Functional Programming With Numbers

The Josephus problem: n people numbered 1 to n are made to stand in a circle. Starting from the person numbered 1, every third live person is killed. This is done till only two persons are left. As an example, if n is 15, then the survivors are the persons who were originally at positions 5 and 14.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Write a function `(canSurvive pos n)` that takes as its arguments a position `pos` and the number of people `n`, and returns `True` if the person at position `pos` is one of the last two survivors otherwise it returns `False`.

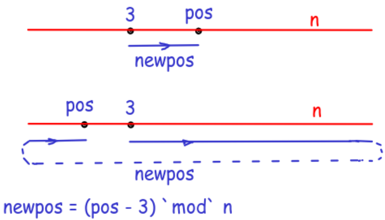
Functional Programming With Numbers

- Kill every third person = Repeatedly kill the third person.
- After every killing, renumber the survivors with the person after the dead person counted as 1.

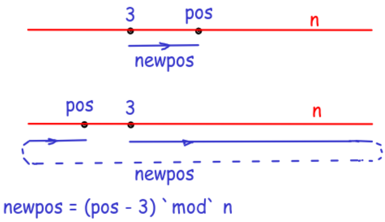
  `canSurvive 8 15` $\Rightarrow$
  `canSurvive 5 14` $\Rightarrow$
  `canSurvive 2 13` $\Rightarrow$
  `canSurvive 12 12` ...

- Stop when `canSurvive` is called with 3 or the number remaining people is less than 2.

Functional programming with numbers



Functional programming with numbers




```
canSurvive pos n | n <= 2 = True
                 | pos == 3 = False
                 | otherwise = canSurvive (mod (pos - 3) n)  (n - 1)
```

Functional programming with numbers

The following iterative sequence is defined for the set of positive integers:

$$
\begin{array}{ll}
n \to n/2, & \text{if } n \text{ is even} \\
n \to 3n+1, & \text{if } n \text{ is odd and} \neq 1 \\
1, & \text{otherwise}
\end{array}
$$

Functional programming with numbers

The following iterative sequence is defined for the set of positive integers:

$$
\begin{array}{ll}
n \to n/2, & \text{if } n \text{ is even} \\
n \to 3n + 1, & \text{if } n \text{ is odd and} \neq 1 \\
1, & \text{otherwise}
\end{array}
$$

Example:

$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Functional programming with numbers

The following iterative sequence is defined for the set of positive integers:

$$\begin{array}{ll} n \to n/2, & \text{if } n \text{ is even} \\ n \to 3n+1, & \text{if } n \text{ is odd and } \neq 1 \\ 1, & \text{otherwise} \end{array}$$

Example:

$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Although nobody has been able to prove it, it is believed that the sequence starting at any number finishes at 1.

Functional programming with numbers

The following iterative sequence is defined for the set of positive integers:

$n \to n/2,$ if n is even
$n \to 3n + 1,$ if n is odd and $\neq 1$
$1,$ otherwise

Example:

$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Although nobody has been able to prove it, it is believed that the sequence starting at any number finishes at 1.

Write a function `longestchain n` which finds the length of the longest chain starting from any number less than or equal to `n`.

Functional programming with numbers

The following iterative sequence is defined for the set of positive integers:

$$\begin{array}{ll} n \rightarrow n/2, & \text{if } n \text{ is even} \\ n \rightarrow 3n+1, & \text{if } n \text{ is odd and} \neq 1 \\ 1, & \text{otherwise} \end{array}$$

Example:

$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

Although nobody has been able to prove it, it is believed that the sequence starting at any number finishes at 1.

Write a function `longestchain n` which finds the length of the longest chain starting from any number less than or equal to `n`.

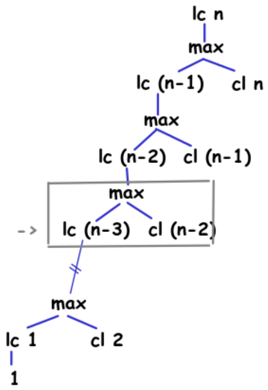
Example: `longestchain 22` is 21.

This corresponds to $18 \rightarrow 9 \rightarrow 28 \rightarrow 14 \rightarrow 7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

Functional programming with numbers

```
longestChain n | n == 1 = 1
               | otherwise = max (longestChain  (n-1)) (chainLength n)
   where chainLength n  |  n == 1 = 1
                        |  even n = 1 + chainLength (n `div` 2)
                        |  odd  n = 1 + chainLength (3 * n + 1)
```

Calculating a tail-recursive form



After computing lc(n-3), it still remains to compute
max (lc (n-3), cl (n-2), cl (n-1), cl n) to compute (lc n).

Suppose we wanted to avoid the computation after the return. A way of doing this is to carry the computation of max (cl (n-2), cl (n-1), cl n) as we go down. At the step shown in the box, we do the computation

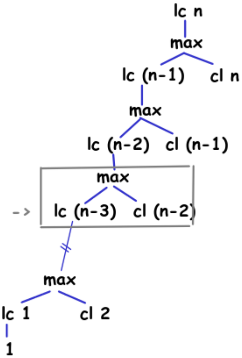
```
max ( lc(n-3), k)
      where k = max ( cl (n-2), cl (n-1), cl n)
```

The boxed computation can be generalized to:

```
lc_tr i k = max (lc i, k) -- specification of lc_tr
```

How does one find a definition of lc_tr that is independent of lc? How does lc_tr get the right k?

Summation of infinite series




```
lc_tr i k = max (lc i, k) -- specification of lc_tr
```

We calculate:

```
lc_tr 1 k = max (lc 1, k)
          = max(1, k)        -- lc 1 = 1

lc_tr i k = max(lc i, k)
          = max(max (lc (i-1), (cl i)), k) -- definition of lc
          = max( lc(i-1), max((cl i), k))  -- assoc. of max
          = lc_tr (i-1)  (max ((cl i), k))  -- specification of
                                               lc_tr
```

Also,

```
lc n = max (lc n, 0) = lc_tr n 0    -- 0 is right identity of max.
```

Initially, lc n passes the right k to lc_tr
Once lc_tr gets the right k, it passes the right k to the next recursive call because of the above caculation.

Tail recursive form of `longestChain`

```
longestChain n = longestChain_tr n 0

longestChain_tr n m
  | n == 1 =  max 1 m
  | otherwise = longestChain_tr (n-1) (max (chainLength n) m)
```

Is the tail-recursive form of `longestChain` better?

Summation of infinite series

1. $tan(x)$ represented as a continued fraction. Only three terms of the fraction are shown:

$$\cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \dots}}}$$

2. The nested expression of square roots

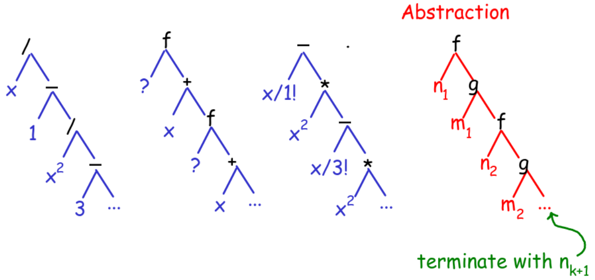
$$\sqrt{x + \sqrt{x + \sqrt{x + \dots}}}$$

3. The series expansion of *sin* as a nested expression:

$$\frac{x}{1!} - x^2 * (\frac{x}{3!} - x^2 * (\frac{x}{5!} - \dots) \dots)$$

Higher order functions

Can we see these computations as instances of a common pattern?



Computing the abstract pattern

```
hof f g n m k i | i <= k = f (n i) (g (m i)(hof f g n m k (i + 1)))
                | otherwise = n (k+1)

tan’ x k =  hof (/) (-)  n m k 1
            where n i = if i == 1 then  x else (x ^ 2)
                  m i = 2 * i - 1

nested_sqrt x k = hof f (+) n m k 1
            where f a b = sqrt b
                  n i = 0
                  m i = x

sin’ x k = hof (-) (*) n m k 1
           where n i = x / factorial (2 * i - 1)
                 m i = x * x
```

A better way to write hof

```
hof f g n m k = hof' 1
      where hof' i | i <= k = f (n i) (g (m i)(hof' (i + 1)))
                   | otherwise = n (i+1)
```

```
tan' x k =  hof (/) (-)  n m k
            where n i = if i == 1 then  x else (x ^ 2)
                  m i = 2 * i - 1
```

A better way to write hof

```
hof f g n m k = hof' 1
      where hof' i | i <= k = f (n i) (g (m i)(hof' (i + 1)))
                   | otherwise = n (i+1)
```

```
tan' x k =  hof (/) (-)  n m k
            where n i = if i == 1 then  x else (x ^ 2)
                  m i = 2 * i - 1
```

What is the type of hof?

A better way to write hof

```
hof f g n m k = hof' 1
      where hof' i | i <= k = f (n i) (g (m i)(hof' (i + 1)))
                   | otherwise = n (i+1)
```

```
tan' x k =  hof (/) (-)  n m k
            where n i = if i == 1 then  x else (x ^ 2)
                  m i = 2 * i - 1
```

What is the type of hof?

```
hof::(Ord a,Num a)=> (t->t2->t)->
                     (t1->t->t2)->
                     (a->t1)->
                     (a->t)->
                     a->
                     t
```

Higer order thinking - Representing a geometric region

Assume that the only use that we shall put a region to is to ask whether a point belongs to it or not.

```
type Point = (Float, Float)
type Region = Point -> Bool
```

`circleMaker` takes a radius and produces a circular region around the origin with the given radius.

```
circleMaker r (x, y) = x ^ 2 + y ^ 2 <= r ^ 2
```

Using lambda notation, we express `circleMaker` as:

```
circleMaker r = \(x, y) -> x ^ 2 + y ^ 2 <= r ^ 2
```

Representing a geometric region

Similarly `rectangleMaker` takes a length and a breadth and produces a rectangle around the origin.

```
rectangleMaker :: Float -> Float -> Region
rectangleMaker  l b = \(x,y) -> (abs x) <= l/2 && (abs y) <= b/2
```

Define the regions notIn, intersection, union, annulus:

```
notIin :: Region -> Region
notIin r = \p -> not(r p)

intersection :: Region -> Region -> Region
intersection r1 r2 = \p -> r1 p && r2 p
```

Representing a geometric region

```
union :: Region -> Region -> Region
union r1 r2 = \p -> r1 p || r2 p

annulus :: Region -> Region -> Region
annulus r1 r2 = intersection r1 (notIn r2)
```

Finally define a function called translate which will translate a region to a given distance:

```
type Distance = (Float, Float)

translate :: Region -> Distance -> Region
translate r (x,y) = \(x1, y1) -> r (x1-x, y1-y)
```

A EMI calculator

Loan amount: 1200000
Interest rate: 10%
Monthly Repayment - 1500

	Balance at beginning of year	Interest for the year	Balance at end of the year
Year 1	1200000	120000	1140000
Year 2	1140000	114000	1074000
Year 3	1074000	107400	1001400
Year 4	1001400	100140	921540
Year 5	921540	92154	833694

A EMI calculator

Loan amount: 1200000
Interest rate: 10%
Monthly Repayment - 1500

	Balance at beginning of year	Interest for the year	Balance at end of the year
Year 1	1200000	120000	1140000
Year 2	1140000	114000	1074000
Year 3	1074000	107400	1001400
Year 4	1001400	100140	921540
Year 5	921540	92154	833694

EMI: For what monthly repayment, does the amount become zero at the end of 5 years?

A EMI (Equated Monthly Installment) calculator.

Loan amount: 1200000
Interest rate: 10%
Monthly Repayment - 26379.58

	Balance at beginning of year	Interest for the year	Balance at end of the year
Year 1	1200000	120000	1003443
Year 2	1003443	100344	787230
Year 3	787230	78723	549396
Year 4	549396	54939	287778
Year 5	287778	28777	0

A EMI (Equated Monthly Installment) calculator.

Loan amount: 1200000
Interest rate: 10%
Monthly Repayment - 26379.58

	Balance at beginning of year	Interest for the year	Balance at end of the year
Year 1	1200000	120000	1003443
Year 2	1003443	100344	787230
Year 3	787230	78723	549396
Year 4	549396	54939	287778
Year 5	287778	28777	0

A EMI is the monthly payment such that the balance is 0 after the given time period.

The EMI for this example is 26379.58

A EMI calculator

The function `balance pr r y mp` gives the balance after `y` years for a monthly payment of `mp`, for a principal of `pr` and an interest rate of `r`:

A EMI calculator

The function `balance pr r y mp` gives the balance after `y` years for a monthly payment of `mp`, for a principal of `pr` and an interest rate of `r`:

```
balance pr r 0 mp = pr
balance pr r y mp = balance newpr r (y-1) mp
     where newpr = (pr + pr * r / 100 - 12 * mp)
```

A EMI calculator

The function `balance pr r y mp` gives the balance after `y` years for a monthly payment of `mp`, for a principal of `pr` and an interest rate of `r`:

```
balance pr r 0 mp = pr
balance pr r y mp = balance newpr r (y-1) mp
     where newpr = (pr + pr * r / 100 - 12 * mp)
```

If `pr`, `r` and `y` are fixed, then `(balance pr r y)` is a function from `mp` to the balance for the given set of values.

A EMI calculator

The function `balance pr r y mp` gives the balance after `y` years for a monthly payment of `mp`, for a principal of `pr` and an interest rate of `r`:

```
balance pr r 0 mp = pr
balance pr r y mp = balance newpr r (y-1) mp
     where newpr = (pr + pr * r / 100 - 12 * mp)
```

If `pr`, `r` and `y` are fixed, then `(balance pr r y)` is a function from `mp` to the balance for the given set of values.

```
b = (balance 1200000 10 5)
```

```
b 1500 = 833694
b 26379 = 0
```

A EMI calculator

EMI is the value of `mp` for which this function returns 0

```
b mp = 0 -- what value of mp satisfies this equation
```

A EMI calculator

EMI is the value of `mp` for which this function returns 0

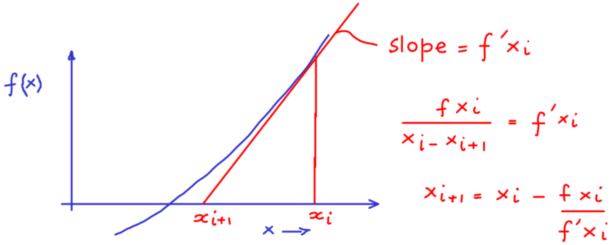
```
b mp = 0 -- what value of mp satisfies this equation
```

If we had a function called `zero` that finds for any function the argument value for which the function returns zero:

```
emi_calc pr r y = zero b
   where b = (balance pr r y)
```

Finding the zero of a function

$(zero\ f)$ is a value x such that $f(x) = 0$.



Approximate using the formula:
$$\begin{aligned} x_{i+1} &= x_i - f(x_i)/f'(x_i) \\ &= improve(x_i) \end{aligned}$$

Finding the zero of a function

First define a function for differentiation:

```
diff f x = (f (x + delta) - f x) / delta
    where delta = 0.0001
```

Next write a function which computes the approximations:
x_0, $f(x_0)$, $f(f(x_0))$, $f(f(f(x_0)))$,..., $f^n(x_0)$

```
until  p  f  x | p x = x
               | otherwise = until p f (f x)
```

Using these, define `zero` as:

```
zero f = until goodenough improve initial
  where initial = 1.0
        improve xi = xi - (f xi)/diff f xi
        goodenough xi = abs (f xi) < 0.0001
```

Integration using Simpson's rule

Simpson's rule: The integral of a function f between a and b is:

$$h/3 * \Sigma_{k=0}^{n} c_k * y_k$$

$h = (b - a)/n$ and n is even
$c_0 = 1$
$c_n = 1$
$c_k = 2$ for $1 < k < n$ and k even.
$c_k = 4$ for $1 < k < n$ and k odd.
$y_k = f(a + k * h)$, $1 < k \leq n$

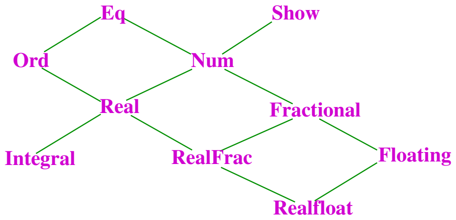
Integration using Simpson's rule

```
simpson f a b n =  (h / 3) * sumseries term n
     where h = (b - a) / n
           c 0 = 1
           c i | i == n = 1
               | even i = 2
               | odd i  = 4
           y k = f (a + k * h)
           term k = (c k) * (y k)


sumseries term 0 = term 0
sumseries term n = sumseries term (n - 1) + term n
```

Integration using Simpson's rule

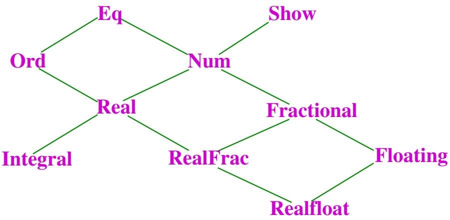
The numeric class hierarchy in Haskell:



1. *Num:* Any numeric type. Coercable
2. *Fractional:* Support non-integral division. Coercable from `Rational`.
3. *Floating:* Supports trigonometric and hyperbolic functions.
4. *Real:* Should be expressible as a ratio.
5. *RealFloat:* Supports operations specific to floats.

Integration using Simpson's rule

The numeric class hierarchy in Haskell:

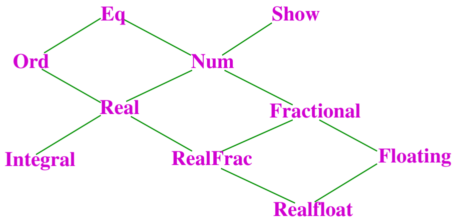



```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

        -- Minimal complete definition: All, except negate or (-)
    x - y            =  x + negate y
    negate x         =  0 - x
```

Integration using Simpson's rule

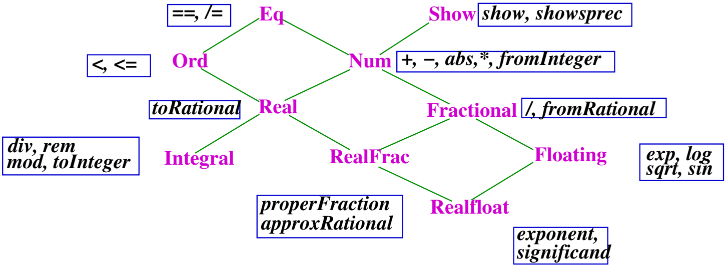
The numeric class hierarchy in Haskell:




```
instance  Num Integer  where
    a + b = ...
    a - b = ...
    a * b = ...
    abs a = ...
    signum a = ...
    fromInteger a = a
```

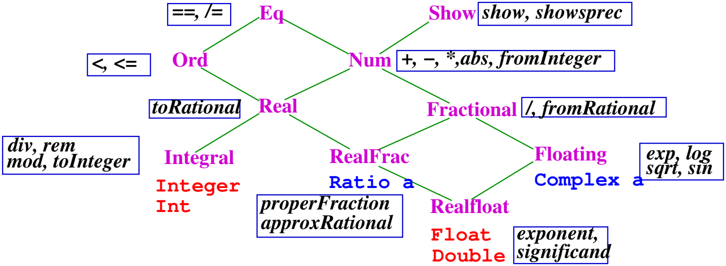
Integration using Simpson's rule

Some of the operators/functions supported by the classes:



Integration using Simpson's rule

The types which belong to the classes:



Integration using Simpson's rule

```
simpson f a b n =  (h / 3) * sumseries term n
     where h = (b - a) / fromInteger n
           c 0 = 1
           c i | i == n = 1
               | i `mod` 2 == 0 = 2
               | i `mod` 2 == 1 = 4
           y k = f (a + k * h)
           term k = (c k) * (y k)


sumseries term 0 = term 0
sumseries term n = sumseries term (n - 1) + term n
```

Integration using Simpson's rule

```
simpson f a b n =  (h / 3) * sumseries term n
     where h = (b - a) / fromInteger n
           c 0 = 1
           c i | i == n = 1
               | i `mod` 2 == 0 = 2
               | i `mod` 2 == 1 = 4
           y k = f (a + fromInteger k * h)
           term k = (c k) * (y k)


sumseries term 0 = term 0
sumseries term n = sumseries term (n - 1) + term n
```

Overloading resolution using the *default* rule

```
*Main> :t 6         -- What is the type of  6?
6 :: Num a => a

*Main> let x = 6  -- Serious business, what is the type of the result
*Main> :t x         -- of this program?
x :: Integer
```

In the second case `x` is assumed to be the result of a program. Similarly:

```
*Main> :t sin 3.1415
sin 3.1415 :: Floating a => a

*Main> let val = sin 3.1415
*Main> :t val
val :: Double
```

Overloading resolution using the *default* rule

- If the type of the result of a program is a overloaded value (ambiguous), it is resolved using the *default rule*.
- *default*s are limited to numeric classes.
- The default *default* is given by the declaration: `default{Integer, Double}`.
- If the type of the result is overloaded, it defaults to the first type in the default list that satisfies all the class contexts in the overloaded type.

Overloading resolution using the *default* rule

Example:

- The overloaded type of the result is:
  `(Show a), (Fractional a) => a -> Int`.
- Assume that the default list is `{Integer, Double}`.
- `Integer` does not satisfy the `(Fractional a)` context.
- `Double` satisfies both `(Show a)` and `(Fractional a)` contexts.
- The type of the result resolves to `Double a -> Int`.
- The default can be altered by a `default` declaration.