15-150 Fall 2013

Lecture 10

Stephen Brookes

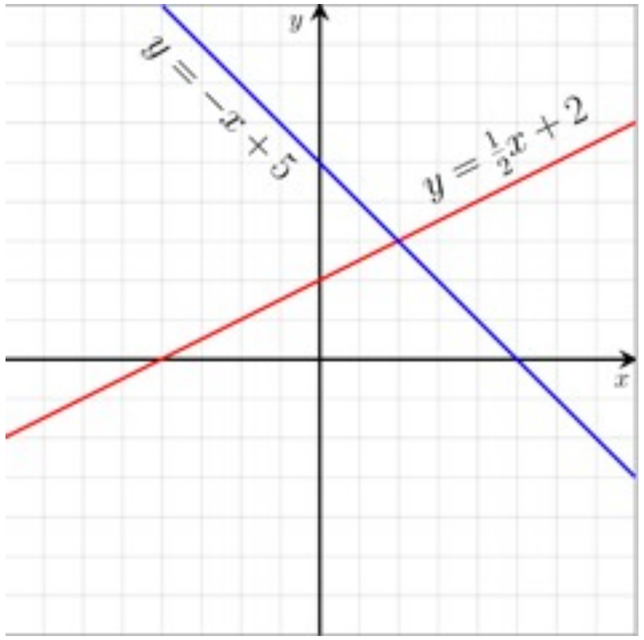
- Functions as values
- Higher-order functions
- The power of polymorphism

Problem

For real numbers a < b,
there is a ***linear function*** f : real -> real
such that

f(a) = ~1.0
f(b) = 1.0

Given a and b, find f.



linear means there are reals α, β
such that for all x:real, f x = α*x + β

Solution

A function

norm : real * real -> (real -> real)

such that for all a, b : real with a < b,

norm(a, b) =>* a linear function f satisfying

f(a) = ~1.0 and f(b) = 1.0

Solution

A function

norm : real * real -> (real -> real)

such that for all a, b : real with a < b,

norm(a, b) =>* a linear function f satisfying

f(a) = ~1.0 and f(b) = 1.0

calculate α, β such that

α*a + β = ~1.0 and α*b + β =1.0

norm

```
fun norm(a, b) = fn x => (2.0 * x - a - b) / (b - a)
```

norm

```
fun norm(a, b) = fn x => (2.0 * x - a - b) / (b - a)
```

```
- val norm = fn : real * real -> real -> real
```

norm

fun norm(a, b) = **fn** x => (2.0 * x - a - b) / (b - a)

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm

```
fun norm(a, b) = fn x => (2.0 * x - a - b) / (b - a)
```

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

norm

```
fun norm(a, b) = fn x => (2.0 * x - a - b) / (b - a)
```

```
- val norm = fn : real * real -> real -> real
```

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

The *type* of norm(~2.0, 2.0) is

norm

fun norm(a, b) = **fn** x => (2.0 * x - a - b) / (b - a)

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

The *type* of norm(~2.0, 2.0) is real -> real

norm

fun norm(a, b) = **fn** x => (2.0 * x - a - b) / (b - a)

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

The *type* of norm(~2.0, 2.0) is real -> real

The *value* of norm(~2.0, 2.0) is

norm

```
fun norm(a, b) = fn x => (2.0 * x - a - b) / (b - a)
```

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

The *type* of norm(~2.0, 2.0) is real -> real

The *value* of norm(~2.0, 2.0) is

```
fn x => (2.0 * x - (~2.0) - 2.0) / (2.0 - (~2.0))
```

norm

fun norm(a, b) = **fn** x => (2.0 * x - a - b) / (b - a)

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

The *type* of norm(~2.0, 2.0) is real -> real

The *value* of norm(~2.0, 2.0) is

fn x => (2.0 * x - (~2.0) - 2.0) / (2.0 - (~2.0))

This value is *equal* to

norm

fun norm(a, b) = **fn** x => (2.0 * x - a - b) / (b - a)

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

norm is a function
that *returns* a function

The *type* of norm(~2.0, 2.0) is real -> real

The *value* of norm(~2.0, 2.0) is

fn x => (2.0 * x - (~2.0) - 2.0) / (2.0 - (~2.0))

This value is *equal* to **fn** x => x / 2.0

norm

- Let $a < b$
- For all x such that $a \leq x \leq b$,

$$\sim 1.0 \leq \text{norm}(a, b)\ (x) \leq 1.0$$

and

$$\text{norm}(a, b)\ a = \sim 1.0$$
$$\text{norm}(a, b)\ b = 1.0$$

norm(a, b) ***normalizes*** the real interval [a...b]

using norm

- Given a pair of reals,
  normalize both components

using norm(~2.0, 2.0),
convert (1.0,1.5) to (0.5, 0.75)

- Given a list of reals,
  normalize each item in the list

using norm(~2.0, 2.0),
convert [1.0,1.5,1.8] to [0.5, 0.75, 0.9]

first attempts

```
fun normpair(a, b) =
      fn (x,y) => (norm(a,b) x, norm(a,b) y)
```

first attempts

```
fun normpair(a, b) =
      fn (x,y) => (norm(a,b) x, norm(a,b) y)
```

```
fun normpair(a, b) =
  fn (x, y) => let
                 val f = norm(a,b)
               in
                 (f x, f y)
               end
```

critique

- We often need to apply a function to components of a data structure
  - *to build a new data structure of the same shape*
- It's the *same idea*, regardless of the function

```
fun normpair(a, b) = fn (x,y) => (norm(a,b) x, norm(a,b) y)
fun fibpair(a, b) = (fib a, fib b)
fun factpair(x, y) = (fact x, fact y)
```

general problem

- For pairs, we want a *generic* way to apply a given function to components
- For lists, we want a *generic* way to apply a given function to list items

"transforming" a data structure
by applying a function to each datum

solutions

- For pairs, a ***polymorphic*** function

  pair : ('a -> 'b) -> 'a * 'a -> 'b * 'b

- For lists, a ***polymorphic*** function

  map : ('a -> 'b) -> 'a list -> 'b list

these are also
higher-order
functions

pair spec

pair : ('a -> 'b) -> 'a * 'a -> 'b * 'b

(* REQUIRES true *)
(* ENSURES pair f (x, y) = (f x, f y) *)

For all types t_1 and t_2,
all values f : t_1 -> t_2, and all values x, y:t_1,
pair f (x, y) = (f x, f y).

pair

```
fun pair f = fn (x, y) => (f x, f y)
```

pair : ('a -> 'b) -> 'a * 'a -> 'b * 'b

pair(norm (~2.0, 2.0)) : real * real -> real * real

pair (norm (~2.0, 2.0)) (1.5, 1.5) =>* ?

map spec

map : ('a -> 'b) -> ('a list -> 'b list)

(* REQUIRES true *)
(* ENSURES For all $n \geq 0$,
map f $[x_1, ..., x_n]$ = $[f\ x_1, ..., f\ x_n]$ *)

For all $n \geq 0$, all types t_1 and t_2,
all values $f : t_1 \rightarrow t_2$, and all values $x_1, ..., x_n : t_1$,

map f $[x_1, ..., x_n] = [f\ x_1, ..., f\ x_n]$.

map

```
fun map f = fn L =>
              case L of
                 [ ]  => [ ]
               | x::R => (f x) :: (map f R)
```

map : ('a -> 'b) -> ('a list -> 'b list)

map

map f R = (map f) R

```
fun map f = fn L =>
              case L of
                 [ ]  => [ ]
               | x::R => (f x) :: (map f R)
```

map : ('a -> 'b) -> ('a list -> 'b list)

map

```
fun map f = fn L =>
              case L of
                 [ ]  => [ ]
               | x::R => (f x) :: (map f R)
```

map f R = (map f) R

map : ('a -> 'b) -> ('a list -> 'b list)

map (norm(~2.0, 2.0))

map

map f R = (map f) R

fun map f = **fn** L =>
case L **of**
[ ] => [ ]
| x::R => (f x) :: (map f R)

map : (’a -> ’b) -> (’a list -> ’b list)

map (norm(~2.0, 2.0)) : real list -> real list

map

map f R = (map f) R

```
fun map f = fn L =>
              case L of
                [ ]  => [ ]
              | x::R => (f x) :: (map f R)
```

map : ('a -> 'b) -> ('a list -> 'b list)

map (norm(~2.0, 2.0)) : real list -> real list

map (norm(~2.0, 2.0)) [1.0, 1.5. 2.0] =>* [0.5, 0.75, 1.0]

syntactic sugar

- ML has a *streamlined* syntax for defining higher-order functions

```
fun pair f = fn (x, y) => (f x, f y)

fun pair f (x,y) = (f x, f y)
```

```
fun map f = fn L => case L of
                  [ ]  => [ ]
                | x::R => (f x) :: (map f R)

fun map f [ ]   = [ ]
  |   map f (x::R) = (f x) :: (map f R)
```

syntactic sugar

$$
\begin{array}{llll}
\textbf{fun} & F\ p_{11}\ p_{12} & = & e_1 \\
\mid & F\ p_{21}\ p_{22} & = & e_2 \\
\ldots & & & \\
\mid & F\ p_{k1}\ p_{k2} & = & e_k
\end{array}
$$

declares $F : t_1 \text{ -> } t_2 \text{ -> } t$

if for each i, p_{i1} fits t_1, p_{i2} fits t_2, and they produce type bindings for which, assuming $F : t_1 \text{ -> } t_2 \text{ -> } t$, e_i has type t

so far

- Using pair and map we can easily apply a function to the data in any data structure built from pairs and lists

lists of pairs

pairs of lists

pairs of pairs

lists of lists of pairs

using map

to build a recursively definable list

(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)

using map

to build a recursively definable list

```
(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)

     fun sublists [ ] = [ [ ] ]
```

using map

to build a recursively definable list

```
(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)
```

fun sublists [ ] = [[ ]]
  | sublists (x::R) =

using map

to build a recursively definable list

```
(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)
    fun sublists [ ] = [ [ ] ]
     |   sublists (x::R) =
          let
             val S = sublists R
          in
```

using map

to build a recursively definable list

```
(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)

    fun sublists [ ] = [ [ ] ]
     |   sublists (x::R) =
            let
               val S = sublists R
            in
               S @ map (fn A => x::A) S
            end
```

using map

to build a recursively definable list

(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)

```
fun sublists [ ] = [ [ ] ]
 |   sublists (x::R) =
        let
          val S = sublists R
        in
          S @ map (fn A => x::A) S
        end
```

sublists [1,2,3] = [[ ],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]

general problem

- Given a data structure representing a *collection* of data, such as a list or a tree
- We want a *generic* way to ***combine*** the data in the collection, using a binary operation and a base value

combining

- Suppose we have a function

  $$F : t_1 * t_2 \text{ -> } t_2$$

  and want to combine the items of a list

  $$[x_1,...,x_n] : t_1 \text{ list}$$

  with a value $z : t_2$

  to get the value of

  $$F(x_1, F(x_2, ..., F(x_n, z)...))$$

examples

- Add the numbers in a **real** list
- Multiply the numbers in an **int** list
- Find the smallest integer in an **int** list
- Find the largest real in a **real** list

It's the same idea in each case:

combining a list of data

using a specific operation and base value

solution

- A ***polymorphic*** function

  foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

  such that

  for all types t_1, t_2,
  all $n \geq 0$, and all values
  F: t_1 * t_2 -> t_2, $[x_1,\ldots,x_n]$: t_1 list, z : t_2 ,

  $$\text{foldr F z } [x_1,\ldots,x_n] = F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))$$

foldr

```
fun foldr F z [ ] = z
  |   foldr F z (x::L) = F(x, foldr F z L)
```

foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

foldr F z $[x_1,\ldots,x_n]$ = $F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))$

sum : int list -> int

(* ENSURES sum L = the sum of the items in L *)

sum : int list -> int

(* ENSURES sum L = the sum of the items in L *)

```
fun sum L = foldr (op +) 0 L
```

sum : int list -> int

(* ENSURES sum L = the sum of the items in L *)

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

sum : int list -> int

(* ENSURES sum L = the sum of the items in L *)

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

foldr (op +) 0 $[x_1,\ldots,x_n] = x_1 + (x_2 + \ldots (x_n + 0)\ldots)$

sum : int list -> int

(* ENSURES sum L = the sum of the items in L *)

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

$$\text{foldr (op +) 0 } [x_1,\ldots,x_n] = x_1 + (x_2 + \ldots (x_n + 0)\ldots)$$
$$= x_1 + x_2 + \ldots + x_n$$

prod : real list -> real

```
fun prod L = foldr (op * ) 1.0 L
val prod = foldr (op * ) 1.0
```

$$\text{foldr (op *) 1.0 } [x_1,\ldots,x_n] = x_1 * (x_2 * \ldots (x_n * 1.0)\ldots)$$

$$= x_1 * x_2 * \ldots * x_n$$

prod : real list -> real

fun prod L = foldr (op *) 1.0 L

val prod = foldr (op *) 1.0

val prod = foldr (op *) 1.0

foldr (op *) 1.0 $[x_1,...,x_n] = x_1 * (x_2 * ... (x_n * 1.0)...)$

$= x_1 * x_2 * ... * x_n$

maxlist : real list -> real

(* REQUIRES L is a non-empty list *)

(* ENSURES maxlist L = the largest item in L *)

```
fun maxlist (x::R) = foldr Real.max x R
```

Real.max : real * real -> real

maxlist : real list -> real

(* REQUIRES L is a non-empty list *)

(* ENSURES maxlist L = the largest item in L *)

```
fun maxlist (x::R) = foldr Real.max x R
```

Real.max : real * real -> real

Warning: non-exhaustive patterns

foldl

```
fun foldl F z [ ] = z
  |  foldl F z (x::L) = foldl F (F(x, z)) L
```

foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

foldl F z $[x_1,\ldots,x_n]$ = $F(x_n, F(x_{n-1},\ldots, F(x_1,z)\ldots))$

≠

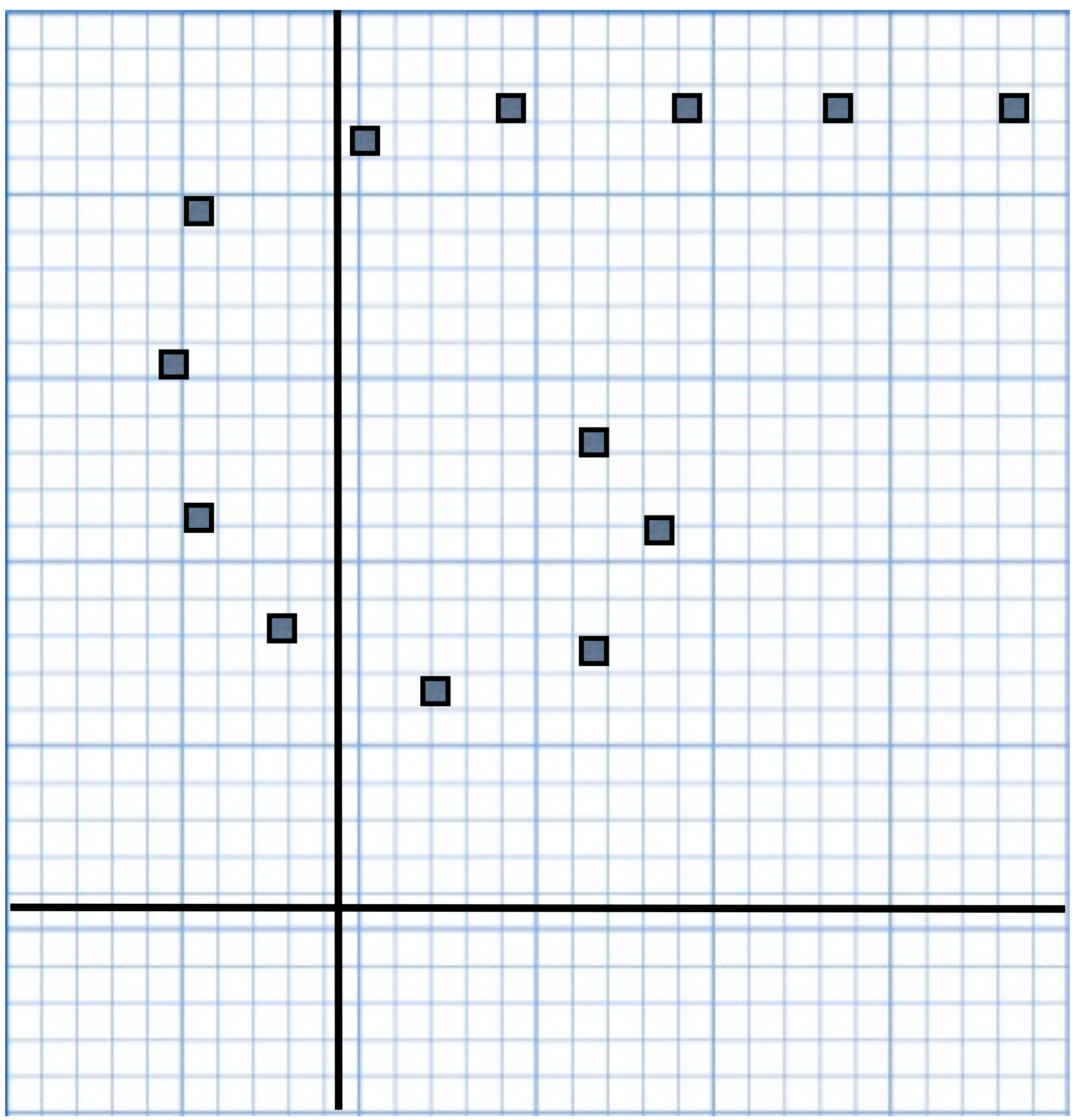
foldr (op @) [ ] [[1,2], [ ], [3,4]] = [1,2,3,4]

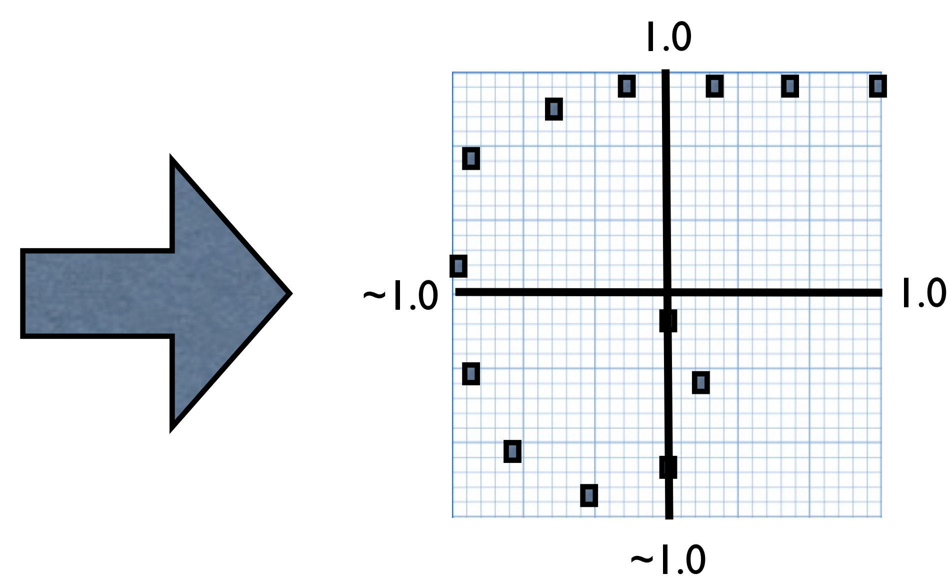
foldl (op @) [ ] [[1,2], [ ], [3,4]] = [3,4,1,2]

In general,
when is
foldr f = foldl f ?

in class problem

- Given a non-empty list of *points*
- *Normalize* to [~1.0 ... 1.0] X [~1.0 ... 1.0]





solution

(using functions from today)

```
(* normalize : (real * real) list -> (real * real) list          *)
(* REQUIRES L is non-empty                                         *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0]  *)
```

solution

(using functions from today)

```
(* normalize : (real * real) list -> (real * real) list                *)
(* REQUIRES L is non-empty                                              *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0]  *)

fun normalize (L : (real * real) list) : (real * real) list =
```

solution
(using functions from today)

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
```

solution

(using functions from today)

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  in

  end
```

solution

(using functions from today)

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  in
     map
  end
```

solution

(using functions from today)

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  in
     map  (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y))
  end
```

solution

(using functions from today)

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  in
     map  (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y)) L
  end
```

solution

(using functions from today)

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)
(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  in
     map  (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y)) L
  end
```

Why does this work?

alternative

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, ylo) = pair minlist (xs, ys)
    val (xhi, yhi) = pair maxlist (xs,ys)
  in
     map  (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y)) L
  end
```

alternative

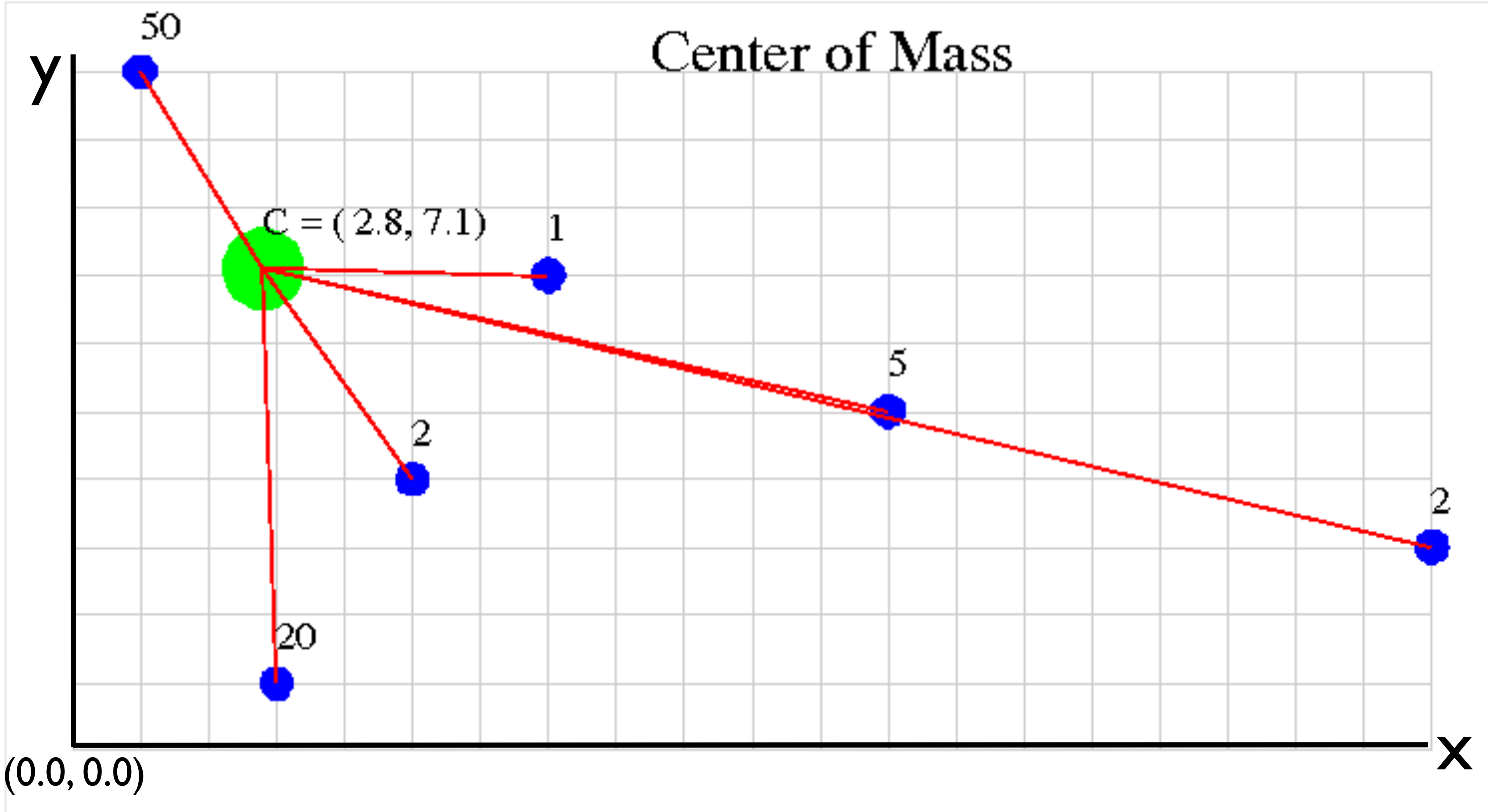
```
fun unzip [ ] = ([ ], [ ])
 |    unzip ((x,y)::R) = let val (xs,ys) = unzip R in (x::xs, y::ys) end

fun normalize (L : (real * real) list) : (real * real) list =
   let
      val zs = unzip L
      val (xlo, ylo) = pair minlist zs
      val (xhi, yhi) = pair maxlist zs
   in
      map  (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y)) L
   end
```

center of mass

- Given a list of *bodies* $[(m_1,(x_1,y_1)),\ldots,(m_n,(x_n,y_n))]$ representing bodies with mass m_i at (x_i,y_i)
- Calculate the *center of mass* (X,Y) : real * real where

$$X = (m_1*x_1 + \ldots + m_n*x_n)/M$$
$$Y = (m_1*y_1 + \ldots + m_n*y_n)/M$$
$$M = m_1 + \ldots + m_n$$

center of mass



caveat

- If all points in L have the same x-value or all points in L have the same y-value, evaluation of normalize L will generate a runtime error.
- Reason: either xlo = xhi or ylo = yhi, so one of the two norm calls will divide by 0.0

solution

```
type point = real * real
type body = real * point
fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)
fun mass (m, (x, y)) = m
fun scale r (m, (x, y))) = (r * m * x, r * m * y)
```

solution

```
type point = real * real
type body = real * point
fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)
fun mass (m, (x, y)) = m
fun scale r (m, (x, y))) = (r * m * x, r * m * y)
fun center (L : body list) : point =
```

solution

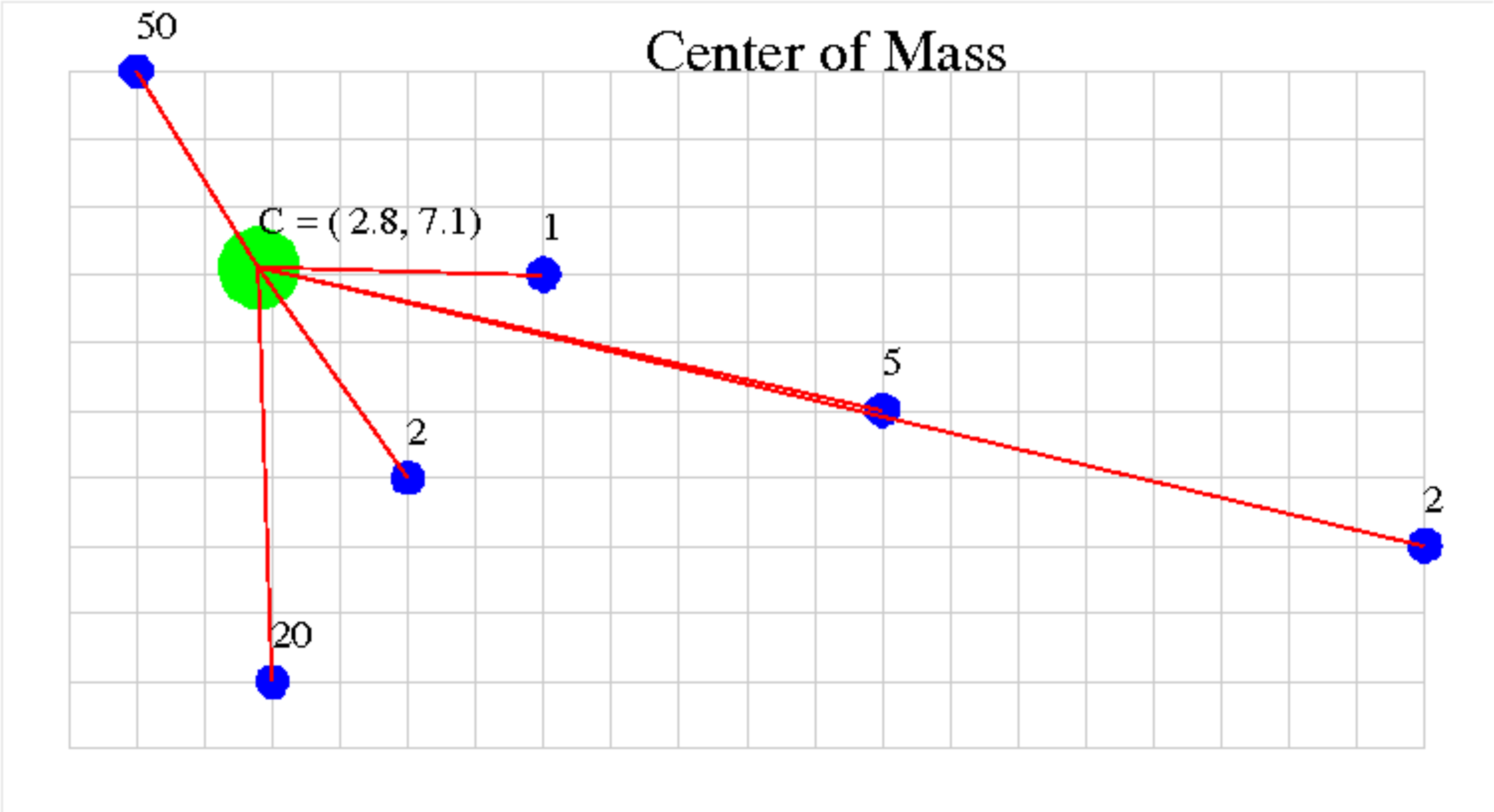
```
type point = real * real
type body = real * point

fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)

fun mass (m, (x, y)) = m

fun scale r (m, (x, y))) = (r * m * x, r * m * y)

fun center (L : body list) : point =
  let
    val M = foldr (op +) 0.0 (map mass L)
  in
```

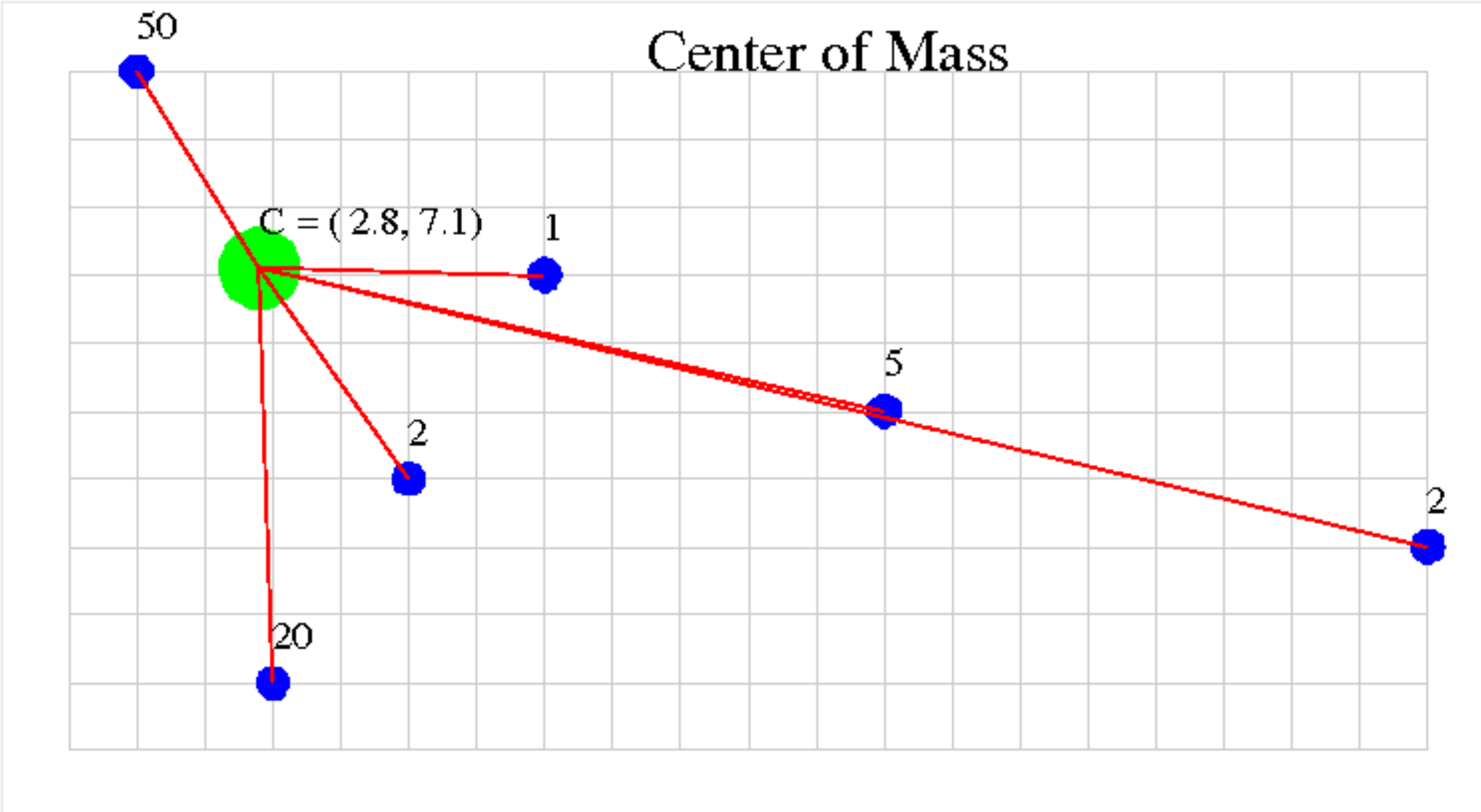
solution

```
type point = real * real
type body = real * point

fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)

fun mass (m, (x, y)) = m

fun scale r (m, (x, y))) = (r * m * x, r * m * y)

fun center (L : body list) : point =
  let
    val M = foldr (op +) 0.0 (map mass L)
  in
    foldr add (0.0, 0.0) (map (scale (1.0/M)) L)
```

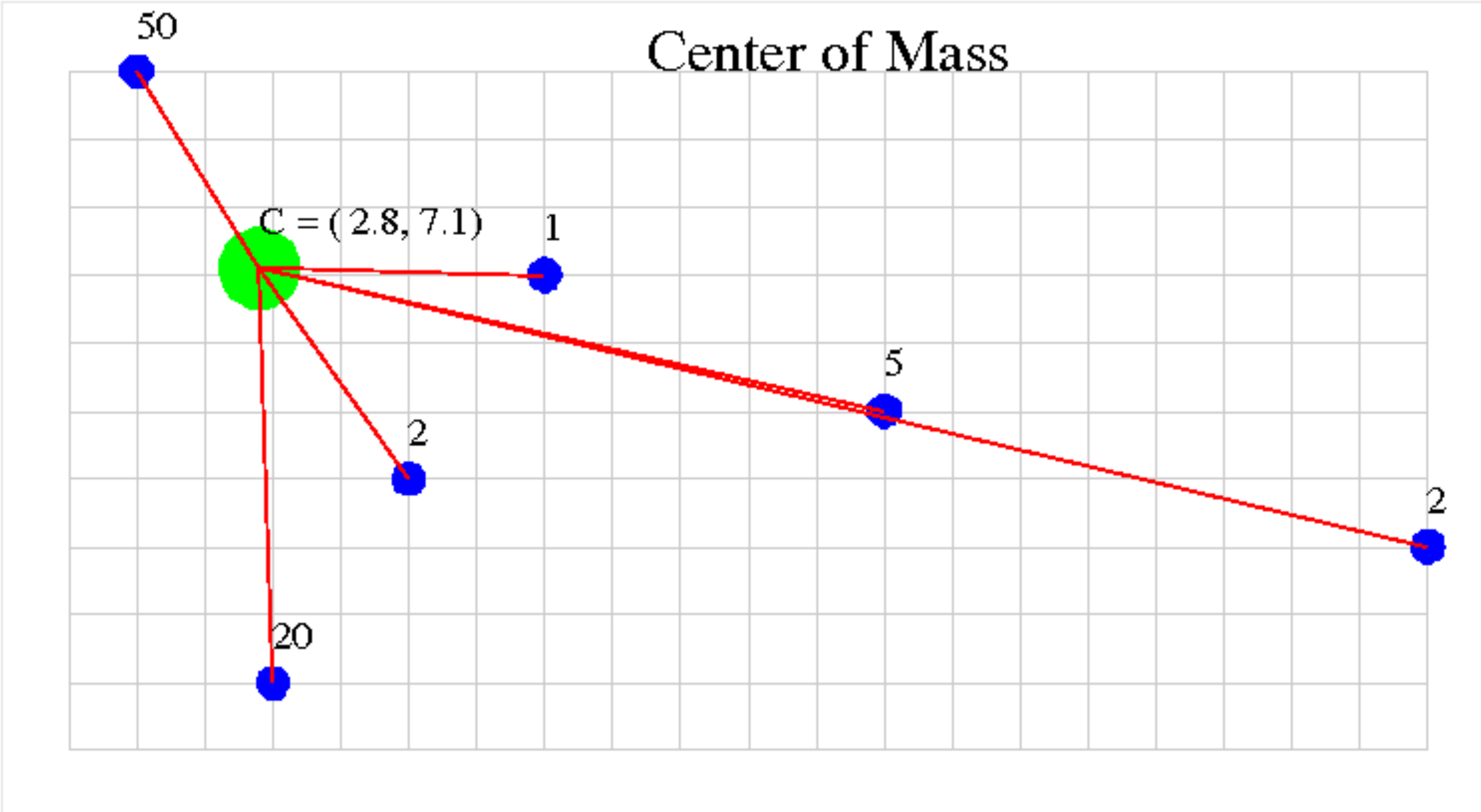
solution

```
type point = real * real
type body = real * point

fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)

fun mass (m, (x, y)) = m

fun scale r (m, (x, y))) = (r * m * x, r * m * y)

fun center (L : body list) : point =
  let
    val M = foldr (op +) 0.0 (map mass L)
  in
    foldr add (0.0, 0.0) (map (scale (1.0/M)) L)
  end
```


Center of Mass
C = (2.8, 7.1)
50
1
5
2
2
20



- center [(50.0,(1.0,10.0)),(20.0,(3.0,1.0)),(2.0,(5.0,4.0)),
  (1.0,(7.0,7.0)),(5.0,(12.0,5.0)),(2.0,(20.0,3.0))];




```
- center [(50.0,(1.0,10.0)),(20.0,(3.0,1.0)),(2.0,(5.0,4.0)),
          (1.0,(7.0,7.0)),(5.0,(12.0,5.0)),(2.0,(20.0,3.0))];
val it = (2.8375,7.075) : real * real
```

more problems

["all ","your "," base "]

"all your base are belong to us"

more problems

["all ","your "," base "]

"all your base are belong to us"

foldr (op ^) "are belong to us"
: string list -> string

more problems

["all ","your "," base "]

"all your base are belong to us"

foldr (op ^) "are belong to us"

: string list -> string

["all ","your ","base "]

["All ","Your ","Base "]

more problems

["all ","your "," base "]

→ "all your base are belong to us"

foldr (op ^) "are belong to us"
: string list -> string

["all ","your ","base "]

→ ["All ","Your ","Base "]

map *capitalize* : string list -> string list

capitalize

explode : string -> char list
implode : char list -> string
Char.toUpper : char -> char

```
- fun capitalize (s:string) : string =
   let
     val (x::L) = explode s
   in
     implode(Char.toUpper x :: L)
   end;

- val capitalize = fn : string -> string
- capitalize "foo";
val it = "Foo" : string
```