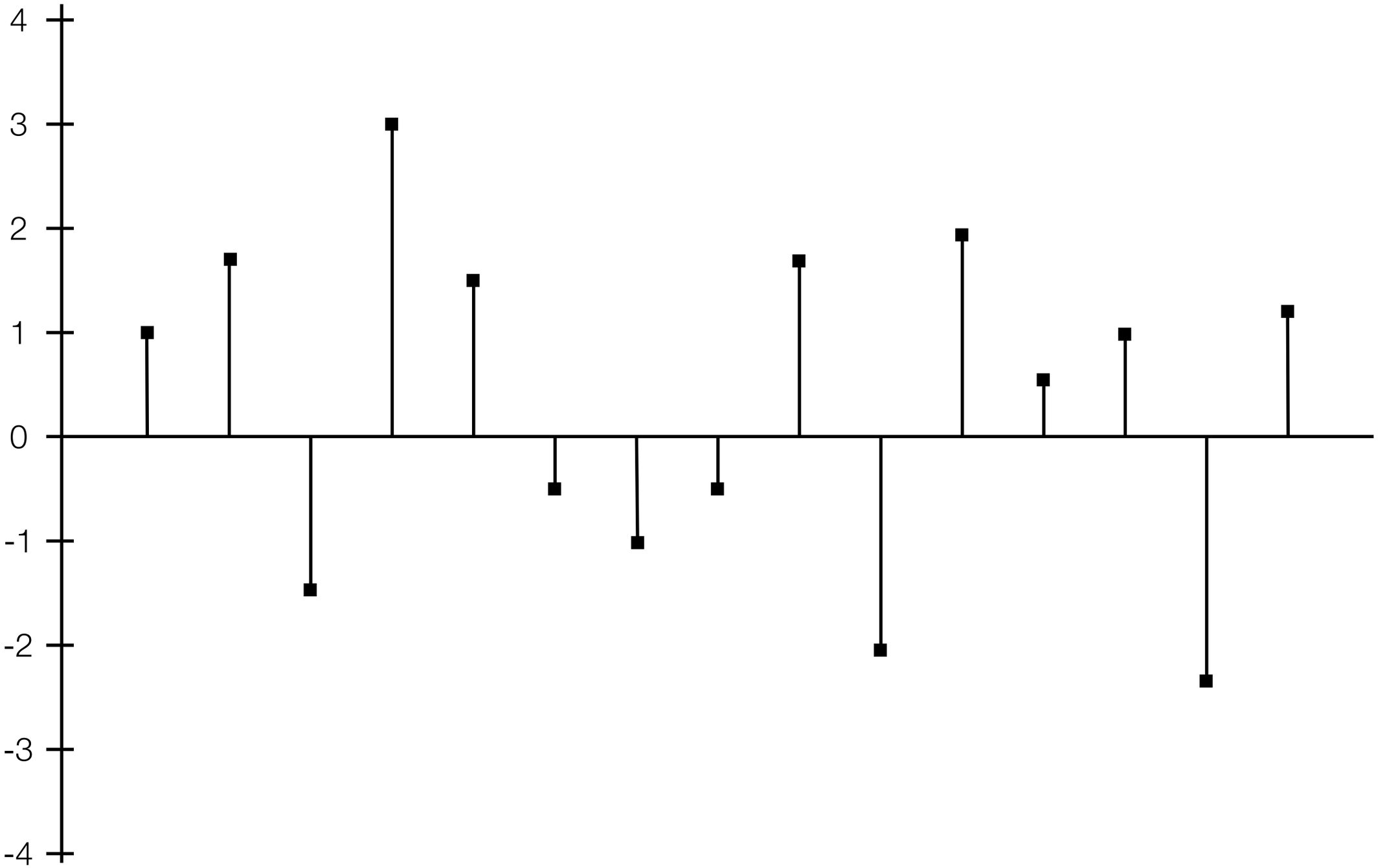
Data Flow Fusion with Series Expressions

Ben Lippmeier, Manuel Chakravarty
Gabriele Keller, Amos Robinson

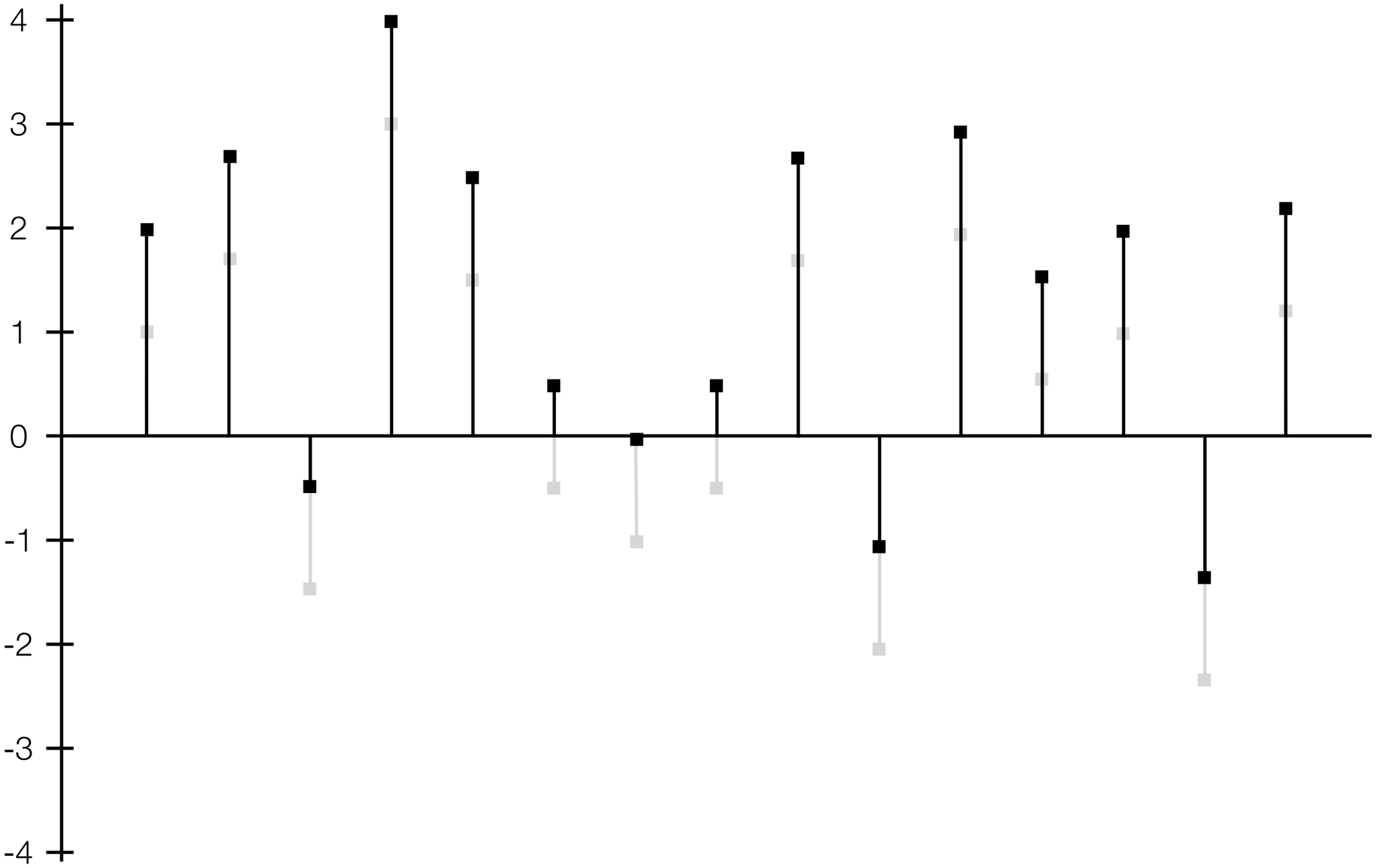
University of New South Wales

Haskell Symposium 2013

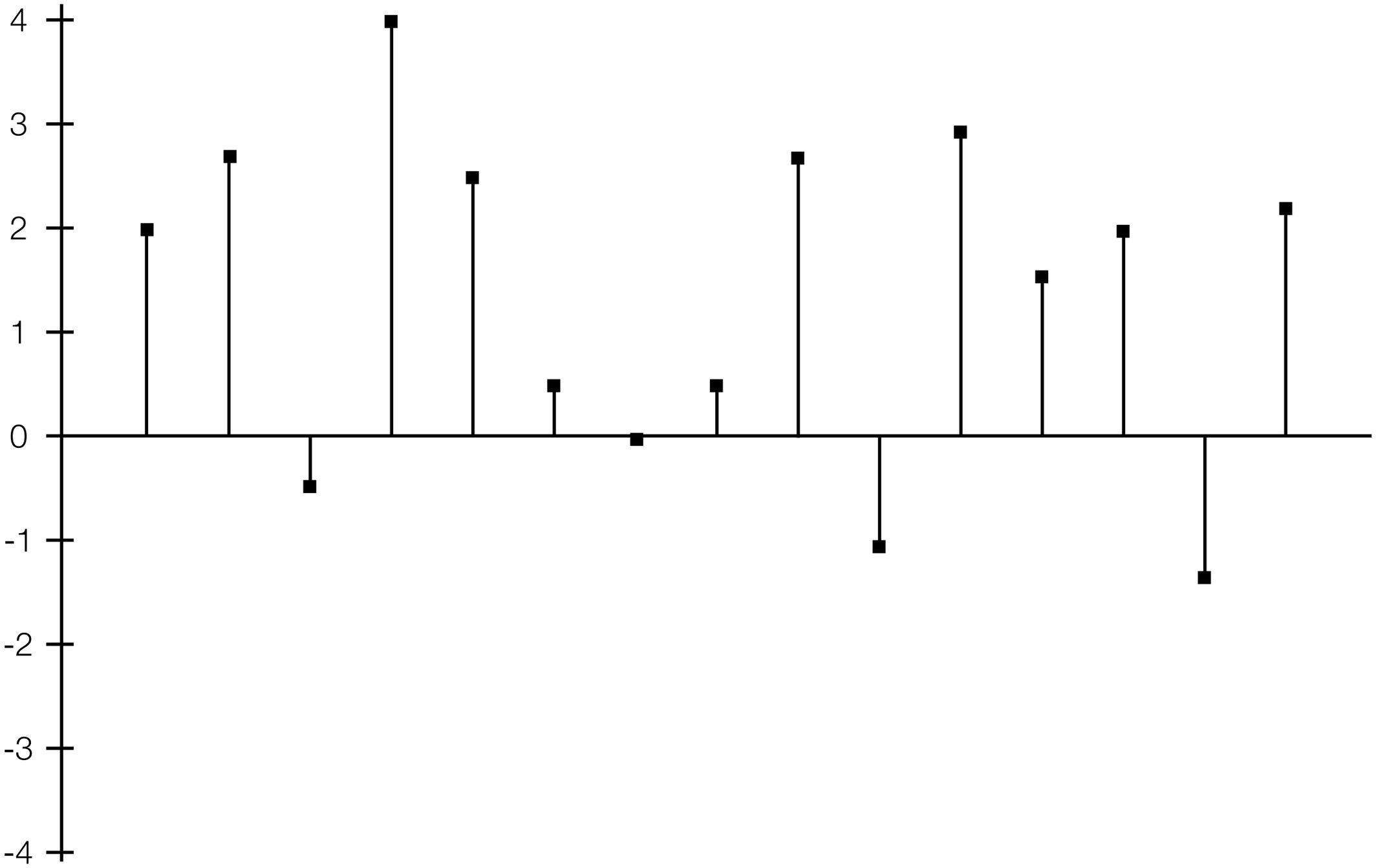
FilterMax



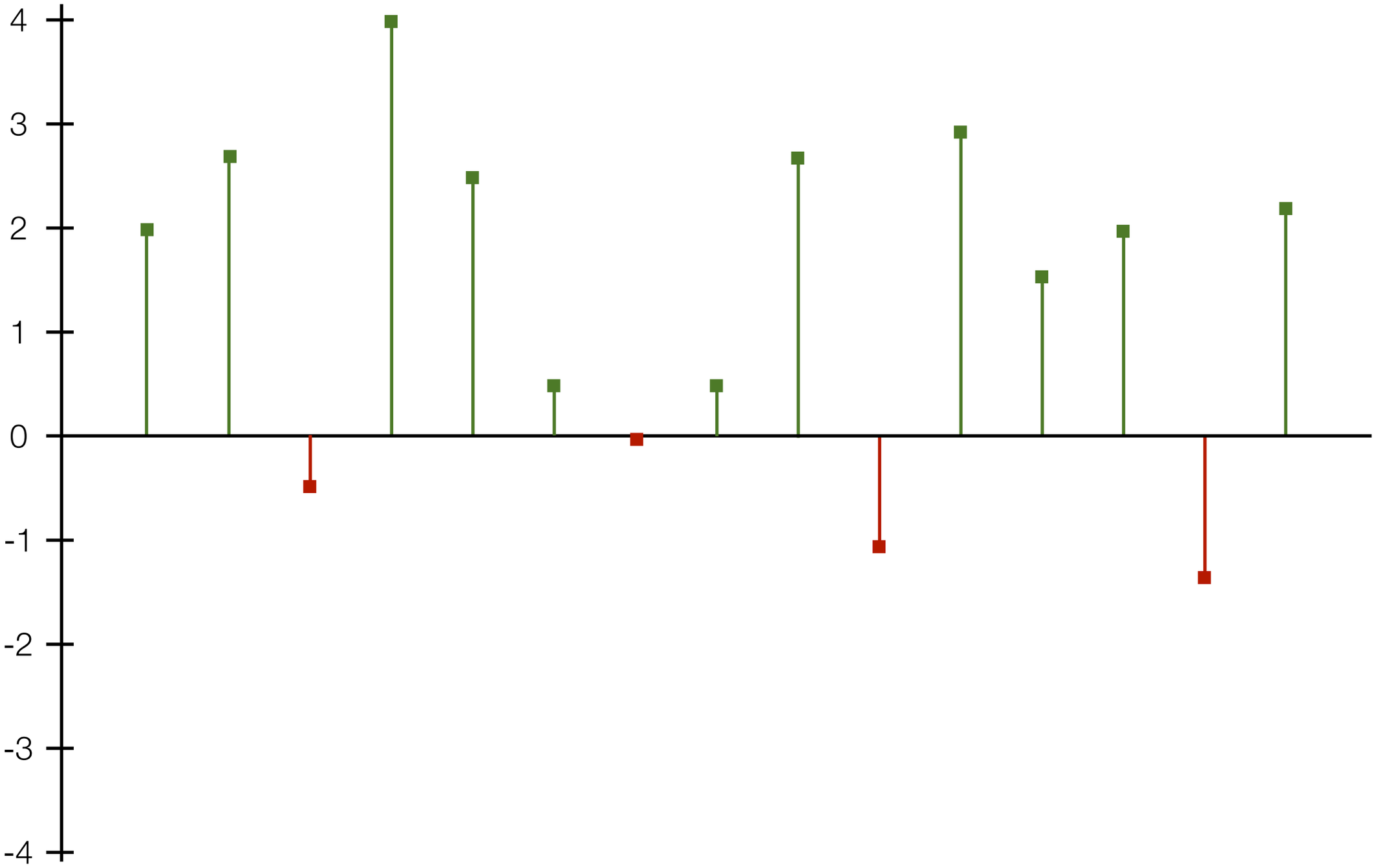
FilterMax



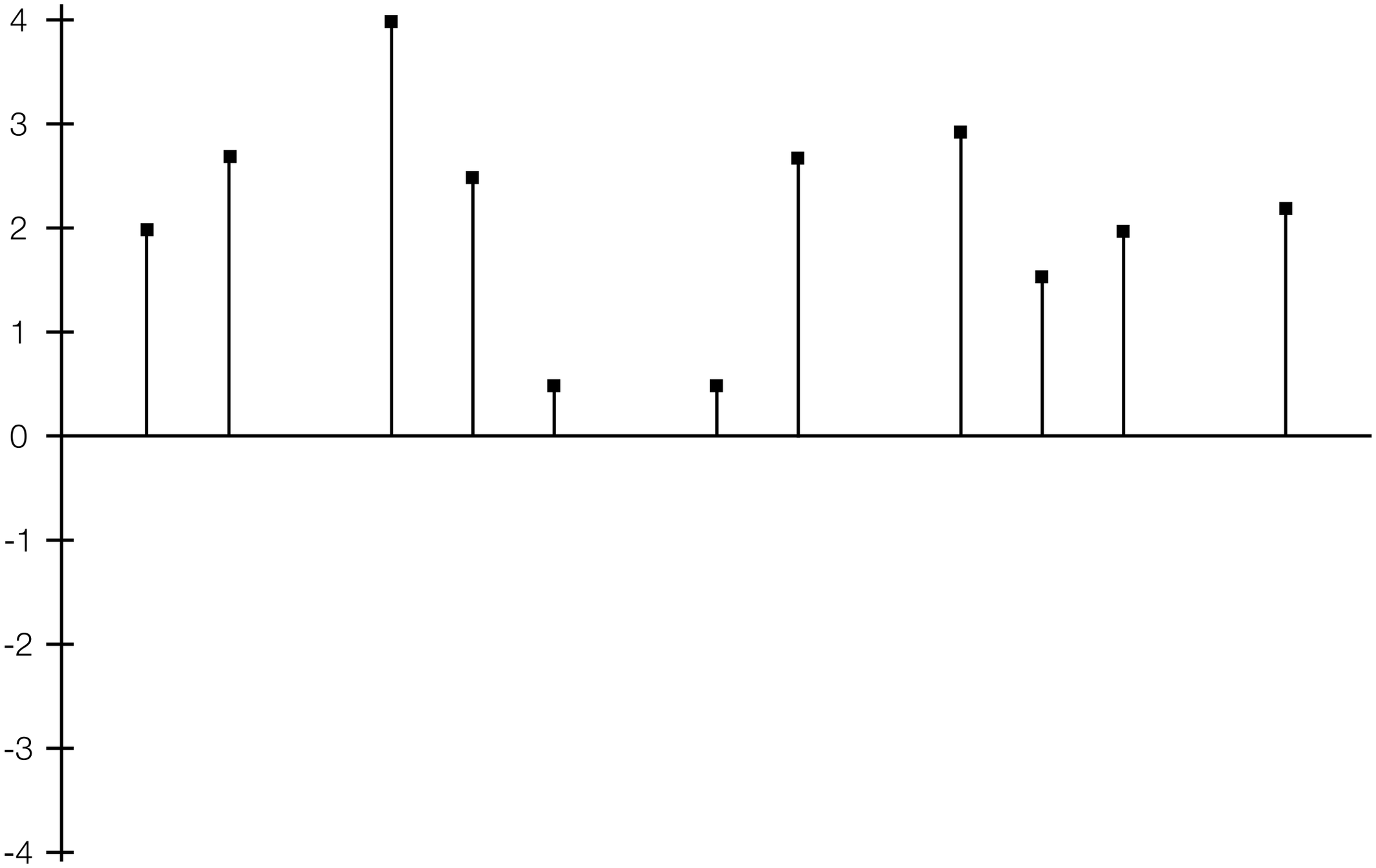
FilterMax



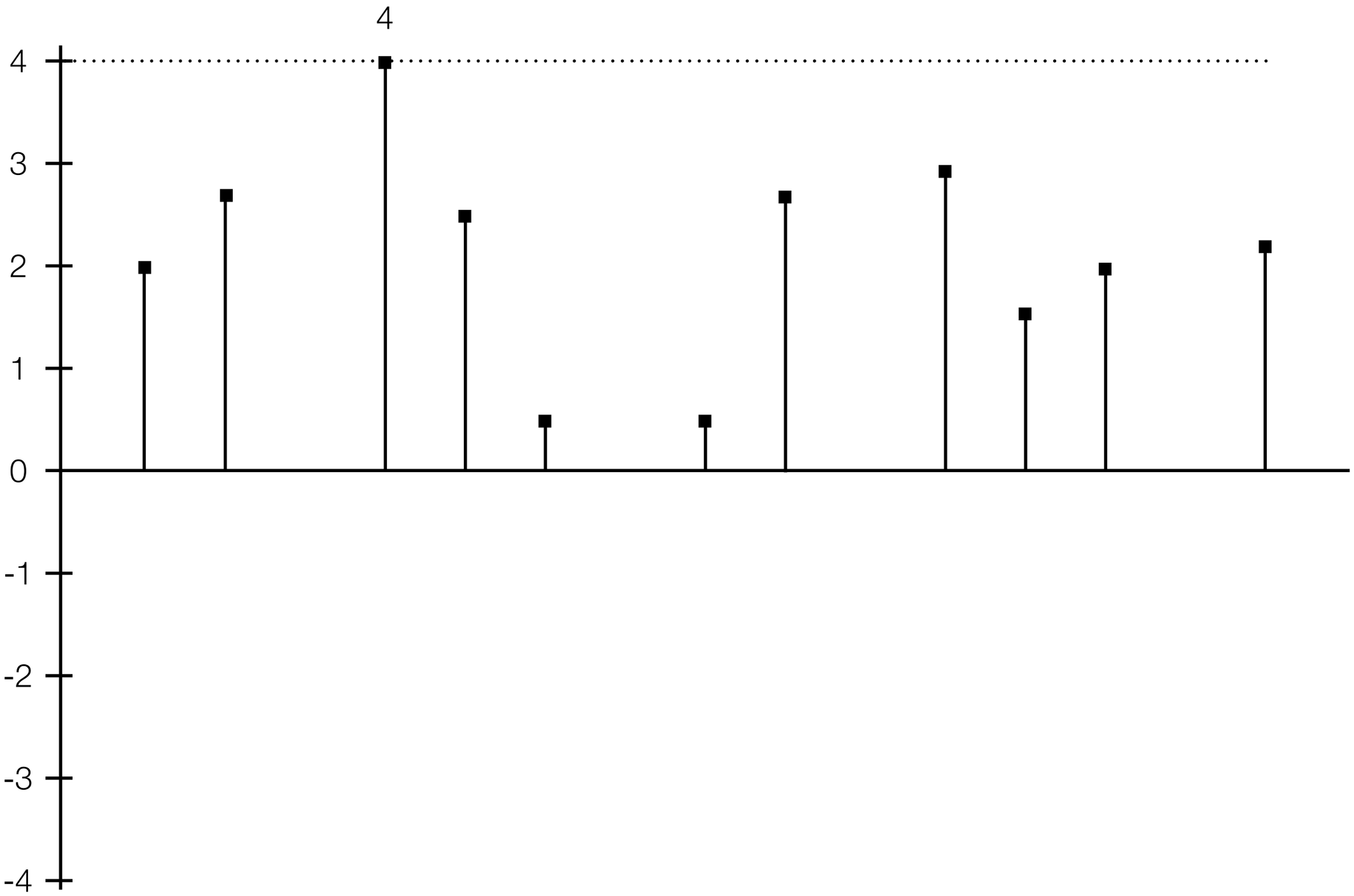
FilterMax



FilterMax



FilterMax



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2 = map (+ 1) vec1
       vec3 = filter (> 0) vec2
       n    = fold max 0 vec3
   in  (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2 = map (+ 1) vec1
       vec3 = filter (> 0) vec2
       n    = fold max 0 vec3
   in  (vec3, n)
```

```
map f    = unstream . mapS f    . stream
filter p = unstream . filterS p . stream
fold f z = foldS f z . stream
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2 = unstream (mapS (+ 1) (stream vec1))
       vec3 = unstream (filterS (> 0) (stream vec2))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```
map f    = unstream . mapS f    . stream
filter p = unstream . filterS p . stream
fold f z = foldS f z . stream
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                    (stream vec1)))))
       n    = foldS max 0 (stream vec3)
  in  (vec3, n)
```

```
map f    = unstream . mapS f    . stream
filter p = unstream . filterS p . stream
fold f z = foldS f z . stream
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                    (stream vec1)))))
       n    = foldS max 0 (stream vec3)
  in  (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                    (stream vec1)))))
       n    = foldS max 0 (stream vec3)
  in  (vec3, n)
```

```
RULE "stream/unstream"
     forall xs. stream (unstream xs) = xs
```

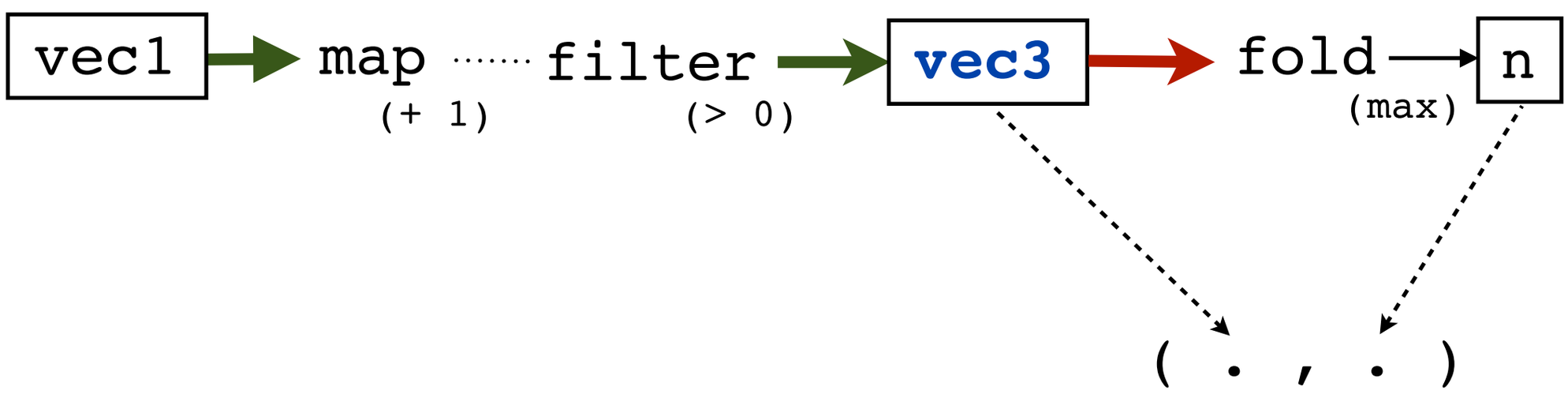
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                    (stream vec1)))))
       n    = foldS max 0 (stream vec3)
  in  (vec3, n)
```

```
RULE "stream/unstream"
     forall xs. stream (unstream xs) = xs
```

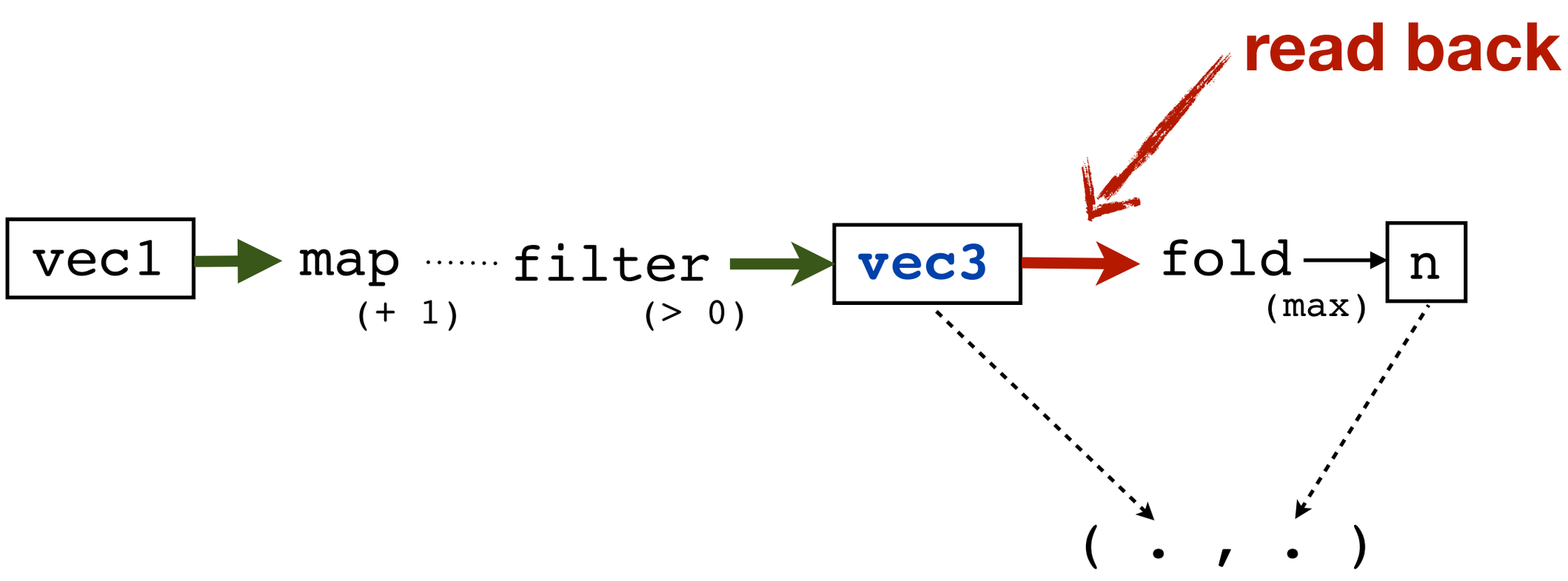
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                      (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                      (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                      (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

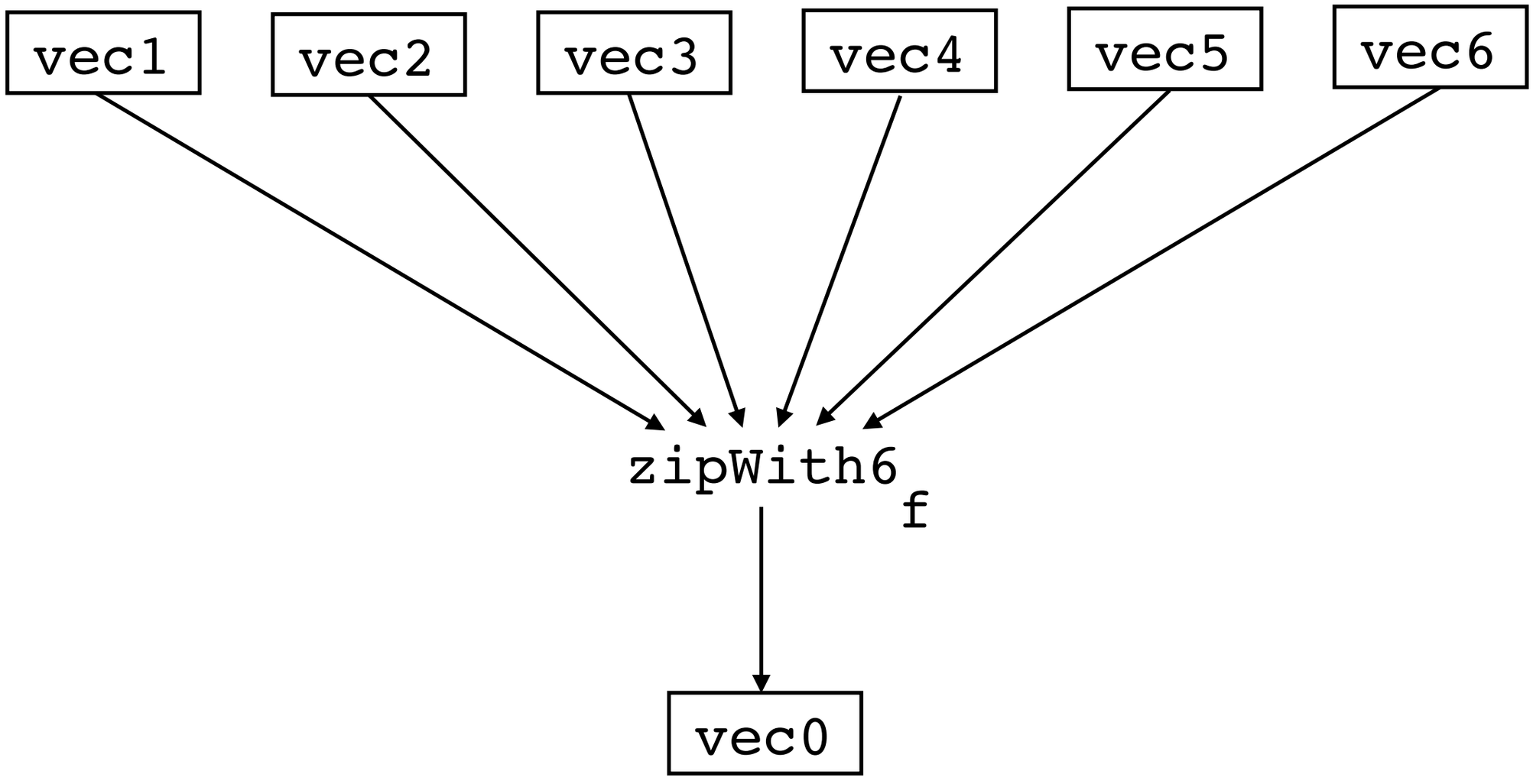


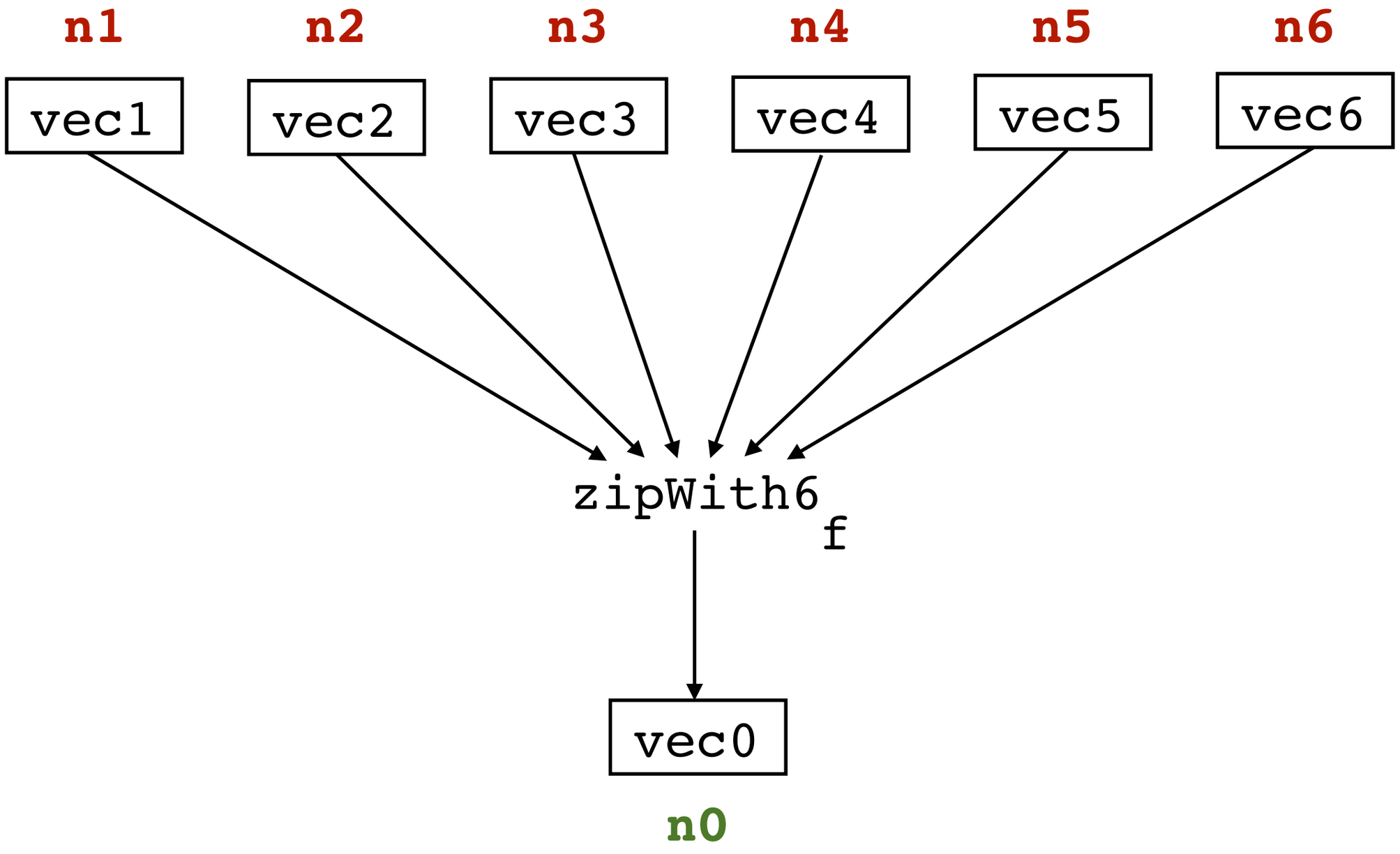
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                      (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

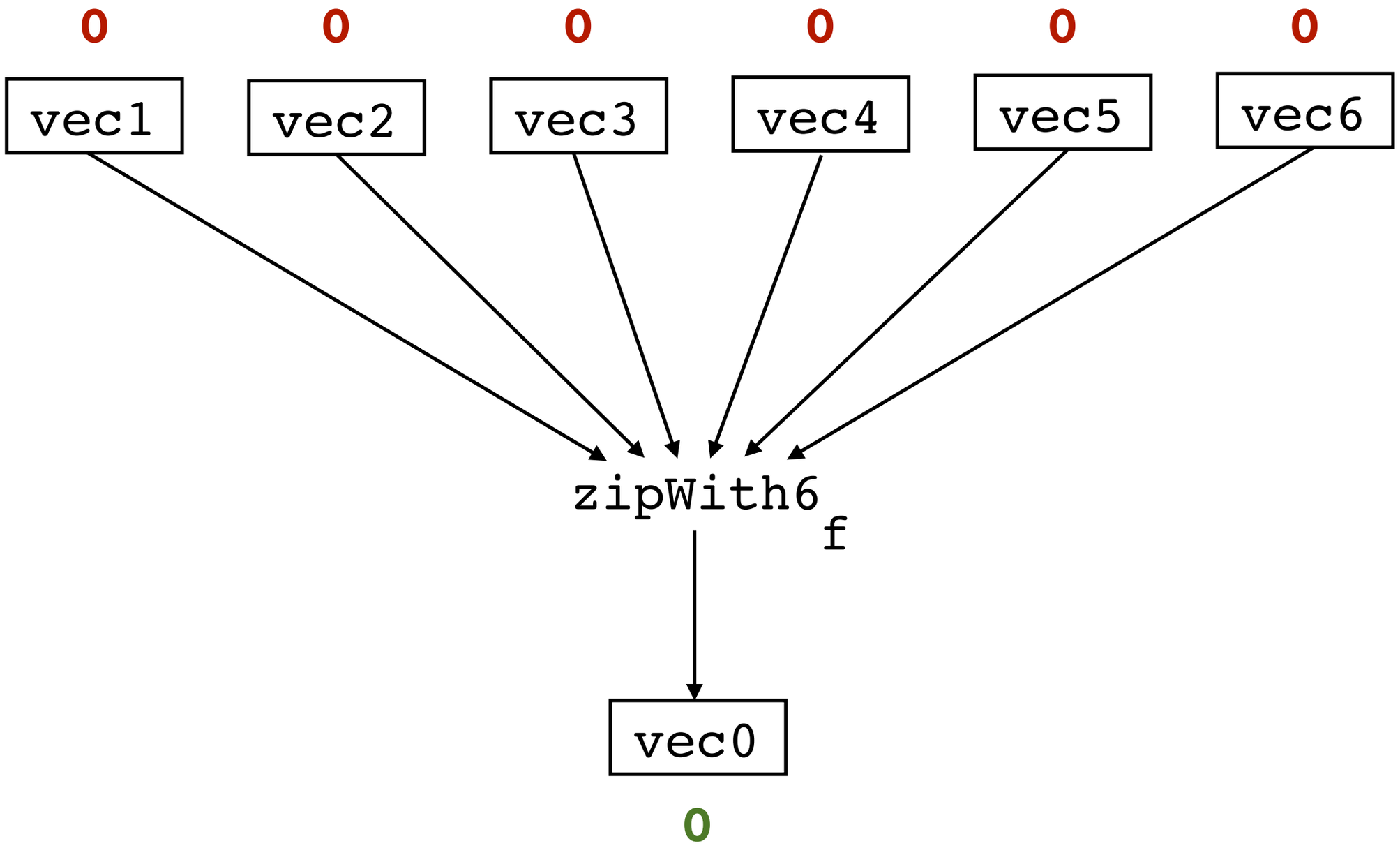


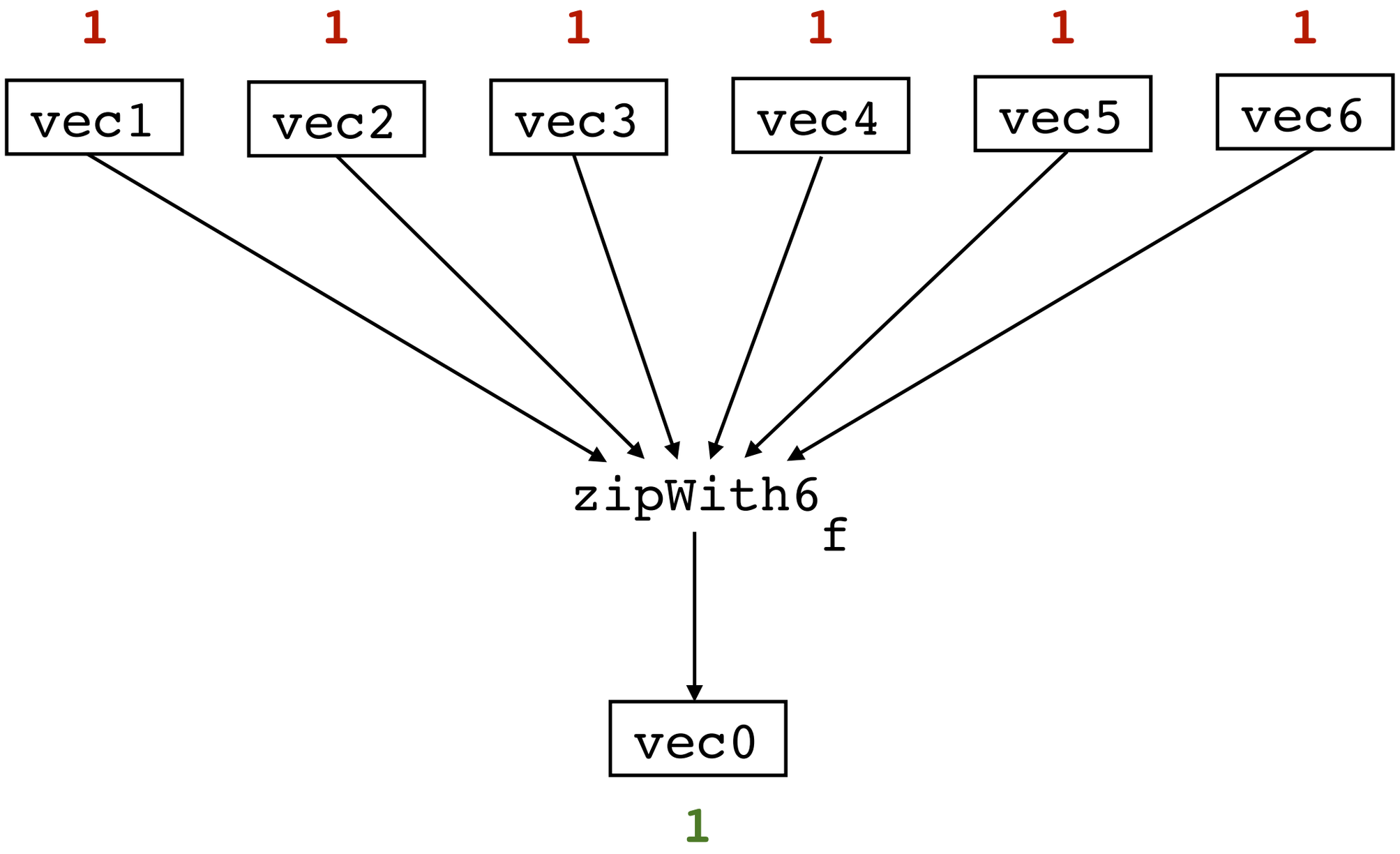
Problem 1

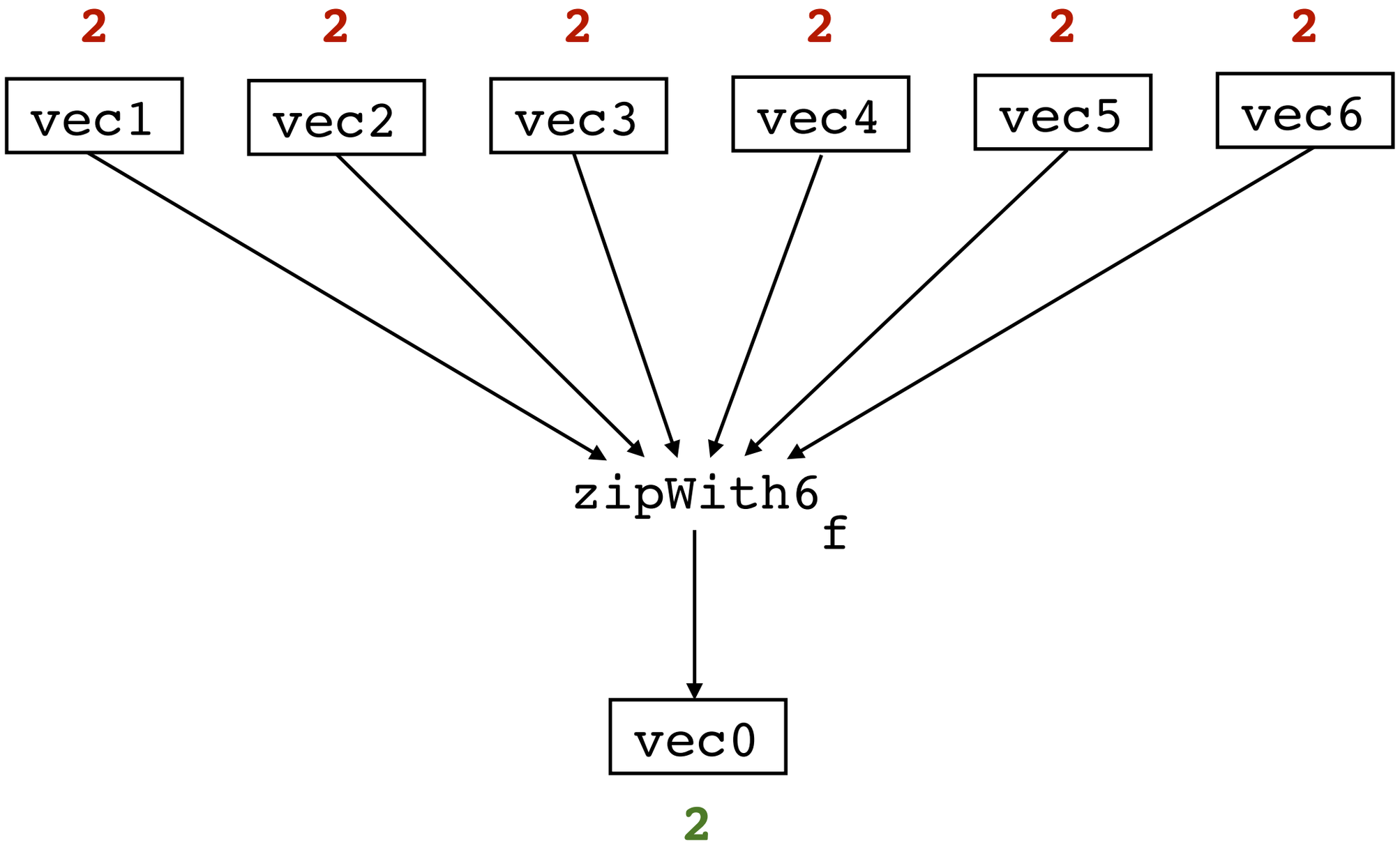
Stream fusion cannot fuse a producer into multiple consumers

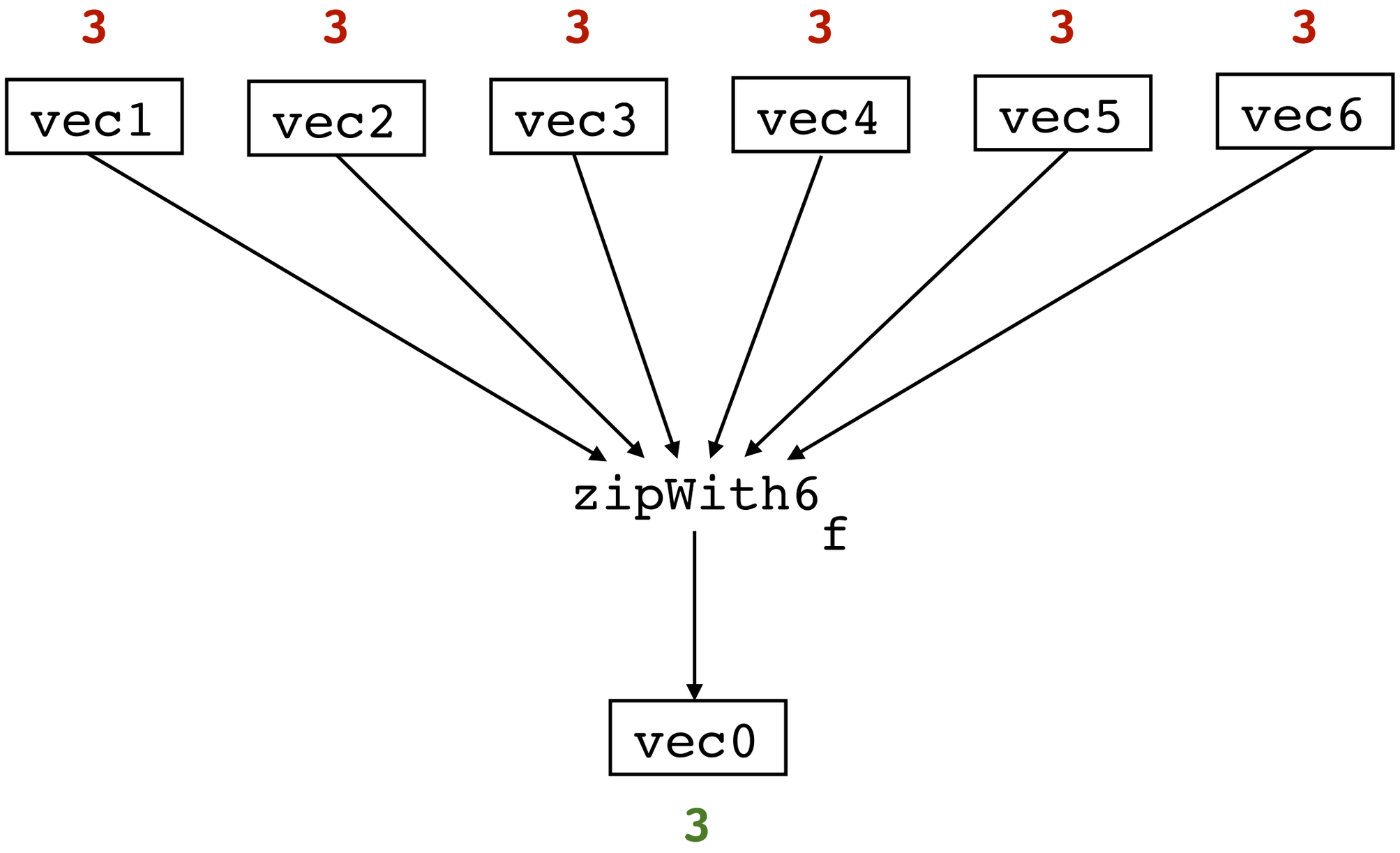

vec1
vec2
vec3
vec4
vec5
vec6
zipWith6
f
vec0


n1
n2
n3
n4
n5
n6
vec1
vec2
vec3
vec4
vec5
vec6
zipWith6
f
vec0
n0


0
0
0
0
0
0
vec1
vec2
vec3
vec4
vec5
vec6
zipWith6
f
vec0
0


1
1
1
1
1
1
vec1
vec2
vec3
vec4
vec5
vec6
zipWith6
f
vec0
1


2
2
2
2
2
2
vec1
vec2
vec3
vec4
vec5
vec6
zipWith6
f
vec0
2


3
3
3
3
3
3
vec1
vec2
vec3
vec4
vec5
vec6
zipWith6
f
vec0
3

Problem 2

Stream fusion produces many duplicate loop counters

Data Flow Fusion

1. **Refactor** slightly to expose desired data flow.

(currently working to make this automatic)

2. **Slurp** out a data flow graph from the source.

3. **Schedule** the graph into an abstract loop nest.

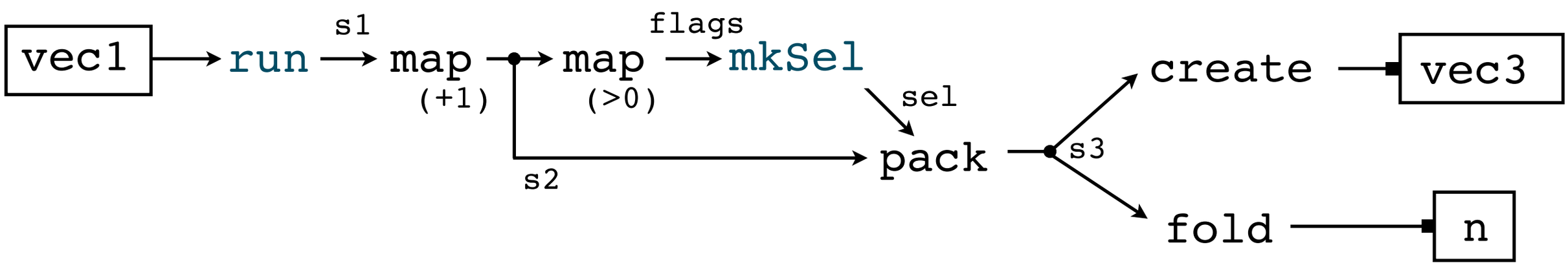
4. **Extract** implementation code from the nest.

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```

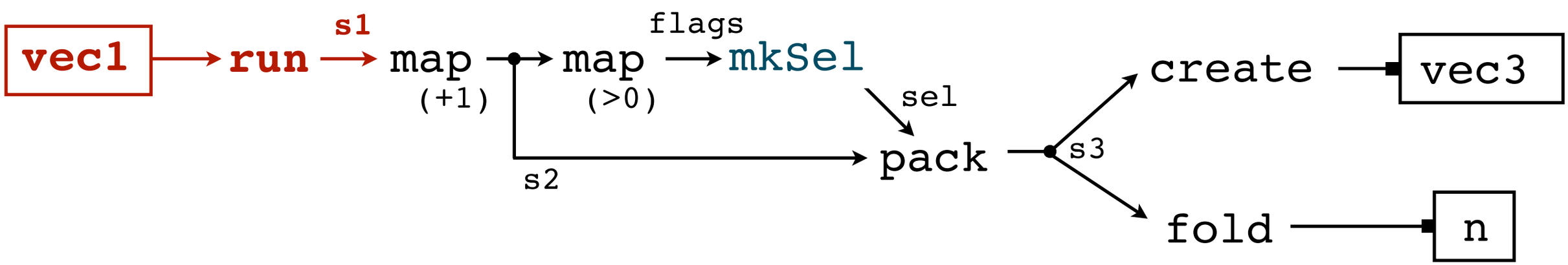
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
    in (vec3, n)))
```

```
pack (Sel [T F F T F]) [1 2 3 4 5] = [1 4]
```

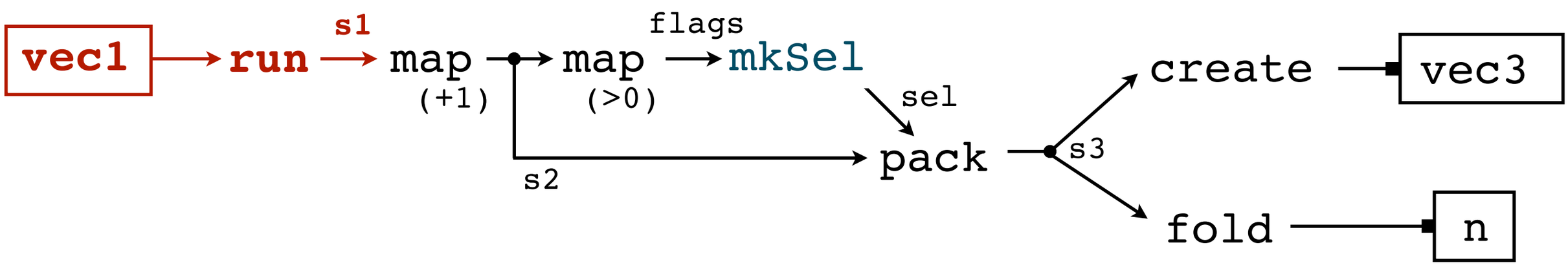
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```

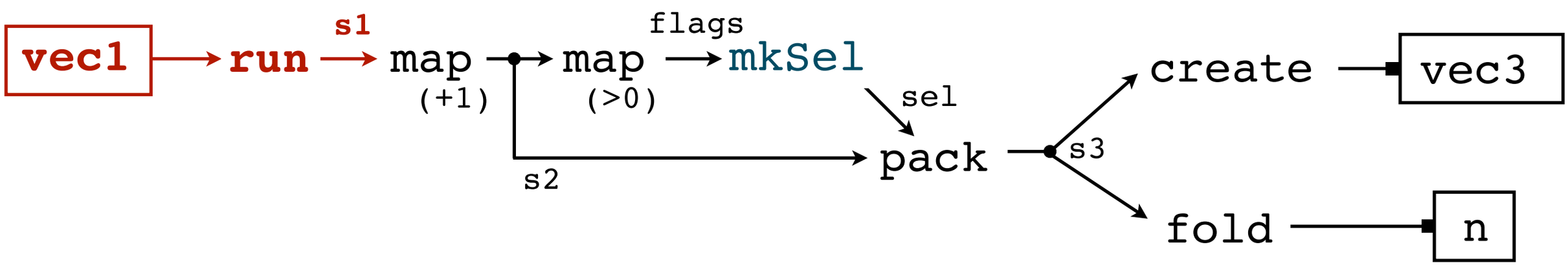


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```



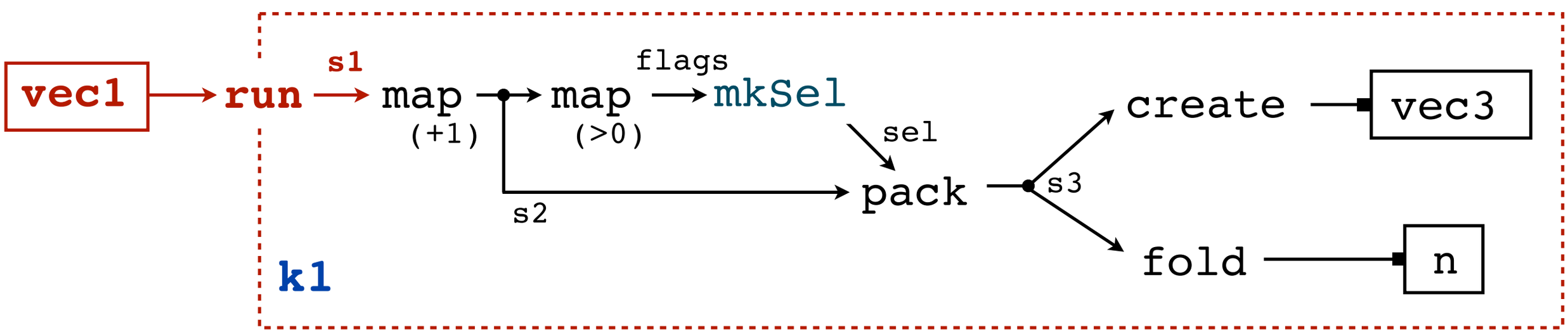
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```

Rate Variable

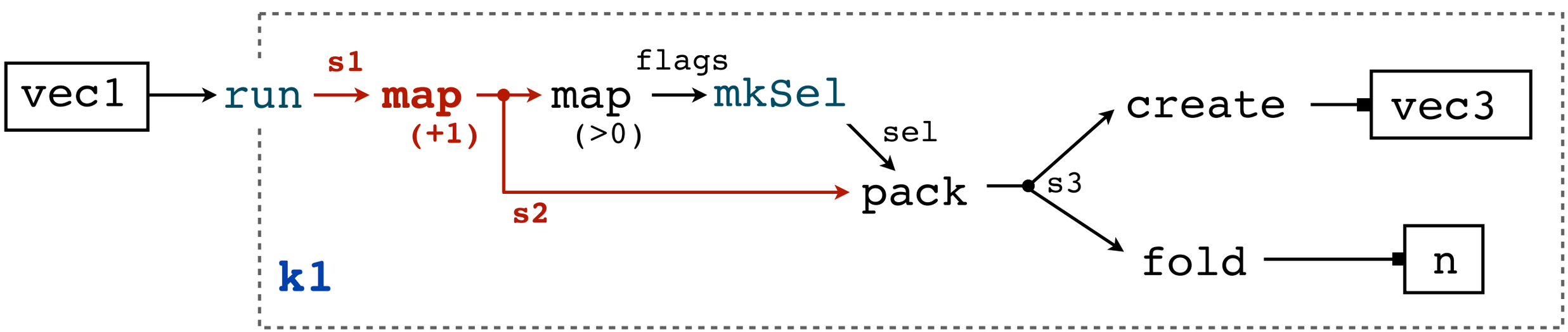


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
    in (vec3, n)))
```

Rate Variable

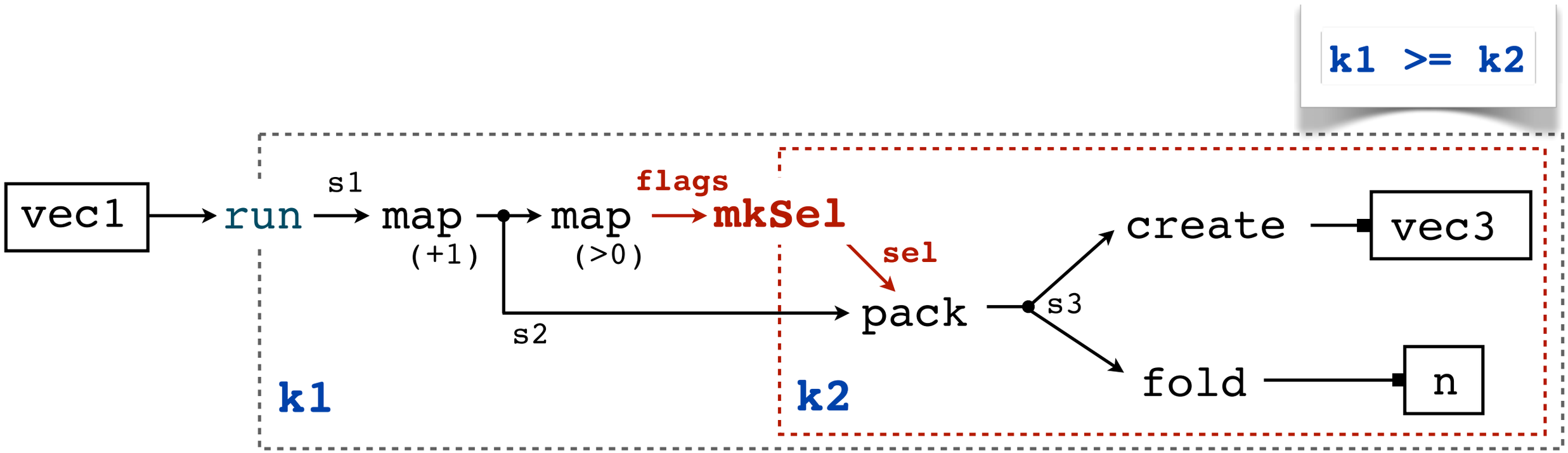


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2     = map (+ 1) s1                s2 :: Series k1 Int
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3    = pack sel s2
             vec3 = create s3
             n     = fold max 0 s3
    in (vec3, n)))
```

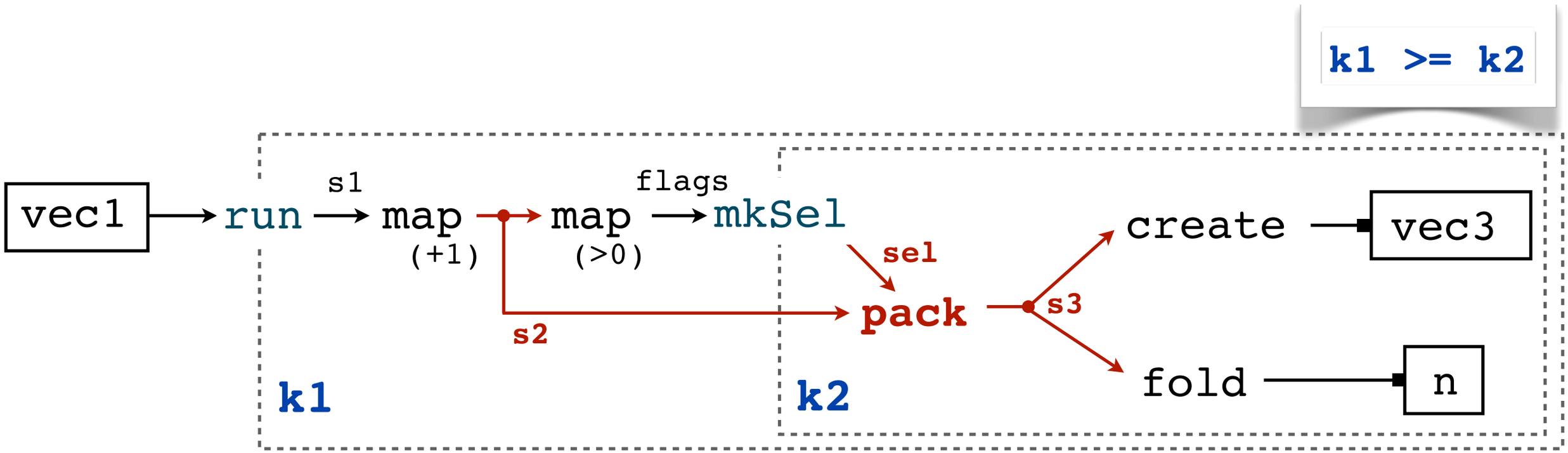


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1                 s2 :: Series k1 Int
        flags = map (> 0) s2           flags :: Series k1 Bool
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```

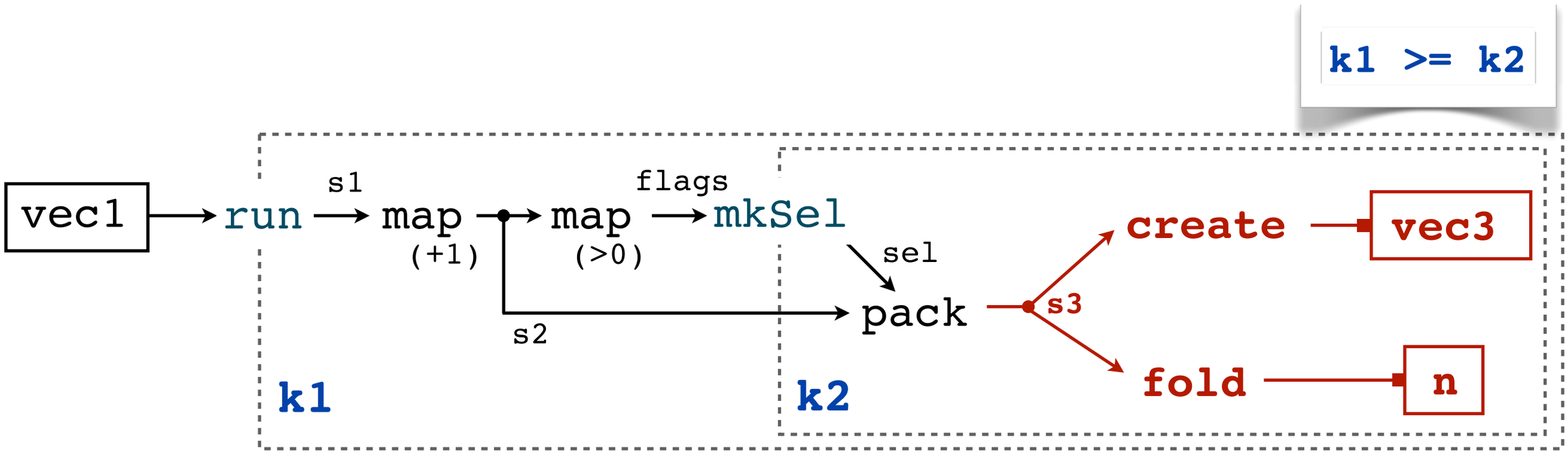
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1                 s2 :: Series k1 Int
        flags = map (> 0) s2           flags :: Series k1 Bool
    in  mkSel flags (\sel ->             sel :: Sel k1 k2
         let s3   = pack sel s2
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```



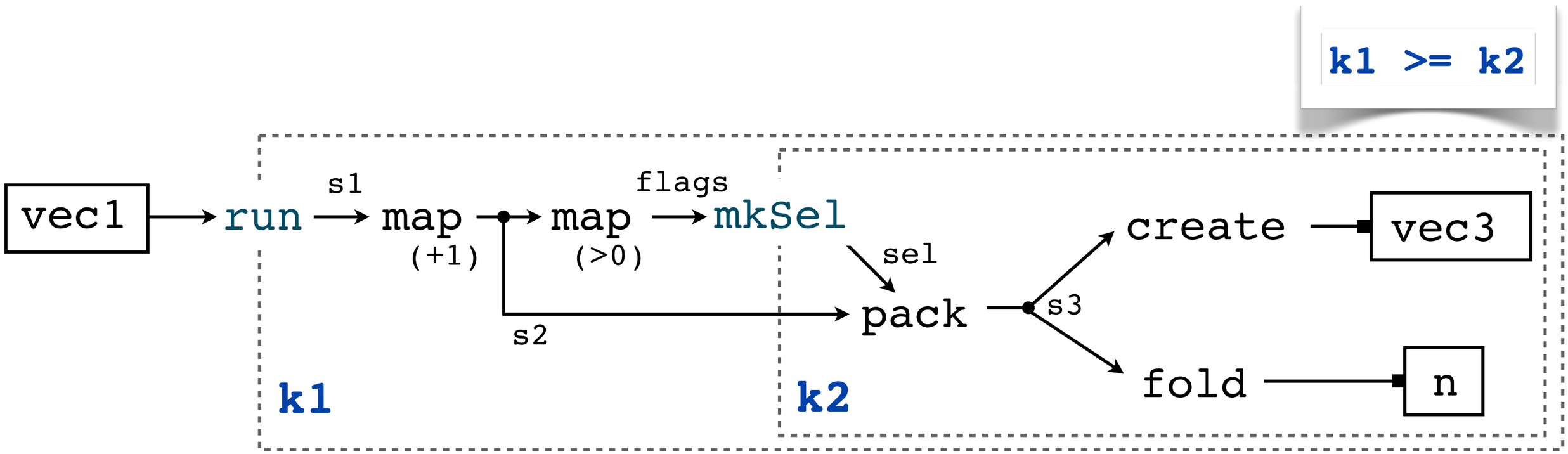
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1                 s2 :: Series k1 Int
        flags = map (> 0) s2              flags :: Series k1 Bool
    in  mkSel flags (\sel ->                sel :: Sel k1 k2
         let s3   = pack sel s2              s3 :: Series k2 Int
             vec3 = create s3
             n    = fold max 0 s3
         in (vec3, n)))
```



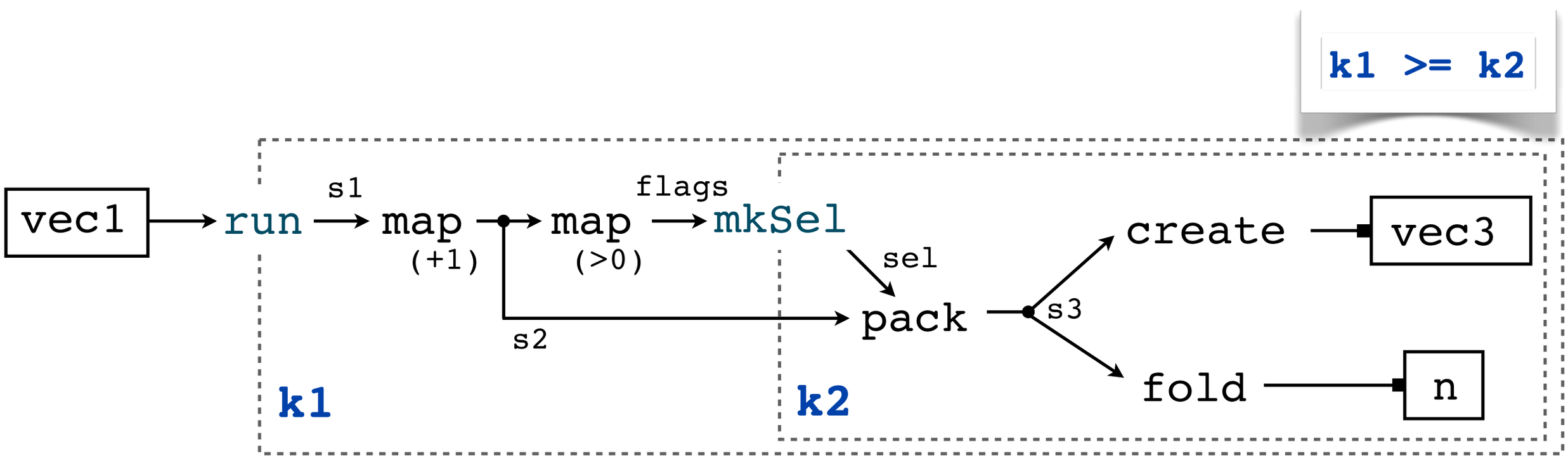
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1                 s2 :: Series k1 Int
        flags = map (> 0) s2              flags :: Series k1 Bool
    in  mkSel flags (\sel ->                sel :: Sel k1 k2
         let s3   = pack sel s2             s3 :: Series k2 Int
             vec3 = create s3             vec3 :: Vector Int
             n    = fold max 0 s3            n :: Int
         in (vec3, n)))
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                          s1 :: Series k1 Int
    let s2    = map (+ 1) s1                 s2 :: Series k1 Int
        flags = map (> 0) s2              flags :: Series k1 Bool
    in  mkSel flags (\sel ->                sel :: Sel k1 k2
         let s3   = pack sel s2             s3 :: Series k2 Int
             vec3 = create s3             vec3 :: Vector Int
             n    = fold max 0 s3            n :: Int
    in (vec3, n)))
```



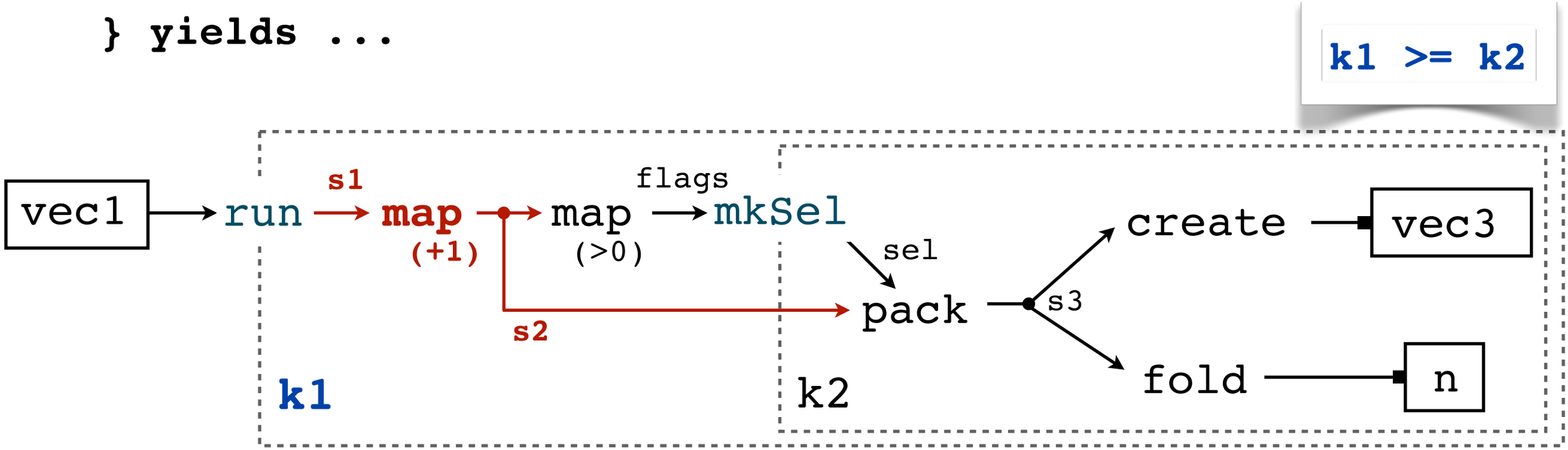
```
run    :: Vector a
       -> (forall k1. Series k1 a -> b) -> b

mkSel :: Series k1 Bool
       -> (forall k2. Sel k1 k2    -> b) -> b
```



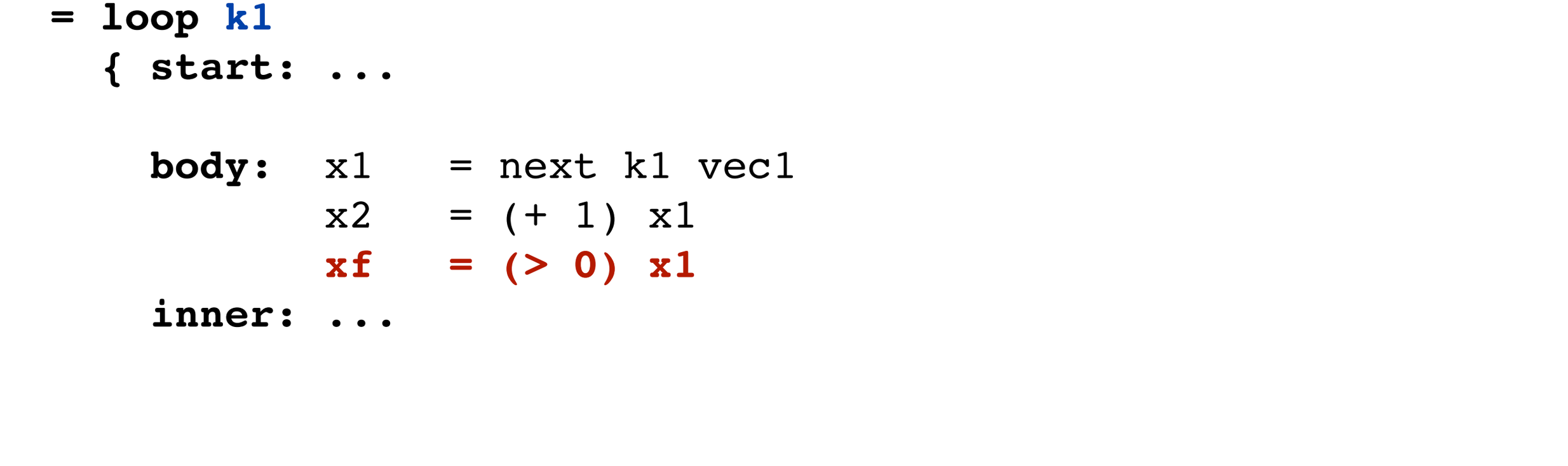
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: ...

      body:  ...


      inner: ...

      end:   ...

    } yields ...
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: ...

      body:  x1    = next k1 vec1


      inner: ...


      end:   ...

    } yields ...
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: ...

      body:  x1    = next k1 vec1
             x2    = (+ 1) x1

      inner: ...

      end:   ...

    } yields ...
```

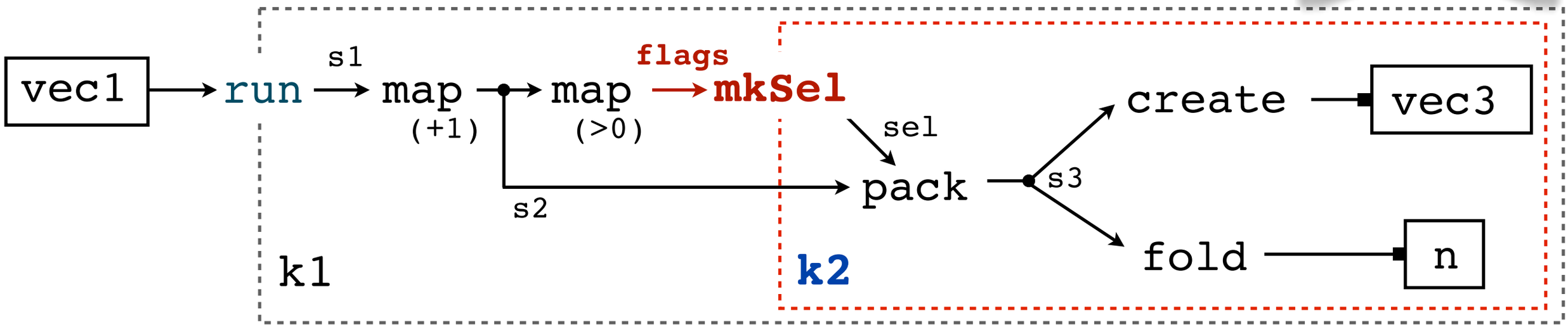


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: ...

      body:  x1    = next k1 vec1
             x2    = (+ 1) x1
             xf    = (> 0) x1
      inner: ...

      end:   ...

    } yields ...
```

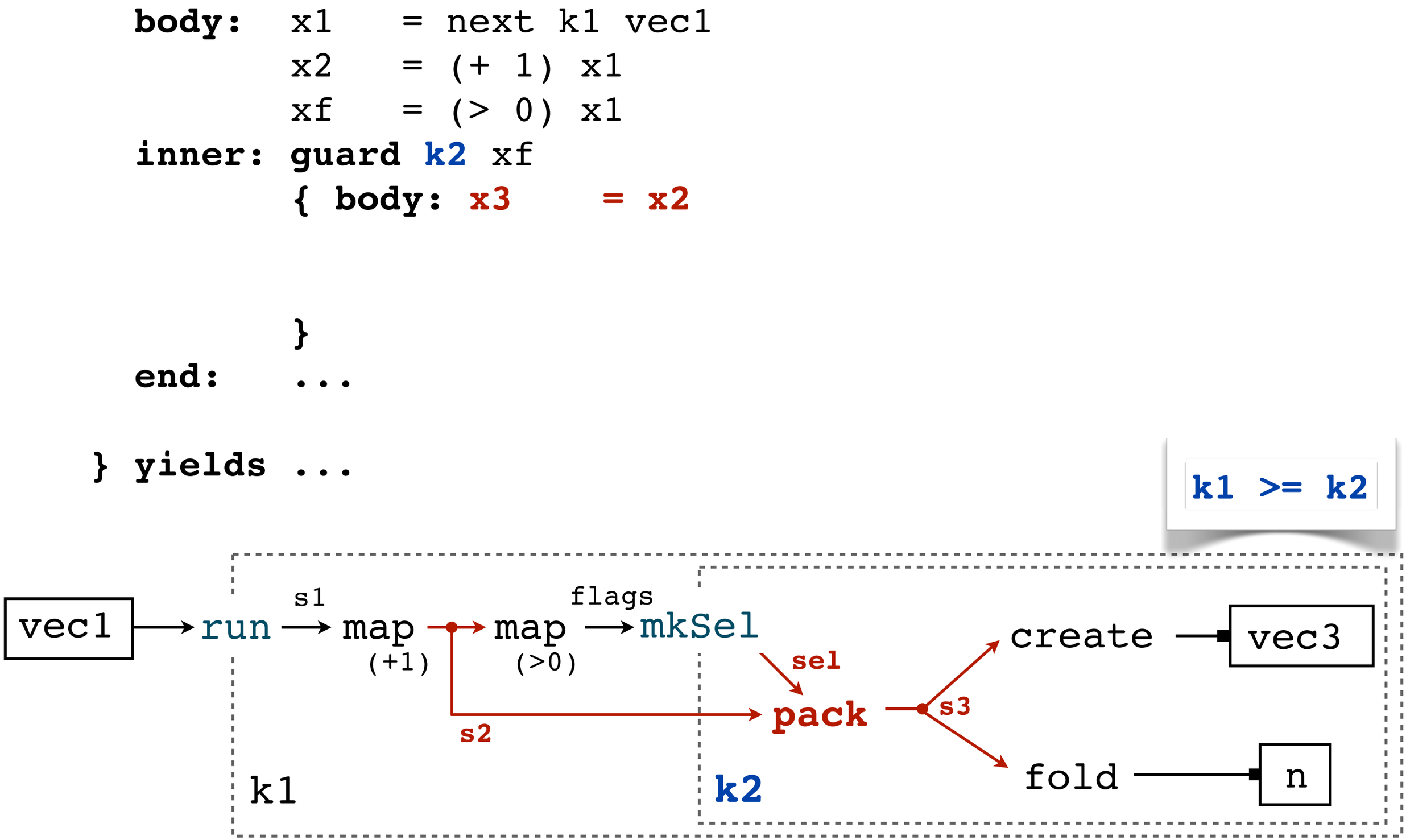


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: ...

      body:  x1    = next k1 vec1
             x2    = (+ 1) x1
             xf    = (> 0) x1
      inner: guard k2 xf
             { body: ...
             }
      end:   ...

    } yields ...
```

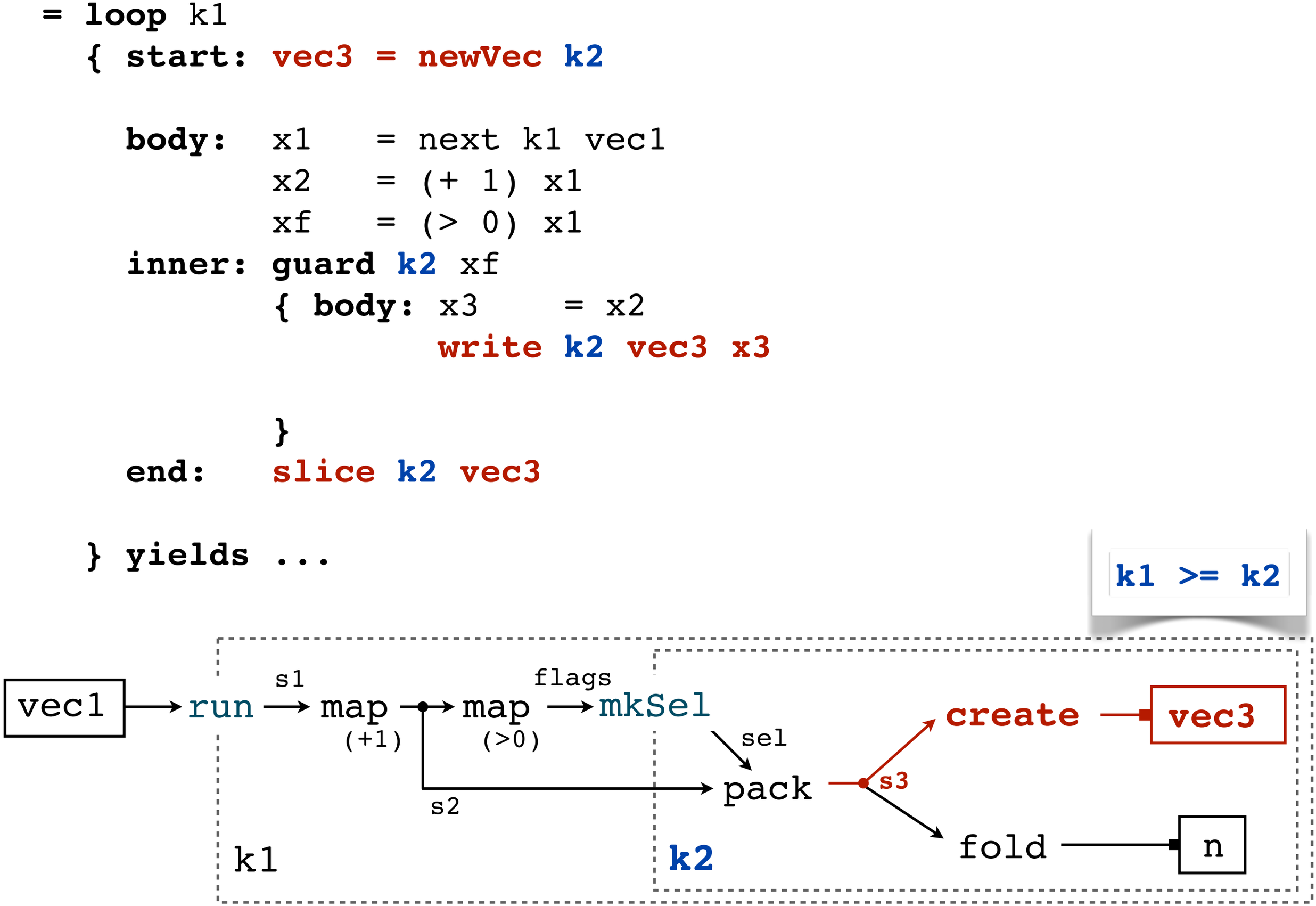
k1 >= k2



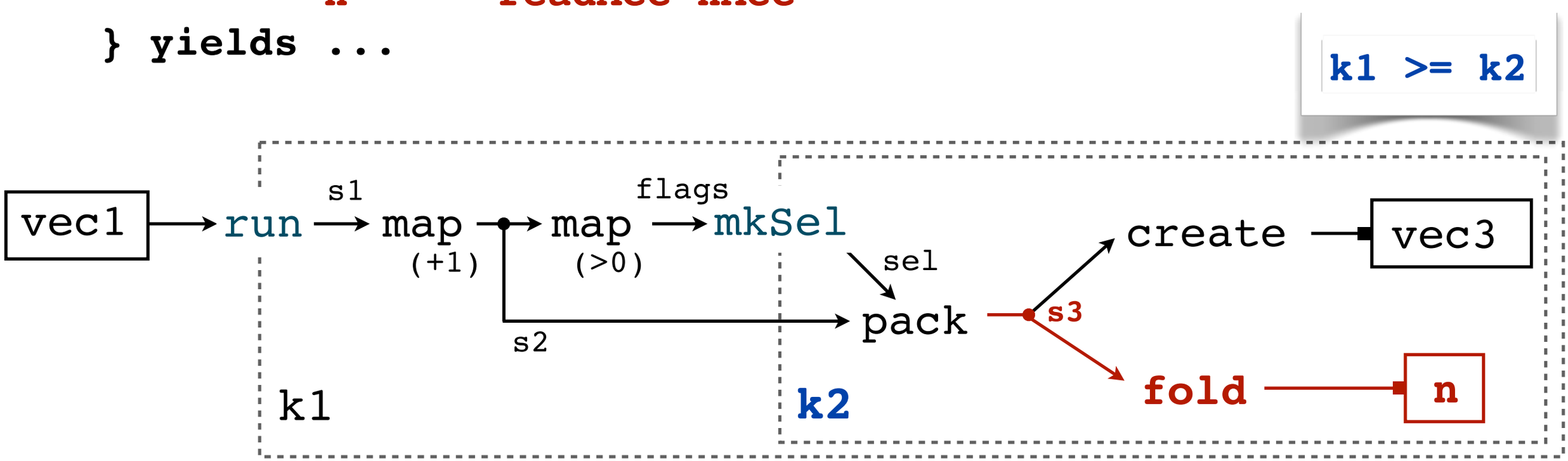
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: ...

      body:  x1    = next k1 vec1
             x2    = (+ 1) x1
             xf    = (> 0) x1
      inner: guard k2 xf
             { body: x3     = x2
             }
      end:   ...

    } yields ...
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: vec3 = newVec k2

      body:  x1    = next k1 vec1
             x2    = (+ 1) x1
             xf    = (> 0) x1
      inner: guard k2 xf
             { body: x3     = x2
                     write k2 vec3 x3
             }
      end:   slice k2 vec3

    } yields ...
```

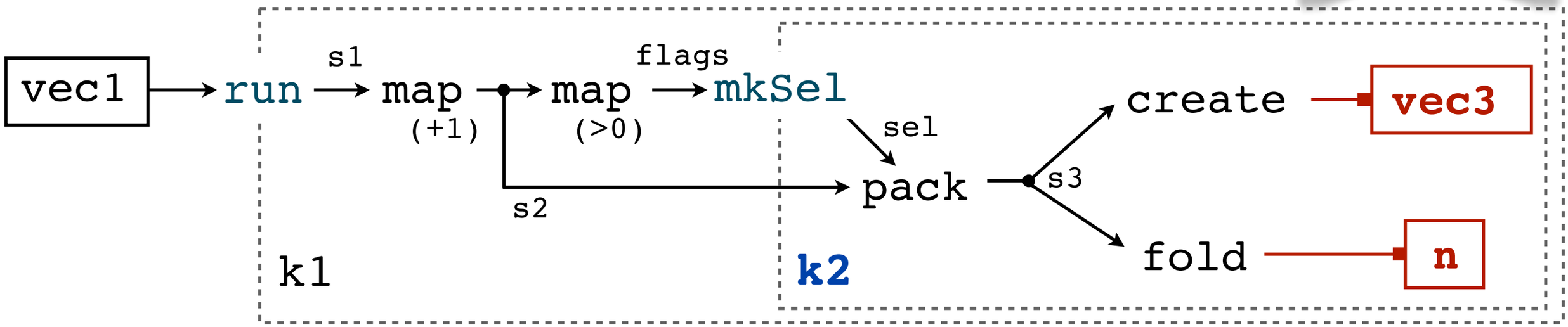


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: vec3 = newVec k2
             nAcc = newAcc 0
      body:  x1   = next k1 vec1
             x2   = (+ 1) x1
             xf   = (> 0) x1
      inner: guard k2 xf
             { body: x3    = x2
                     write k2 vec3 x3
                     nAcc := (+) nAcc x3
             }
      end:   slice k2 vec3
             n    = readAcc nAcc
    } yields ...
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: vec3 = newVec k2
             nAcc = newAcc 0
      body:  x1   = next k1 vec1
             x2   = (+ 1) x1
             xf   = (> 0) x1
      inner: guard k2 xf
             { body: x3    = x2
                     write k2 vec3 x3
                     nAcc := (+) nAcc x3
             }
      end:   slice k2 vec3
             n    = readAcc nAcc
    } yields (vec3, n)
```

k1 >= k2



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2
            nAcc = newAcc 0
     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3
                    nAcc := (+) nAcc x3
            }
   end:    slice k2 vec3
           n    = readAcc nAcc
} yields (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: vec3 = newVec k2
             nAcc = newAcc 0
      body:  x1   = next k1 vec1
             x2   = (+ 1) x1
             xf   = (> 0) x1
      inner: guard k2 xf
             { body: x3    = x2
                     write k2 vec3 x3
                     nAcc := (+) nAcc x3
             }
    end:    slice k2 vec3
            n    = readAcc nAcc
} yields (vec3, n)
```

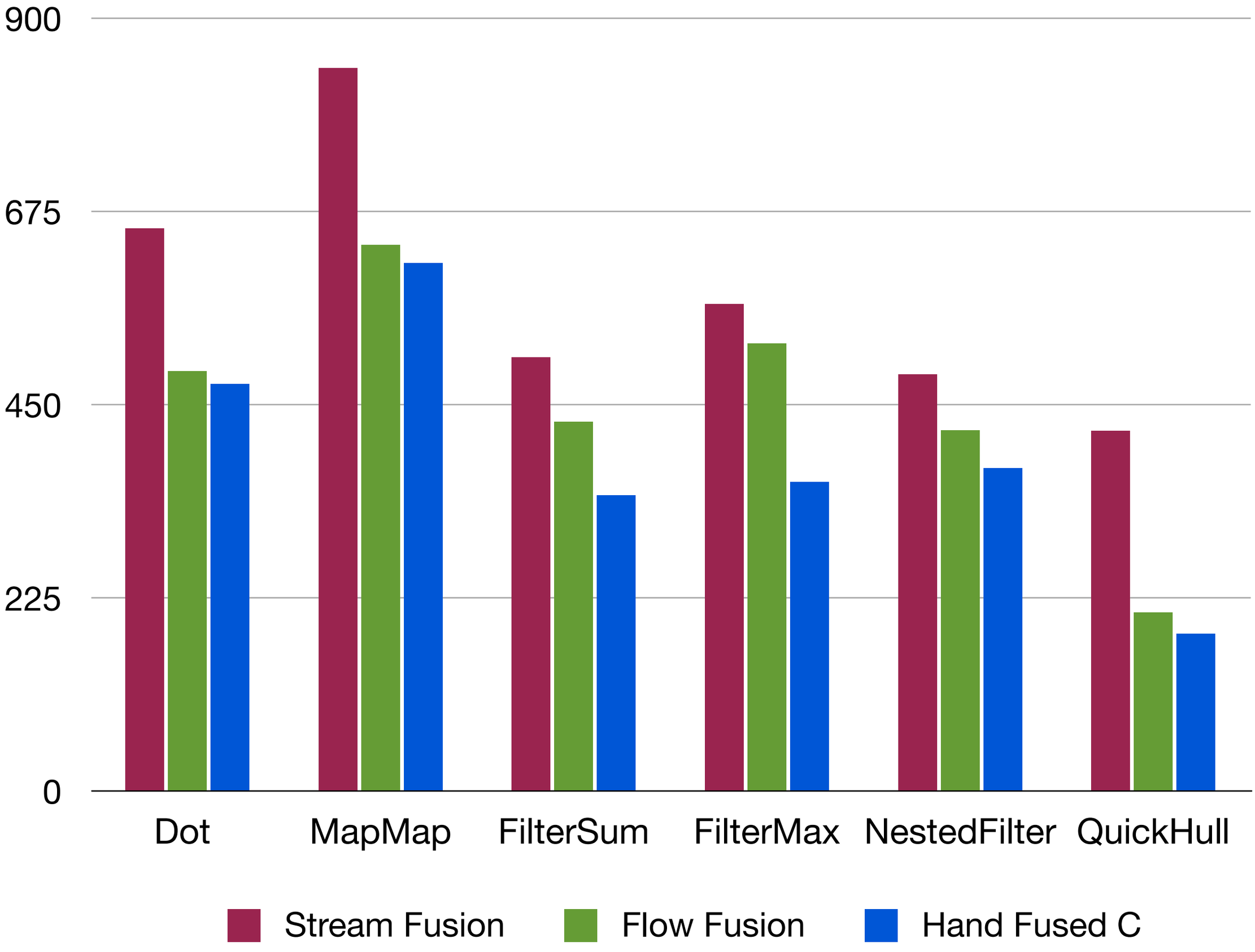
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
    { start: vec3 = newVec k2
             nAcc = newAcc 0
      body:  x1   = next k1 vec1
             x2   = (+ 1) x1
             xf   = (> 0) x1
      inner: guard k2 xf
             { body: x3    = x2
                     write k2 vec3 x3
                     nAcc := (+) nAcc x3
             }
      end:   slice k2 vec3
             n    = readAcc nAcc
    } yields (vec3, n)
```

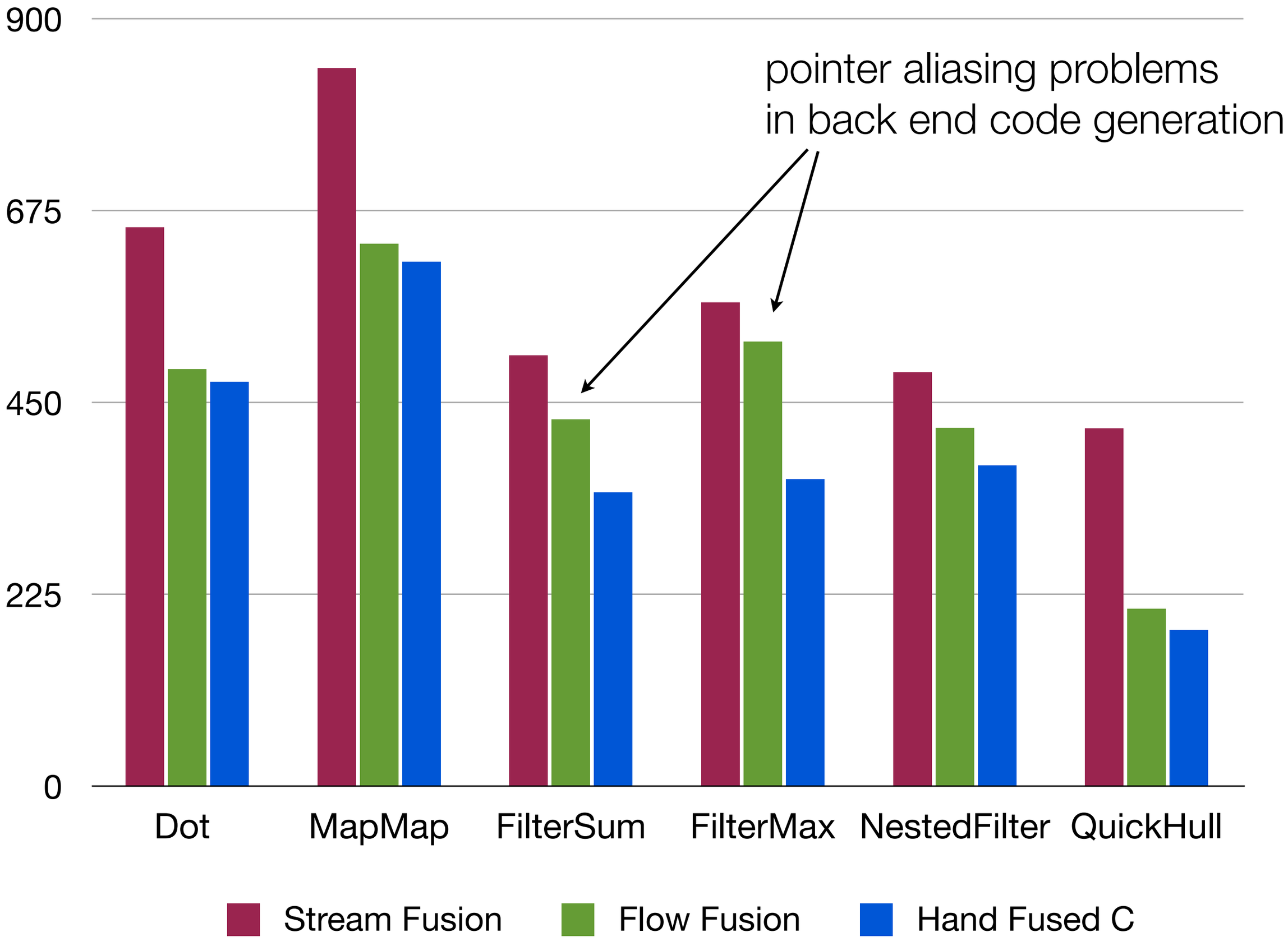
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec (length vec1)
      nAcc  <- newAcc 0
      k2Acc <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do e1      <- next ix1 vec1
            let e2 = (+ 1) e1
            let ef = (> 0) e1
            guardM k2Acc ef (\ix2 ->
             do let s3 = s2
                write ix2 vec3
                modifyAcc nAcc (\x -> (+) x s3))
sliceVec k2Acc vec3
n       <- readAcc nAcc
return (vec3, n)
```

Implementation

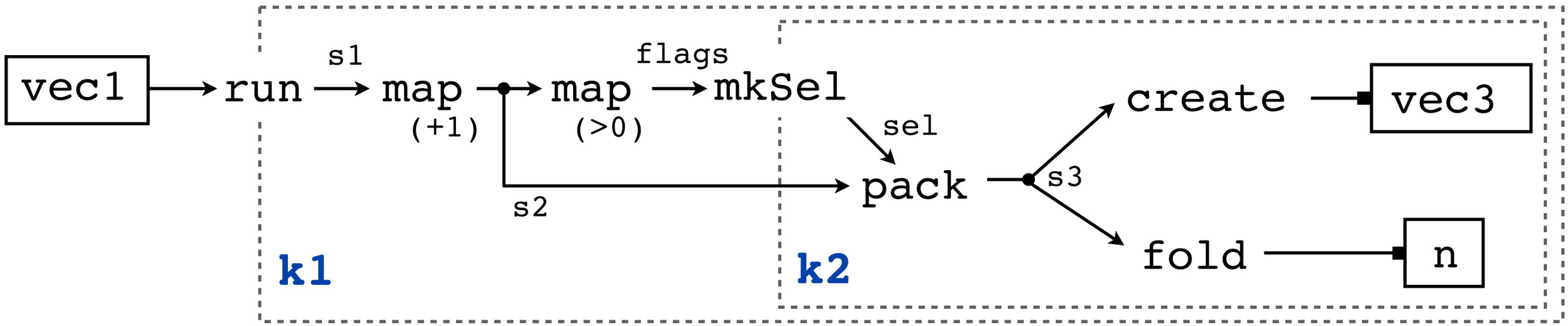
- We use a GHC plugin to hijack Core code during compilation.
- It's a data flow compiler inside a Haskell compiler.
- The back-end combinators like `loopM` and `guardM` are implemented in a separate library.

Benchmarks

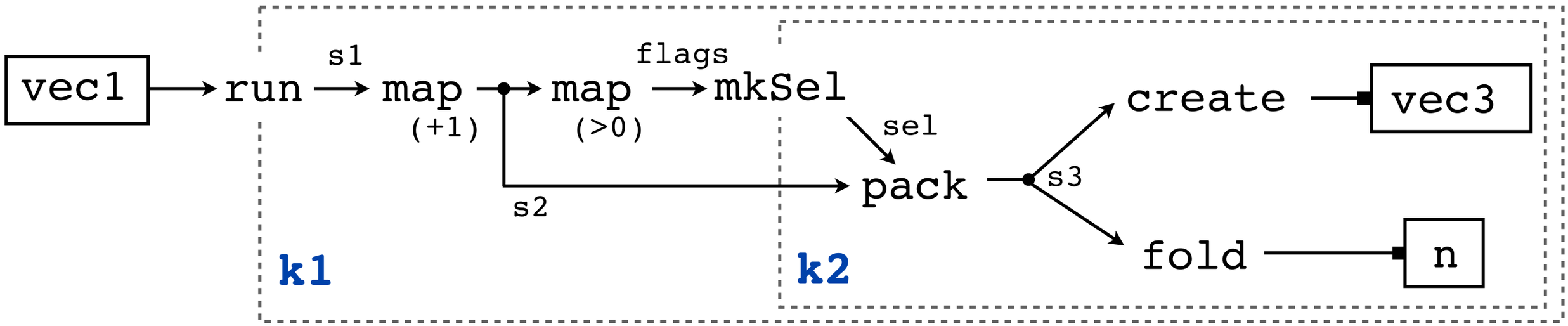

900
675
450
225
0
Dot
MapMap
FilterSum
FilterMax
NestedFilter
QuickHull
Stream Fusion
Flow Fusion
Hand Fused C


900
675
450
225
0
pointer aliasing problems
in back end code generation
Dot
MapMap
FilterSum
FilterMax
NestedFilter
QuickHull
Stream Fusion
Flow Fusion
Hand Fused C

Summary

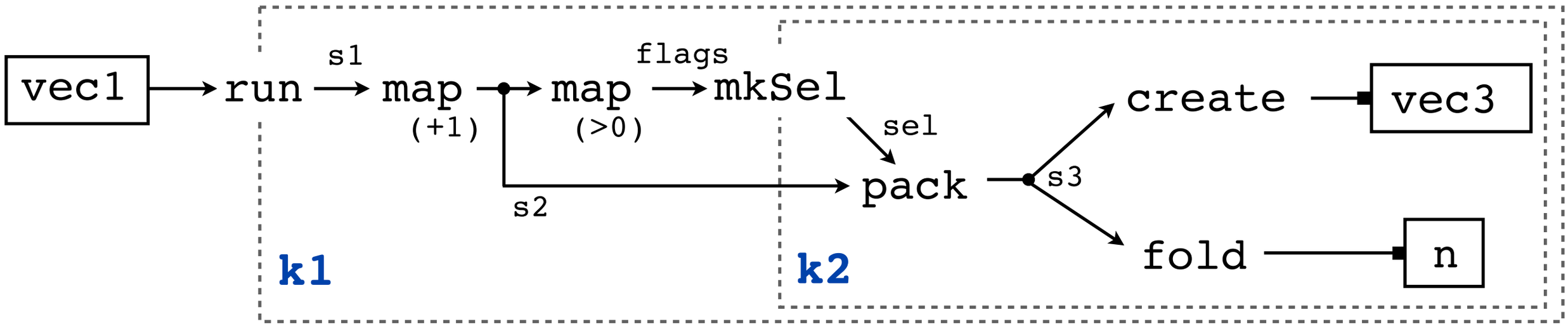


- **IF** your program is a first order, non-recursive, synchronous, finite, data flow program using our combinators.



- **IF** your program is a first order, non-recursive, synchronous, finite, data flow program using our combinators.

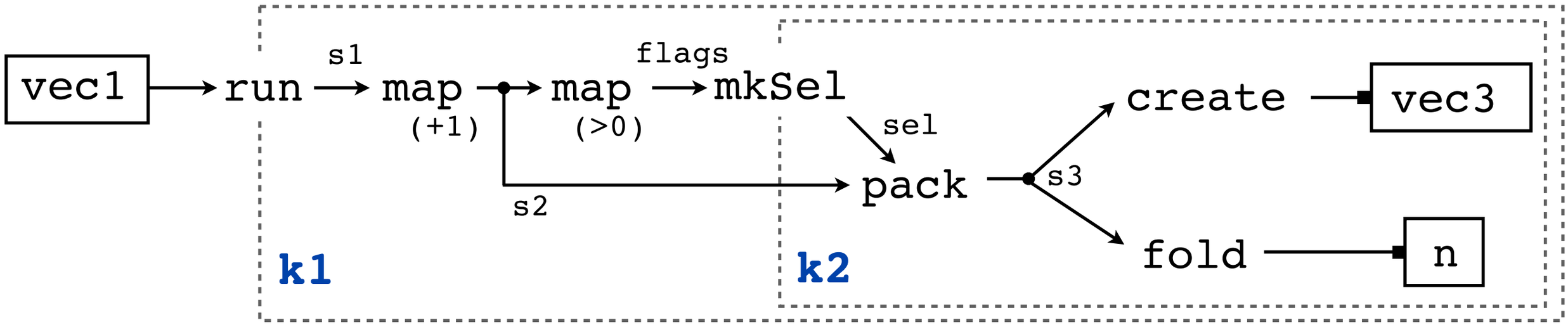
- **THEN** it will be perfectly fused.



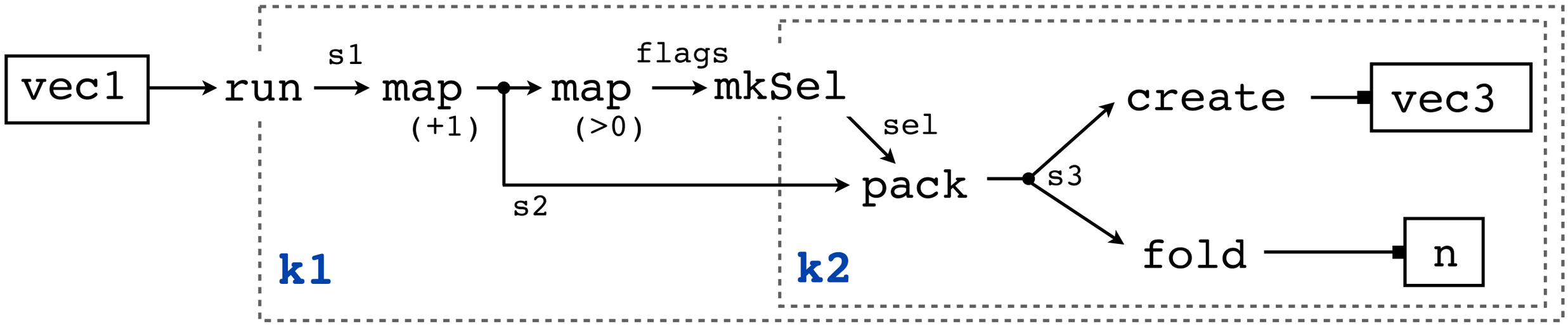
- **IF** your program is a first order, non-recursive, synchronous, finite, data flow program using our combinators.

- **THEN** it will be perfectly fused.

- If you program is NOT one of those then the plugin complains.



- In numerical code, a large number of cycles are spent evaluating data flows of this form.



- In numerical code, a large number of cycles are spent evaluating data flows of this form.

- No need to *stare at Core code* to guarantee success (at least for client programmers).

Questions?

Future Work

- It is fairly easy to generate SIMD code for map/fold programs based on this approach. We already have this working.

- Generating SIMD code for segmented operations will be more annoying, but no conceptual issues.

- Handling filters in an efficient way should be possible with the Intel AVX-512 instruction set (with predicated instructions).

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec k2
      nAcc  <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do e1    <- next ix1 vec1
            let e2 = (+ 1) e1
            let ef = (> 0) e1
         guardM k2 ef (\ix2 ->
          do let s3 = s2
             write ix2 vec3 x3
             modifyAcc nAcc (\x -> (+) x s3))
sliceVec k2 vec3
n     <- readAcc nAcc
return (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec k2
      nAcc  <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do e1    <- next ix1 vec1
            let e2 = (+ 1) e1
            let ef = (> 0) e1
            guardM k2 ef (\ix2 ->
             do let s3 = s2
                write ix2 vec3 x3
                modifyAcc nAcc (\x -> (+) x s3))
sliceVec k2 vec3
n     <- readAcc nAcc
return (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec k2
      nAcc  <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do e1    <- next ix1 vec1
            let e2 = (+ 1) e1
            let ef = (> 0) e1
            guardM k2 ef (\ix2 ->
             do let s3 = s2
                write ix2 vec3 x3
                modifyAcc nAcc (\x -> (+) x s3))
sliceVec k2 vec3
n     <- readAcc nAcc
return (vec3, n)
```

k1 >= k2

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec (length vec1)
      nAcc  <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do e1    <- next ix1 vec1
            let e2 = (+ 1) e1
            let ef = (> 0) e1
            guardM k2 ef (\ix2 ->
             do let s3 = s2
                write ix2 vec3 x3
                modifyAcc nAcc (\x -> (+) x s3))
sliceVec k2 vec3
n     <- readAcc nAcc
return (vec3, n)
```

k1 >= k2

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec (length vec1)
      nAcc  <- newAcc 0
      k2Acc <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do e1      <- next vec1
            let e2 = (+ 1) e1
            let ef = (> 0) e1
            guardM k2Acc ef (\ix2 ->
             do let s3 = s2
                write ix2 vec3
                modifyAcc nAcc (\x -> (+) x s3))
sliceVec k2Acc vec3
n       <- readAcc nAcc
return (vec3, n)
```

```
k1 >= k2
```