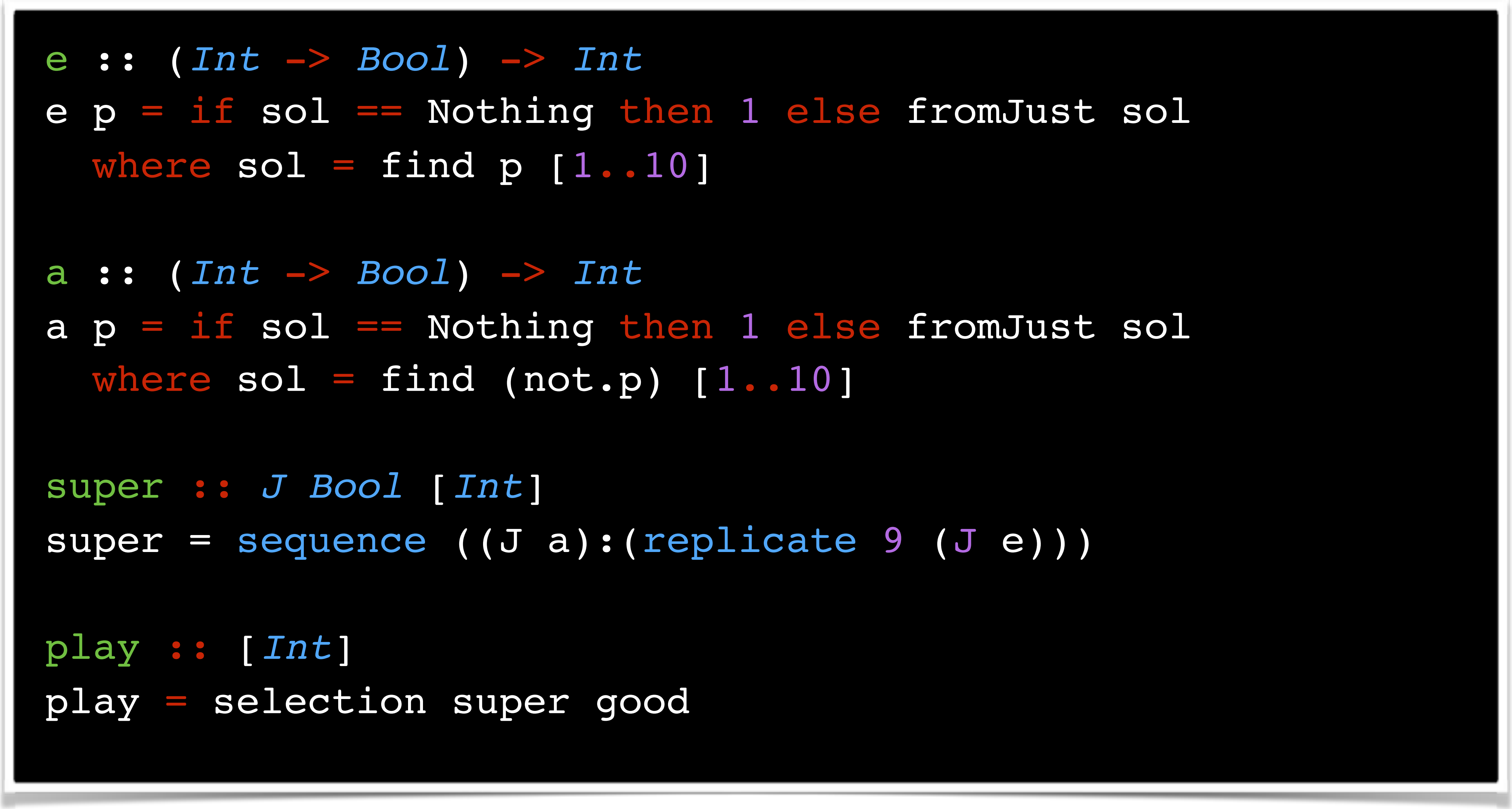


0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Green wins if a solution is achieved

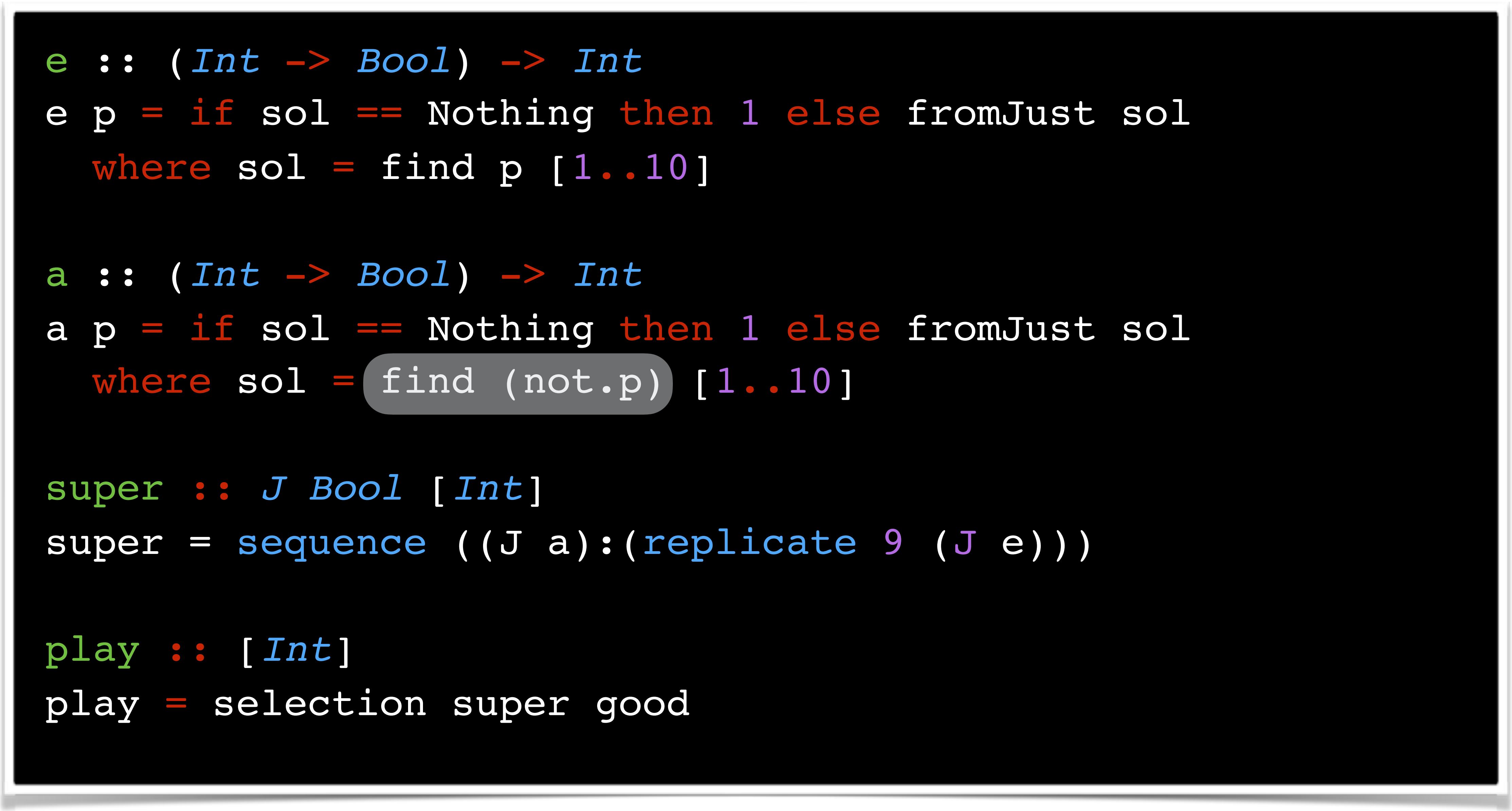
Purple wins otherwise

Haskell Implementation



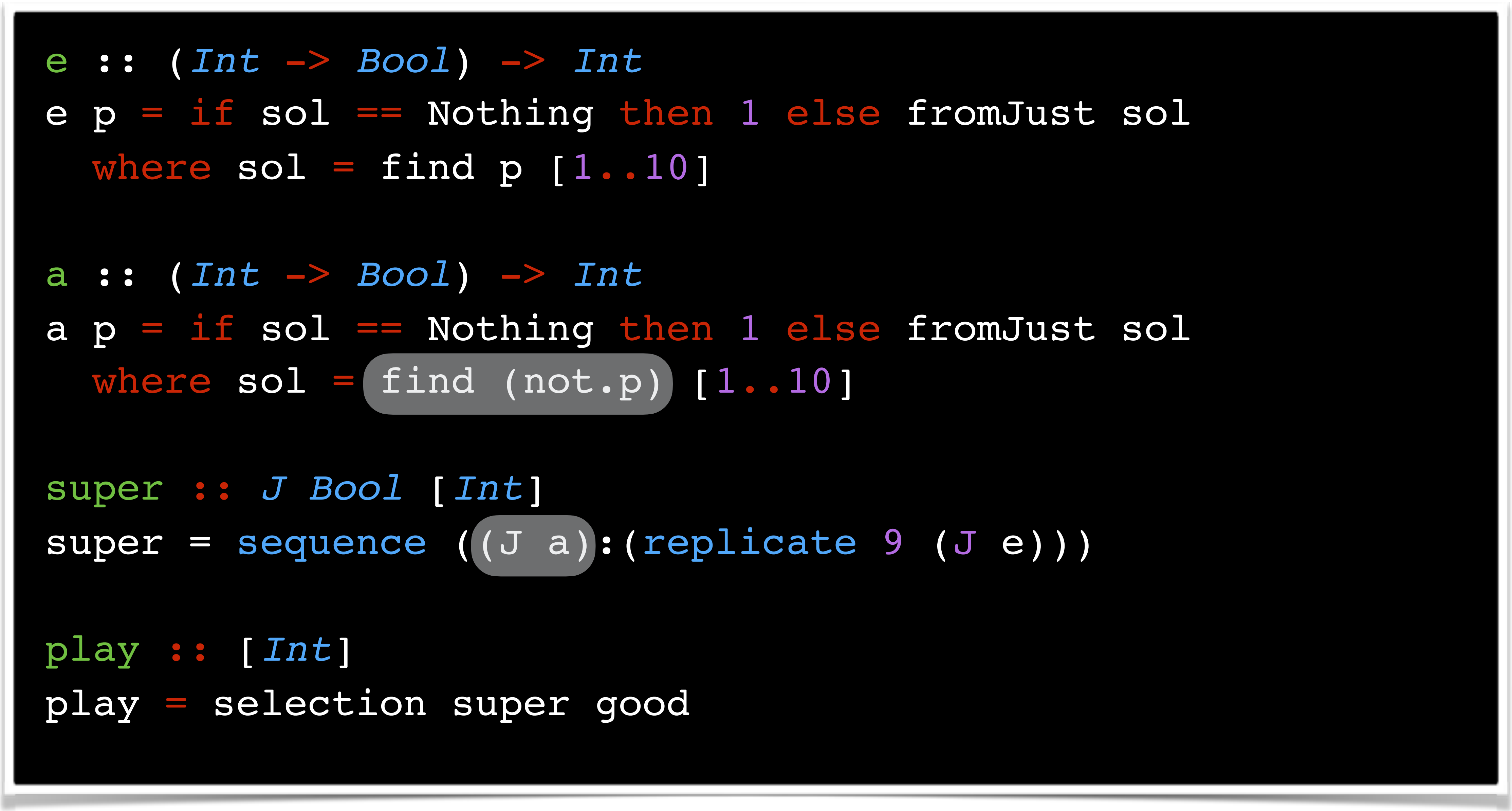
```haskell
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 1 else fromJust sol
  where sol = find p [1..10]

a :: (Int -> Bool) -> Int
a p = if sol == Nothing then 1 else fromJust sol
  where sol = find (not.p) [1..10]

super :: J Bool [Int]
super = sequence ((J a):(replicate 9 (J e)))

play :: [Int]
play = selection super good
```

Haskell Implementation



```
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 1 else fromJust sol
  where sol = find p [1..10]

a :: (Int -> Bool) -> Int
a p = if sol == Nothing then 1 else fromJust sol
  where sol = find (not.p) [1..10]

super :: J Bool [Int]
super = sequence ((J a):(replicate 9 (J e)))

play :: [Int]
play = selection super good
```

Haskell Implementation



```
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 1 else fromJust sol
  where sol = find p [1..10]

a :: (Int -> Bool) -> Int
a p = if sol == Nothing then 1 else fromJust sol
  where sol = find (not.p) [1..10]

super :: J Bool [Int]
super = sequence ((J a):(replicate 9 (J e)))

play :: [Int]
play = selection super good
```

Modular search on more complex games…

Pirates and Treasures[1]

A group of 7 pirates has 100 gold coins

They have to decide amongst themselves how to divide the treasure, but must abide by pirate rules:

- The most senior pirate proposes the division
- All of the pirates (including the most senior) vote on the division
    - If half or more vote for the division, it stands
    - If less than half vote for it, they throw the most senior pirate overboard and start again
- The pirates are perfectly logical, and entirely ruthless (only caring about maximising their own share of the gold)

What division should the most senior pirate suggest to the other six?

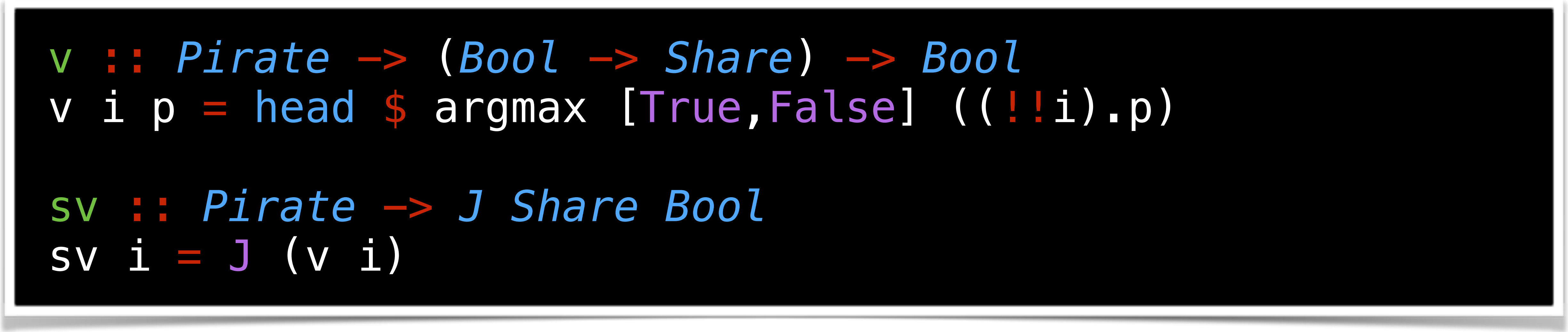
[1] http://www.ox.ac.uk/admissions/undergraduate/applying-to-oxford/interviews/sample-interview-questions

Basic player 1: The voter

- Input
  - Pirate index i
  - Continuation p :: Bool → Share
- Choose boolean that maximises his share

Basic player 1: The voter

- Input
  - Pirate index i
  - Continuation p :: Bool → Share
- Choose boolean that maximises his share



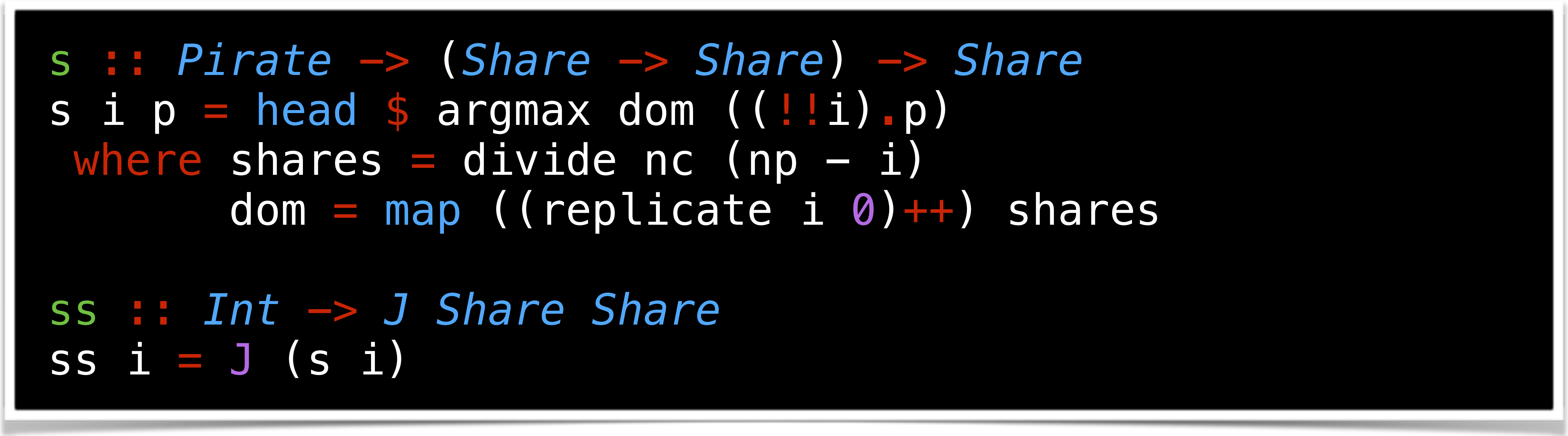
```haskell
v :: Pirate -> (Bool -> Share) -> Bool
v i p = head $ argmax [True,False] ((!!i).p)

sv :: Pirate -> J Share Bool
sv i = J (v i)
```

Basic player 2: The sharer

- Input
  - Pirate index i
  - Continuation p :: Share → Share
- Choose global share that maximises his share

Basic player 2: The sharer

- Input
  - Pirate index i
  - Continuation p :: Share → Share
- Choose global share that maximises his share

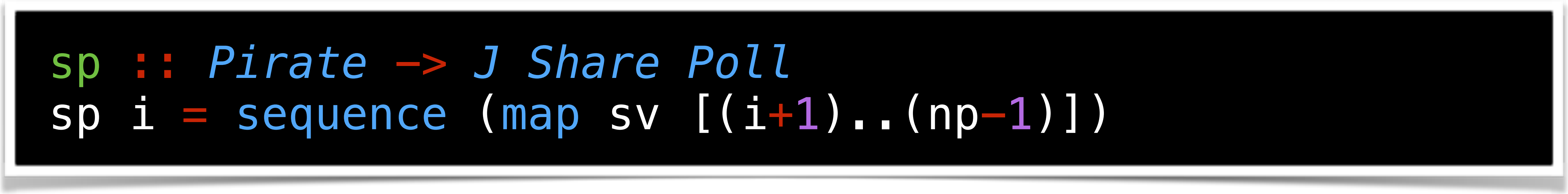


```
s :: Pirate -> (Share -> Share) -> Share
s i p = head $ argmax dom ((!!i).p)
 where shares = divide nc (np - i)
       dom = map ((replicate i 0)++) shares

ss :: Int -> J Share Share
ss i = J (s i)
```

Composing players…

Composing players…

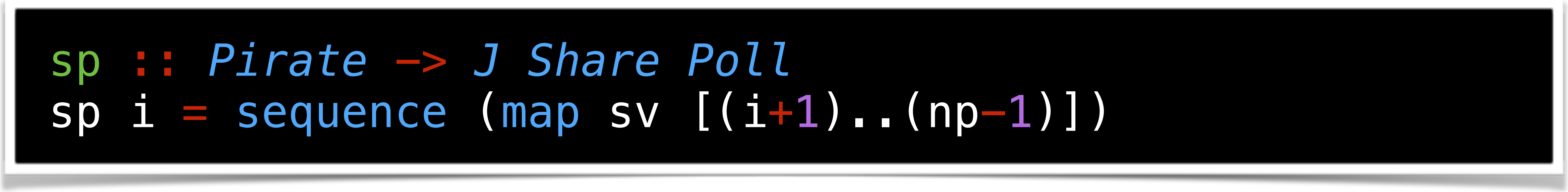
Round player = Product of share and poll players



```
sp :: Pirate -> J Share Poll
sp i = sequence (map sv [(i+1)..(np-1)])
```

Composing players…

Round player = Product of share and poll players

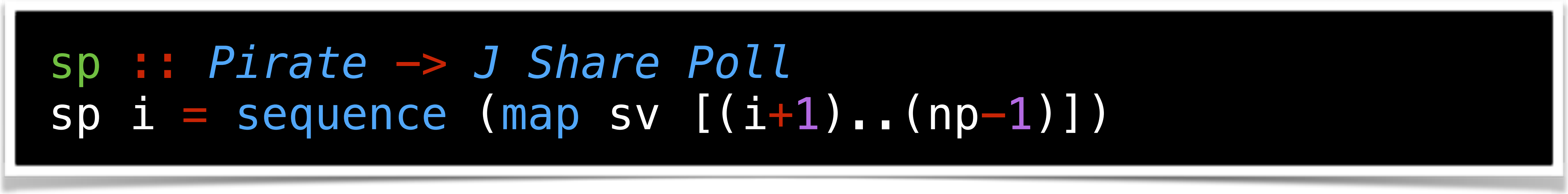


```
sp :: Pirate -> J Share Poll
sp i = sequence (map sv [(i+1)..(np-1)])
```

Poll player = sequencing of voters

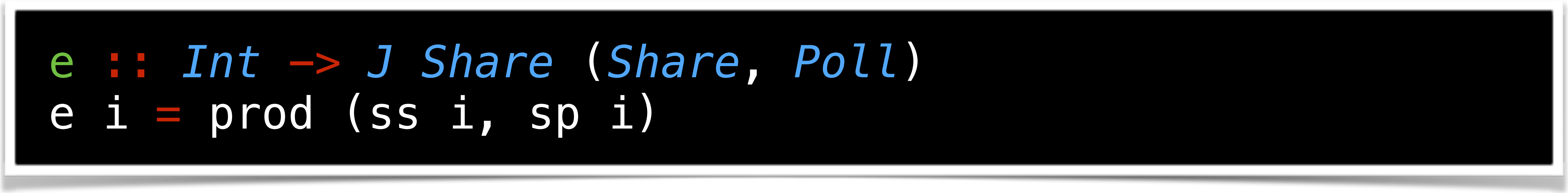
Composing players…

Round player = Product of share and poll players



```
sp :: Pirate -> J Share Poll
sp i = sequence (map sv [(i+1)..(np-1)])
```

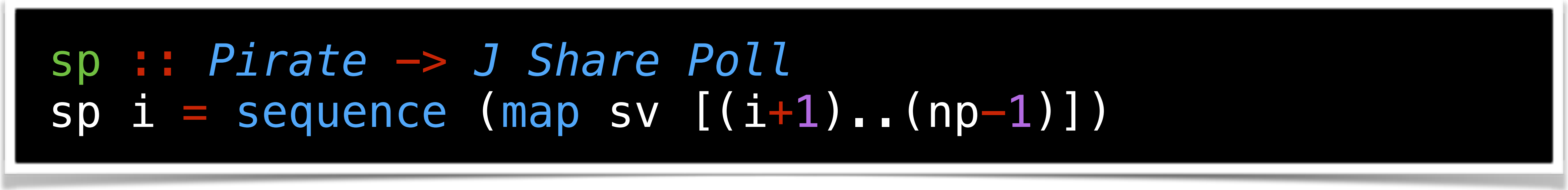
Poll player = sequencing of voters



```
e :: Int -> J Share (Share, Poll)
e i = prod (ss i, sp i)
```

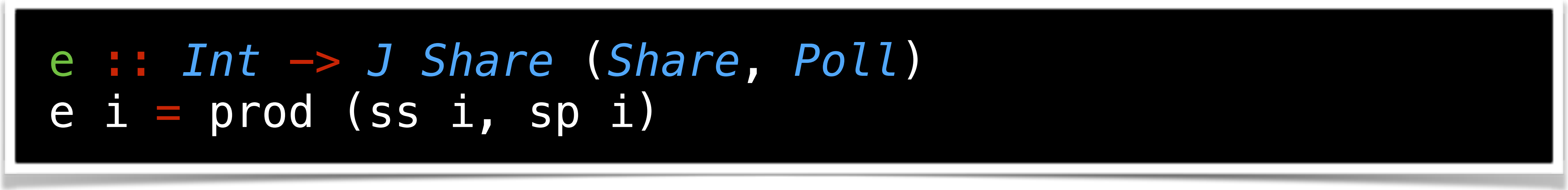
Composing players…

Round player = Product of share and poll players



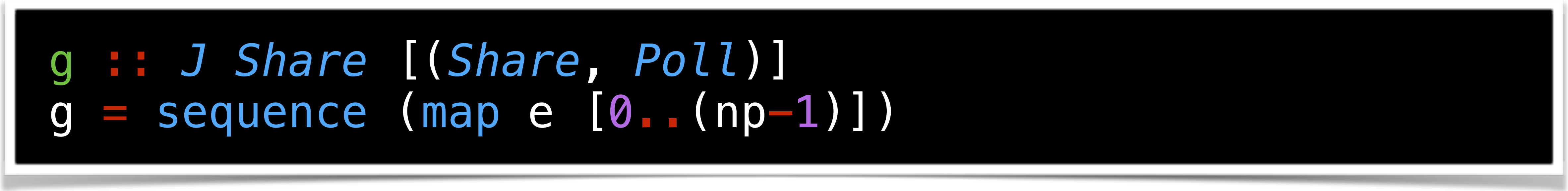
```
sp :: Pirate -> J Share Poll
sp i = sequence (map sv [(i+1)..(np-1)])
```

Poll player = sequencing of voters



```
e :: Int -> J Share (Share, Poll)
e i = prod (ss i, sp i)
```

Global player = Sequence of round players

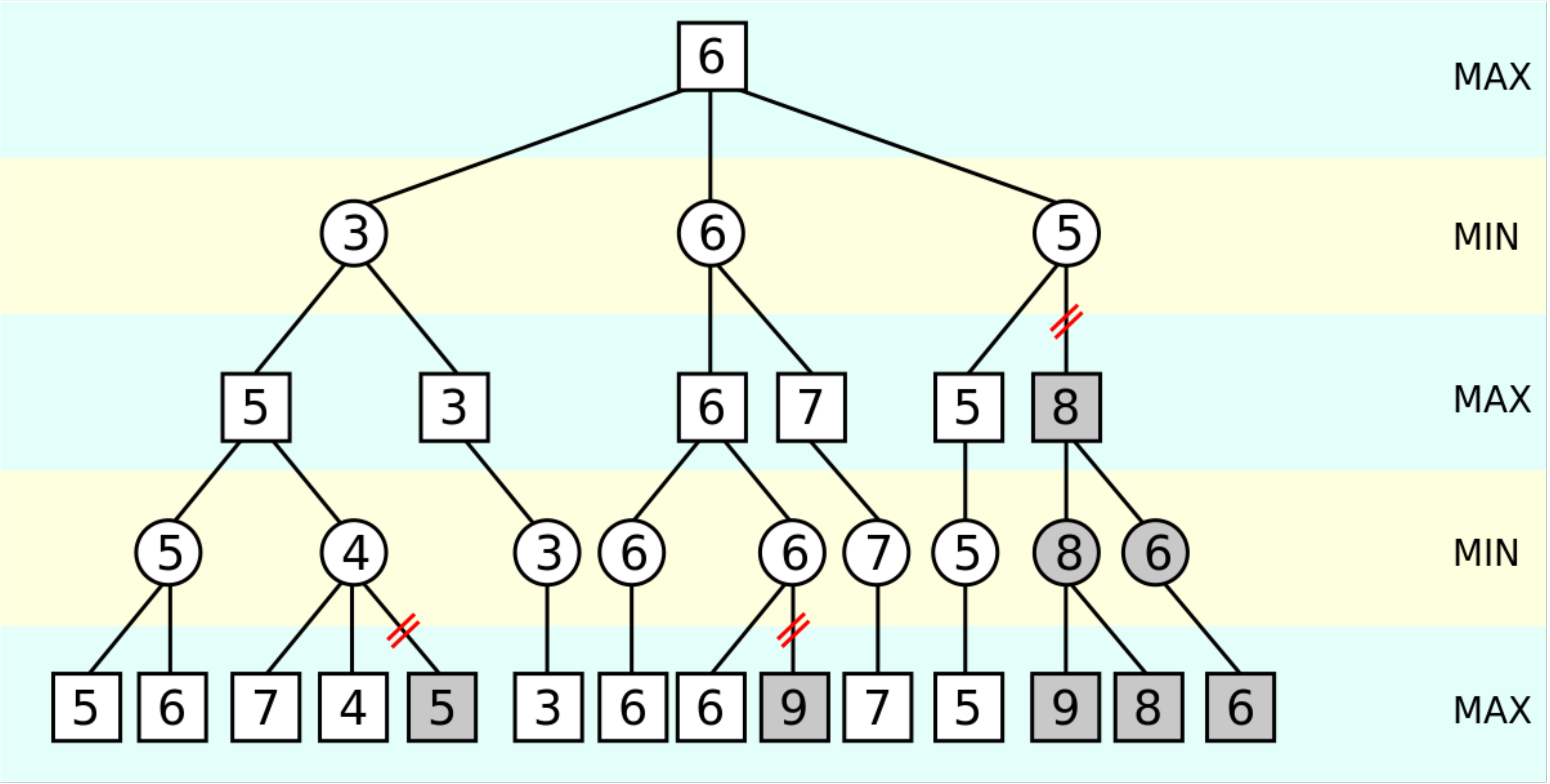


```
g :: J Share [(Share, Poll)]
g = sequence (map e [0..(np-1)])
```

alpha-beta pruning

alpha-beta pruning

- Prunes search tree on zero-sum two player games
- E.g. state-of-the-art chess programs use it
- Idea:
  - Continuing a sub-search will only improve my payoff
  - If current payoff already discourages opponent to visit sub-tree
  - Then may as well give up searching sub-tree



https://commons.wikimedia.org/wiki/File:AB_pruning.svg

alpha-beta pruning

Keep a record of alpha-beta values for each move

$$Y = X \times (\mathbb{N} \times \mathbb{N})$$

$$R = \mathbb{N}$$

Corresponds to doing a search using

$$\phi :: X \times \mathbb{N} \times \mathbb{N} \to K_R(X \times \mathbb{N} \times \mathbb{N})$$

$$\varepsilon :: X \times \mathbb{N} \times \mathbb{N} \to J_R(X \times \mathbb{N} \times \mathbb{N})$$

Summary

Summary

- Selection/continuation monads perform "local search" and modelling of players

Summary

- Selection/continuation monads perform “local search” and modelling of players
- Sequencing of selection/continuation monad gives

Summary

- Selection/continuation monads perform “local search” and modelling of players
- Sequencing of selection/continuation monad gives
    - Efficient global search

Summary

- <u>Selection/continuation monads</u> perform “local search” and modelling of players
- <u>Sequencing of selection/continuation monad</u> gives
  - Efficient global search
  - Implementation of backward induction

Summary

- Selection/continuation monads perform “local search” and modelling of players
- Sequencing of selection/continuation monad gives
    - Efficient global search
    - Implementation of backward induction
    - Computational interpretation of countable choice

Summary

- Selection/continuation monads perform “local search” and modelling of players
- Sequencing of selection/continuation monad gives
  - Efficient global search
  - Implementation of backward induction
  - Computational interpretation of countable choice
  - Computational version of Tychonoff’s theorem

References

- Escardó and Oliva. *Selection functions, bar recursion and backward induction*. Mathematical Structures in Computer Science, 20(2):127-168, 2010

- Escardó and Oliva. *Sequential games and optimal strategies*. Proceedings of the Royal Society A, 467:1519-1545, 2011

- Hedges, Oliva, Sprits, Zahn, and Winschel. *A higher-order framework for decision problems and games*, ArXiv, http://arxiv.org/abs/1409.7411, 2014