CS-4620 Functional Programming I 2013-2014

Lecturer:

Joseph Manning / Western Gateway 1.80 / ☎ : 420 5909 / ✉ : manning@cs.ucc.ie

Lectures (Period 1) :

Mon : 11:00am – 12:00nn : Western Gateway G.18
Wed : 11:00am – 12:00nn : Western Gateway G.13

Labs:

This module does not have formal labs. Students are expected to complete the programming assignments in their own time. You may use the computers in the Western Gateway G.21 lab.

Recommended Book:

"Programming in Haskell", Graham Hutton (Cambridge University Press, 2007).

Web Sites:

`www.haskell.org`
`www.learnyouahaskell.com`
`www.cs.nott.ac.uk/~gmh/book.html`
`book.realworldhaskell.org`

Module Overview:

This module presents an introduction to *functional programming*, using the language *Haskell*.

This style of programming has a long history, with its foundations pre-dating the development of electronic computers, yet it offers perhaps the best prospects for taking advantage of the emerging power and availability of multi-core processors. It is radically different from the more conventional style of *imperative programming*, and functional programs are more compact, and often considered more elegant, than their imperative counterparts.

The central theme of *functional programming* is to *define* the solution to a given problem; the task of the computer is to evaluate this definition and find the corresponding solution. Functional programming does without the concepts of *variables* and *assignment*. Moreover, in some sense there is no notion of *time* in the execution of a functional program.

By contrast, the central theme of *imperative programming* is to write a list of instructions; the task of the computer is to carry out these instructions, and thus construct the solution. The concepts of *variables* and *assignment*, along with the progressive execution of a program over *time*, are central to imperative programming,

Despite this contrast, however, the ideas encountered and the experience gained in this module may have a significant positive impact even when reverting to imperative programming.

Grading:

70% : Summer Examination
30% : Year's Work

Year's Work consists of regular assignments, each requiring several short practice programs. Assignments have strict due dates; late submissions will not be accepted, unless under extreme and verifiable circumstances. All students are expected to work *individually* on assignments; those found collaborating with others will receive a score of zero for their work.

Three Golden Tips:

- *attend the lectures*
- *keep up to date*
- *do all the assignments*

Core-Haskell

Reference Manual

Joseph Manning
Department of Computer Science
University College Cork

September 2013

Core-Haskell is a very small subset of the programming language Haskell.

Yet it is remarkably powerful and contains many of the key ideas of the full language.

Its purpose is to reveal the essence of functional programming in a simplified setting.

It was designed and implemented at UCC by Haodong Guo and Joseph Manning.

Throughout this manual, the term 'Core-Haskell' will be written as 'C-H'.

PROGRAMS

A C-H program is composed of a sequence of **expressions** and/or **definitions**. Each of these expressions is evaluated, and its value is written out; each definition attaches a name to an item.

ITEMS and EXPRESSIONS

The atomic data entities occurring in C-H are called **items**. There are four types of items in C-H :

numbers, booleans, lists, functions.

Items are denoted by means of **expressions**. For example, each of the expressions 2 + 3, 5, 9 - 4 denotes the same item, the number 5 . Every expression is simply a means of denoting an item, and it may always be replaced by any other expression which denotes the same item.

Apart from processing definitions, the C-H interpreter is just an expression simplifier; it reads in expressions, simplifies them, and then writes out the items which they denote. Thus, for example, the input 2 + 3 produces the output 5 .

NUMBERS

A **number** is an integer, written as a sequence of decimal digits, possibly preceded by a '-' sign

e.g. `5, 0, -28, 4371`

The arithmetic operators

`+` (infix)   `-` (prefix, infix)   `*` (infix)

and the arithmetic functions

`div` (prefix)   `mod` (prefix)

generate number items from number items. For example,

`17+3  ⇒  20,  17-3  ⇒  14,  17*3  ⇒  51,  div 17 3  ⇒  5,  mod 17 3  ⇒  2`

BOOLEANS

A **boolean** is one of the two values `True` and `False` .

The boolean operators

`&&` (and; infix)   `||` (or; infix)

and the boolean function

`not` (prefix)

generate boolean items from boolean items.

LISTS

A **list** is an ordered sequence of items called its **components**. A list is either of the form

`[]`   (the empty list)

or

`h : t`   (`h` an item, `t` a list).

Colon (:) is an infix operator whose left operand `h` is an item and whose right operand `t` is a list, and which produces a new list one component longer by attaching the item to the front of the list.

LAZY EVALUATION

An important and powerful aspect of C-H is its use of *lazy evaluation*: operations are not performed unless/until their results are actually needed. This has several significant consequences:

Call-by-Need: When a function is applied to an argument, the argument remains unevaluated until its value is needed within the function. In particular, if its value is never needed, it is never evaluated; thus, `( \ n -> 2 + 3 ) ( div 1 0 )` evaluates to 5, rather than giving an error.

Short-Circuit Boolean Operators: Unless its value is needed, the second operand of `&&` or `||` is not evaluated; thus

if *P* evaluates to `False` then *P* `&&` *Q* is known to be `False`

if *P* evaluates to `True` then *P* `||` *Q* is known to be `True`

and in each case *Q* need not, and will not, be evaluated.

Lazy Definitions: When a definition is processed, its <*EXPRESSION*> is not actually *evaluated*, and thus can contain occurrences of <*NAME*>s which have yet to be defined. This allows:

Recursive Definitions:

```
factorial = \ n -> if n == 0 then 1 else n * factorial ( n - 1 )
```

Forward Definitions:

```
a = b + 1
b = 4
```

Mutually Recursive Definitions:

```
iseven = \ n -> n == 0 || isodd  ( n - 1 )
isodd  = \ n -> n /= 0 && iseven ( n - 1 )
```

Infinite Lists: The `:` operator is lazy, which allows infinite lists to be specified and manipulated with ease in C-H. For example,

```
ones = 1 : ones
```

defines the infinite list `1 : 1 : 1 : 1 : ...`, while

```
from = \ n -> n : from ( n + 1 )
```

results in `from 1` being the infinite list `1 : 2 : 3 : 4 : 5 : ...`.

Entire infinite lists are never actually *constructed*; instead, their components are produced only upon demand, with the list being expanded just as far as is strictly necessary. For example,

```
head ( tail ( from 1 ) )
⇒ head ( tail ( 1 : from ( 1 + 1 ) ) )
⇒ head ( from ( 1 + 1 ) )
⇒ head ( ( 1 + 1 ) : from ( ( 1 + 1 ) + 1 ) )
⇒ 1 + 1
⇒ 2
```

FURTHER DETAILS

Operator Precedence: The operators of C-H, in decreasing order of precedence, are as follows:

```
Function Application ( note that div, mod, not, head, tail are functions )
*
+  -
:
==  /=  <  <=  >  >=
&&
||
```

In evaluating a multi-operator expression, operators of higher precedence are applied before those of lower precedence. Operators of equal precedence are applied from left-to-right, apart from `:` which is applied from right-to-left. However, parentheses may be used to override precedence and explicitly control the order of application of operators; this is the *only* use of parentheses in C-H.

Syntax of a <*NAME*>**:** A <*NAME*> must start with a *lower*-case letter, and can continue with any sequence of lower-case letters, upper-case letters, digits, or the characters `'''` or `'_'`. The words `if`, `then`, `else` are reserved and cannot be used as <*NAME*>s.

Scope: The scope of the <*NAME*> in a *function* is the associated <*EXPRESSION*>, apart from any nested functions in which that <*NAME*> is re-used; the scope of the <*NAME*> in a *definition* is the entire program, apart from any functions in which that <*NAME*> is re-used. For example, with the definition

```
n = 1
```

the following top-level expression has the value 12 (= 1 + 3 + 2 * 2 + 3 + 1) :

```
n + ( \ n -> ( n + ( \ n -> n * n ) 2 + n ) ) 3 + n
```

Expressions Evaluated Once: A <*NAME*> becomes bound to an <*EXPRESSION*> during either a function application or a definition. Under lazy evaluation, when that <*NAME*> is first used, the <*EXPRESSION*> is evaluated; at that point, *all* occurrences of the <*NAME*> are re-bound to the *value* of the <*EXPRESSION*>. Thus, for example, in the function application

```
( \ n -> n * n ) ( 2 + 3 )
```

the expression 2 + 3 is evaluated only *once*; likewise, with the definition

```
hoursperweek = 24 * 7
```

the expression `24 * 7` will be evaluated only the *first* time that `hoursperweek` is used.

Format of Output: Output is produced when a top-level expression is evaluated.

A **number** item is written out in standard decimal form.

A **boolean** item is written out as `True` or `False`.

A **list** item is written out by writing out each component, with sublists in parentheses, separated by `:` symbols and terminated by `[]`.

A **function** item is simply written out as `<FUNCTION>`.

Comments: The symbol `--` introduces a *comment*, and all further text on that line is ignored.

Code Layout: Blanks, tabs, and newlines are called *whitespace* characters. When writing C-H, whitespace may be used freely to separate tokens. C-H does not strictly enforce the 'offside rule' of Haskell; however, in multi-line definitions, lines after the first one must be indented.

Simple Recursive Functions on Lists

```
----------------------------------------------------------------------- null

-- null xs : is list 'xs' empty ?

null = \xs -> xs == []

----------------------------------------------------------------------- length

-- length xs : the number of components in list 'xs'

length = \xs -> if null xs then
                   0
                else
                   1 + length ( tail xs )

----------------------------------------------------------------------- elem

-- elem x xs : does item 'x' occur in list 'xs' ?

elem = \x -> \xs -> not ( null xs )
                    &&
                    ( x == head xs || elem x ( tail xs ) )

----------------------------------------------------------------------- count

-- count x xs : the number of times that item 'x' occurs in list 'xs'

count = \x -> \xs -> if null xs then
                        0
                     else
                        ( if x == head xs then 1 else 0 ) + count x ( tail xs )

----------------------------------------------------------------------- append

-- append xs ys : the list formed by joining lists 'xs' and 'ys', in that order

append = \xs -> \ys -> if null xs then
                          ys
                       else
                          head xs : append ( tail xs ) ys

--------------------------------------------------------------------------------
```

```
===>>  C o r e - H a s k e l l   I n t e r p r e t e r          2013-Oct-02 09:26

> :load simple

-------------------------------------------------- simple.hs

> length []

0

-------------------------------------------------- 13 evals

> length ( True : False : True : [] )

3

-------------------------------------------------- 73 evals

> elem 4 ( 1 : 2 : 3 : 4 : 5 : [] )

True

-------------------------------------------------- 107 evals

> elem 8 ( 1 : 2 : 3 : 4 : 5 : [] )

False

-------------------------------------------------- 209 evals

> count 5 ( 3 : 5 : 2 : 5 : 5 : 7 : 1 : 5 : 6 : [] )

4

-------------------------------------------------- 320 evals

> count 5 []

0

-------------------------------------------------- 15 evals

> append ( 5 : 2 : 6 : [] ) ( 1 : 4 : [] )

5 : 2 : 6 : 1 : 4 : []

-------------------------------------------------- 107 evals

> append [] ( 1 : 2 : 3 : [] )

1 : 2 : 3 : []

-------------------------------------------------- 22 evals

> append ( 1 : 2 : 3 : [] ) []

1 : 2 : 3 : []

-------------------------------------------------- 103 evals
```

The Higher-Order Function ‘foldr’

```
---------------------------------------------------------------------- foldr

-- foldr f z xs : the result of appending item ‘z’ to the right end of list ‘xs’
--                and then cumulatively applying the two-parameter function ‘f’
--                from right to left on this augmented list

foldr = \f -> \z -> \xs -> if null xs then
                              z
                           else
                              f ( head xs ) ( foldr f z ( tail xs ) )

---------------------------------------------------------------------- sum

-- sum ns : the sum of all items in the numeric list ‘ns’

sum = foldr ( \n1 -> \n2 -> n1 + n2 ) 0

---------------------------------------------------------------------- product

-- product ns : the product of all items in the numeric list ‘ns’

product = foldr ( \n1 -> \n2 -> n1 * n2 ) 1

---------------------------------------------------------------------- factorial

-- factorial n : the number  1 * 2 * ... * n  for a non-negative integer ‘n’

factorial = \n -> product ( range 1 n )  -- range : SEE Assignment #1

---------------------------------------------------------------------- and

-- and bs : do all components of the boolean list ‘bs’ equal ‘True’ ?

and = foldr ( \b1 -> \b2 -> b1 && b2 ) True

---------------------------------------------------------------------- or

-- or bs : does any component of the boolean list ‘bs’ equal ‘True’ ?

or = foldr ( \b1 -> \b2 -> b1 || b2 ) False

---------------------------------------------------------------------- all

-- all p xs : do all components of list ‘xs’ satisfy predicate ‘p’ ?

all = \p -> \xs -> and ( map p xs )

---------------------------------------------------------------------- any

-- any p xs : does any component of list ‘xs’ satisfy predicate ‘p’ ?

any = \p -> \xs -> or ( map p xs )

---------------------------------------------------------------------- elem

-- elem x xs : does item ‘x’ occur in list ‘xs’ ?

elem = \x -> any ( \e -> e == x )

-------------------------------------------------------------------------------
```

```
---------------------------------------------------------------------- length

-- length xs : the number of components in list ‘xs’

length = foldr ( \x -> \acc -> acc + 1 ) 0

---------------------------------------------------------------------- map

-- map f xs : the list formed by applying function ‘f’
--            to each component of list ‘xs’

map = \f -> foldr ( \x -> \acc -> f x : acc ) []

---------------------------------------------------------------------- filter

-- filter p xs : the list formed by those components of list ‘xs’
--               which satisfy predicate ‘p’

filter = \p -> foldr ( \x -> \acc -> if p x then x : acc else acc ) []

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

> sum ( 3 : 5 : 2 : 1 : 4 : [] )

15

---------------------------------------------------- 208 evals

> factorial 5

120

---------------------------------------------------- 308 evals

> elem 5 ( 1 : 2 : 3 : 4 : 5 : 6 : 7 : [] )

True

---------------------------------------------------- 359 evals

> length ( 8 : 4 : 3 : 1 : 7 : [] )

5

---------------------------------------------------- 178 evals

> map ( \n -> n * n ) ( 1 : 2 : 3 : 4 : 5 : [] )

1 : 4 : 9 : 16 : 25 : []

---------------------------------------------------- 250 evals

> filter ( \n -> mod n 2 == 0 ) ( 1 : 2 : 3 : 4 : 5 : [] )

2 : 4 : []

---------------------------------------------------- 286 evals
```

Infinite Lists : Fibonacci Numbers and Prime Numbers

```
--------------------------------------------------------------------------
-- Generating Fibonacci Numbers : Exponential-Time and Linear-Time Algorithms
--------------------------------------------------------------------------

-- fibsSlow : the infinite list of Fibonacci Numbers : 0, 1, 1, 2, 3, 5, 8, ...

fibsSlow = map fib ( from 1 )

-- fib n : the 'n'th Fibonacci number, for any positive integer 'n'

fib = \n -> if n == 1 then
               0
            else
            if n == 2 then
               1
            else
               fib ( n - 1 ) + fib ( n - 2 )

--------------------------------------------------------------------------

-- fibsFast : the infinite list of Fibonacci Numbers : 0, 1, 1, 2, 3, 5, 8, ...

fibsFast = 0 : 1 : zipWith ( \f1 -> \f2 -> f1 + f2 ) fibsFast ( tail fibsFast )

--------------------------------------------------------------------------
--------------------------------------------------------------------------

> take 16 fibsSlow

0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89 : 144 : 233 : 377 : 610 : []

------------------------------------------------------------ 83669 evals

> take 16 fibsFast

0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89 : 144 : 233 : 377 : 610 : []

------------------------------------------------------------ 1403 evals
```

```
--------------------------------------------------------------------------
-- Generating Primes Numbers : Sieve of Eratosthenes
--------------------------------------------------------------------------

-- primes : the infinite list of prime numbers : 2, 3, 5, 7, 11, 13, 17, ...

primes = sieve ( from 2 )

--------------------------------------------------------------------------

-- sieve ns : the result of applying the Sieve of Eratosthenes to the list 'ns'

sieve = \ns -> head ns : sieve ( dropMultiples ( head ns ) ( tail ns ) )

--------------------------------------------------------------------------

-- dropMultiples d ns : the numeric list 'ns' with all multiples of 'd' removed

dropMultiples = \d -> filter ( \n -> mod n d /= 0 )

--------------------------------------------------------------------------
--------------------------------------------------------------------------

> take 16 primes                                  -- the first 16 primes

2 : 3 : 5 : 7 : 11 : 13 : 17 : 19 : 23 : 29 : 31 : 37 : 41 : 43 : 47 : 53 : []

------------------------------------------------------------ 9371 evals

> take 16 primes                                  -- again, now faster

2 : 3 : 5 : 7 : 11 : 13 : 17 : 19 : 23 : 29 : 31 : 37 : 41 : 43 : 47 : 53 : []

------------------------------------------------------------ 583 evals

> takeWhile ( \p -> p <= 40 ) primes              -- the primes below 40

2 : 3 : 5 : 7 : 11 : 13 : 17 : 19 : 23 : 29 : 31 : 37 : []

------------------------------------------------------------ 624 evals

> head ( drop 99 primes )                         -- the 100th prime

541

------------------------------------------------------------ 277839 evals

> head ( dropWhile ( \p -> p <= 1000 ) primes )   -- the first prime above 1000

1009

------------------------------------------------------------ 502815 evals

> head ( dropWhile ( \p -> p <= 1000 ) primes )   -- again, now faster

1009

------------------------------------------------------------ 6930 evals
```

The Functions ‘take’, ‘drop’, ‘takeWhile’, ‘dropWhile’, ‘zipWith’

```
--------------------------------------------------------------------------------

-- take n xs : the list of the first 'n' components of 'xs',
--             or 'xs' itself if 'n' exceeds its length

take = \n -> \xs -> if n <= 0 || null xs then
                       []
                    else
                       head xs : take ( n - 1 ) ( tail xs )

--------------------------------------------------------------------------------

-- drop n xs : the list 'xs' with the first 'n' components removed,
--             or the empty list if 'n' exceeds its length

drop = \n -> \xs -> if n <= 0 || null xs then
                       xs
                    else
                       drop ( n - 1 ) ( tail xs )

--------------------------------------------------------------------------------

-- takeWhile p xs : the longest prefix of 'xs' whose components
--                  all satisfy predicate 'p'

takeWhile = \p -> \xs -> if null xs || not ( p ( head xs ) ) then
                            []
                         else
                            head xs : takeWhile p ( tail xs )

--------------------------------------------------------------------------------

-- dropWhile p xs : the longest suffix of 'xs' whose first component
--                  does not satisfy predicate 'p'

dropWhile = \p -> \xs -> if null xs || not ( p ( head xs ) ) then
                            xs
                         else
                            dropWhile p ( tail xs )

--------------------------------------------------------------------------------

-- zipWith f xs ys : the list formed by applying function 'f' to pairs
--                   of corresponding components in lists 'xs' and 'ys',
--                   stopping as soon as either list is exhausted

zipWith = \f -> \xs -> \ys -> if null xs || null ys then
                                 []
                              else
                                 f ( head xs ) ( head ys )
                                    : zipWith f ( tail xs ) ( tail ys )

--------------------------------------------------------------------------------
```

```
> take 4 ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

7 : 3 : 5 : 8 : []

-------------------------------------------------------------- 150 evals

> take 0 ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

[]

-------------------------------------------------------------- 12 evals

> take 9 ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : []

-------------------------------------------------------------- 308 evals

> drop 4 ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

4 : 1 : 9 : 2 : []

-------------------------------------------------------------- 133 evals

> drop 0 ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : []

-------------------------------------------------------------- 29 evals

> drop 9 ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

[]

-------------------------------------------------------------- 245 evals

> takeWhile ( \n -> mod n 2 == 1 ) ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

7 : 3 : 5 : []

-------------------------------------------------------------- 210 evals

> dropWhile ( \n -> mod n 2 == 1 ) ( 7 : 3 : 5 : 8 : 4 : 1 : 9 : 2 : [] )

8 : 4 : 1 : 9 : 2 : []

-------------------------------------------------------------- 197 evals

> zipWith ( \n1 -> \n2 -> n1 + n2 ) ( 1 : 2 : 3 : [] ) ( 4 : 5 : 6 : [] )

5 : 7 : 9 : []

-------------------------------------------------------------- 194 evals

> zipWith ( \n1 -> \n2 -> n1 * n2 ) ( 1 : 2 : [] ) ( 4 : 5 : 6 : 7 : [] )

4 : 10 : []

-------------------------------------------------------------- 133 evals
```

8/10

```
-- Title: Assignment 2
-- Student Name: Brian O Regan
-- Student ID: 110707163
-- Due Date: Fri 31st October 2013 @4.00pm

-- Question 1
-- Returns a list is formed by calling each function
-- in function list 'fs' on item 'x'
applyAll = \fs -> \x -> map (\f -> f x) fs
```

✓

```
-- Question 2
-- Returns a list formed by those components of list 'xs',
-- which do not satisfy predicate 'p'
remove = \p -> \xs -> filter(\x -> not (p x)) xs
```

ok, but can omit

```
-- Question 3
-- Returns the number of times that item 'x' occurs in list 'xs'
count = \x -> \xs -> foldr(\y -> \n -> if x == y then n+1 else n) 0 xs

-- Questions 4
-- Return the maximum number in the non-empty numeric list 'ns'
maximum = \ns -> foldr(\x -> \y -> if x>y then x else y)(head ns)
```

(tail ns)

this 'foldr' needs one more argument

```
-- Question 5
-- This returns a list formed by joining 'xs' and 'ts', in that order
append = \xs -> \ts -> foldr(\x -> \ys -> x:ys) xs ts
```

✓

5/10

```
-- Title: Assignment 1
-- Student Name: Brian O Regan
-- Student ID: 110707163
-- Due Date: Mon 14th October 2013 @10:30am

-- Null Definition
-- null = \bs -> bs == []

-- Question 1
-- Check if all items in a list to see if they are the same.
-- Return True if all the same and False otherwise.
and = \bs -> not(null bs) && ((if b == head bs && head(tail bs)
&& head (head(tail bs))then True else False))
```

what's b ?

```
-- Why?
-- This is because the first successful match is taken,
-- in this case it is True, even though there are no other elements,
-- it still returns True, because the first item checked is True and
-- there is nothing to compare it to, so it remains True.
```

✓

```
-- Question 2
-- Check if any items in a list are true.
-- If they are all False then return False, otherwise return True.
or = \bs -> ((if b == head || head(tail bs)
&& head (head(tail bs))then True else False))
```

? head bs

```
-- Why?

-- Question 3
-- Check if the numeric list is sorted in ascending order.
-- Return True if it is and False otherwise.
issorted = \ ns -> (tail ns == [] || (head ns <= head(tail ns)
&& issorted(tail ns)))
```

what if ns == [] ? this test will crash

```
-- Questions 4
-- List the range of numbers from the lowest number entered to the highest.
-- Return ~~Print~~ a list of numbers from a range.
-- The user inputs the lo and hi numbers of the range
range = \lo -> \hi -> (if hi < lo then [] else lo:(range (lo+1) hi))
```

✓

```
-- Question 5
-- Create a list where the user enters the number of copies,
-- it wants of an item.
copies = \n -> \x -> (if n <= 0 then [] else x : (copies (n-1) x))
```

✓

- this code does not actually run, due to faulty indentation (see the "Code Layout" section of the Core-Haskell manual)
- can you eliminate 'if-then-else' from and, or, issorted ?

Assignment #1

Simple Core-Haskell Functions

Write definitions for each of the following *Core-Haskell* functions.

For each function, include a clear and concise *comment* to describe its purpose.

Note that the function 'null' is already defined in the Core-Haskell Standard Prelude.

1. and bs

   Do all components of the boolean list 'bs' equal 'True' ?

   ```
   and ( ( 5<6 ) : not False : ( 1+2 == 3 ) : [] )  ⇒  True
   and ( True : True : False :True : [] )            ⇒  False
   and []                                            ⇒  True    (why?)
   ```

2. or bs

   Does any component of the boolean list 'bs' equal 'True' ?

   ```
   or ( ( 5>6 ) : not True : ( 1+2 == 4 ) : [] )  ⇒  False
   or ( False : False : True :False : [] )        ⇒  True
   or []                                          ⇒  False    (why?)
   ```

3. issorted ns

   Is the numeric list 'ns' sorted in ascending order?

   ```
   issorted ( 2 : 3 : 3 : 7 : [] )  ⇒  True
   issorted ( 5 : [] )              ⇒  True
   issorted []                      ⇒  True
   issorted ( 1 : 2 : 4 : 3 : [] )  ⇒  False
   ```

4. range lo hi

   The list of numbers from the number 'lo' up to the number 'hi', inclusive

   ```
   range 3 7  ⇒  3 : 4 : 5 : 6 : 7 : []
   range 3 3  ⇒  3 : []
   range 3 2  ⇒  []
   ```

5. copies n x

   The list of 'n' copies of item 'x' (assume that 'n' is a non-negative integer)

   ```
   copies 4 7       ⇒  7 : 7 : 7 : 7 : []
   copies 0 True    ⇒  []
   ```

Program Submission:

Store the function definitions in a file named 'a1.hs', and turn it in for grading by typing:

```
submit-cs4620 a1.hs
```

Due Date: Mon Oct 14, 10:30am

Assignment #2

Higher-Order Functions

Write *non-recursive* definitions for each of the following *Core-Haskell* functions.
For each function, include a clear and concise *comment* to describe its purpose.

1. `applyAll fs x`

   The list formed by calling each function in function list 'fs' on item 'x'

   `applyAll ( (\n -> n+1) : (\n -> -n) : (\n -> n*n) : [] ) 3  ⇒  4 : -3 : 9 : []`

2. `remove p xs`

   The list formed by those components of list 'xs' which do not satisfy predicate 'p'

   `remove ( \n -> n<0 ) ( 3 : -2 : -5 : 7 : 4 : -1 : [] ) ⇒  3 : 7 : 4 : []`

3. `count x xs`

   The number of times that item 'x' occurs in list 'xs'

   `count 5 ( 3 : 5 : 2 : 5 : 5 : 7 : 1 : 5 : 6 : [] ) ⇒  4`

4. `maximum ns`

   The maximum number in the non-empty numeric list 'ns'

   `maximum ( 4 : 2 : 7 : 1 : 5 : 9 : 8 : 6 : [] ) ⇒  9`

5. `append xs ys`

   The list formed by joining lists 'xs' and 'ys', in that order

   `append ( 5 : 8 : 3 : [] ) ( 4 : 7 : [] ) ⇒  5 : 8 : 3 : 4 : 7 : []`

Program Submission:

Store the function definitions in a file named 'a2.hs', and turn it in for grading by typing:

```
submit-cs4620 a2.hs
```

Due Date: Fri Oct 25, 4:00pm

Assignment #3

Infinite Lists

Write efficient and compact definitions for each of the following *Core-Haskell* items.
For each item, include a clear and concise *comment* to describe its purpose.

1. `partialSums ns`

   The list of partial sums of the numeric list 'ns'

   ```
   partialSums ( 3 : 2 : 4 : 5 : 2 : [] )  ⇒  3 : 5 : 9 : 14 : 16 : []
   partialSums []                          ⇒  []
   partialSums ( from 1 )                  ⇒  1 : 3 : 6 : 10 : 15 : 21 : 28 : . . .
   ```

2. `powers n`

   The list of all positive powers of the number 'n'

   ```
   take 7 ( powers 2 )       ⇒  2 : 4 : 8 : 16 : 32 : 64 : 128 : []
   take 7 ( powers (-1) )  ⇒  -1 : 1 : -1 : 1 : -1 : 1 : -1 : []
   ```

3. `factorials`

   The list of factorials of all positive integers

   ```
   take 7 factorials  ⇒  1 : 2 : 6 : 24 : 120 : 720 : 5040 : []
   ```

Program Submission:

Store the definitions in a file named 'a3.hs', and turn it in for grading by typing:

```
submit-cs4620 a3.hs
```

Due Date: Wed Nov 13, 10:30am

The *Core-Haskell* Standard Prelude includes the following pre-defined functions:

```
div      mod      even       odd
head     tail
not
null     length   reverse    elem
map      filter   foldr
sum      product
and      or       all        any
take     drop     takeWhile  dropWhile
zipWith
from
```

CS4620 Monday 16th September 2013

Joseph Manning

Programming

Imperative
(Java, C,
Phyton, Fortran
VB etc)
- Variables
- Assignment.
- Time

Functional
(Haskell, ML, LISP etc)
- No variables
- No statements.
- No time

λ - Calculus (landa Calculus)

Haskell (compiler) GHC
(Glasgow Haskell Compiler)

CS4620 Wednesday 18th September 2013

Haskell (1989 - collab. with a number of other functional programming languages around at the time).

→ Core Haskell

eCopy of handout ⇒ /users/staff/joseph/cs4620/ch.pdf.

Expressions & Definitions

4 different data types in CH.

- Numbers
- Booleans
- Lists
- Functions

<u>List</u>

Empty or Not Empty list

[] or h : t

↙ ↘

elem list

first item in list is the head, the remainder is referred to as the tail.

CS4620 Monday 23rd September 2013

$ ch to enter Core Haskell Interpreter

Atomic Data ⇒ can't simplify any further.

- > :load primes
  > take 10 primes ⇒ first 10 prime numbers

- take – have a list and take n from that list eg take 10

evals = evaluations – the number of times (computational effort required) to complete the table.

2 + 3 is the answer, what CH does is simplify the answer i.e 5.

- > :load sort
  > sort (5 : 7 : 2 : 8 : 1 : 6 : 9 : 4 : [ ])
  the above sorts the list in ascending order.
  Once sorted, you can not sort it any simpler.

- ctrl + D - stops it and returns to regular shell

▲ Linux Only

Add to path : /users/staff/joseph/bin

then you can type $ch.

Install CH.

/users/staff/joseph/CH/INSTALL

the above is a simple text file with instructions to install it on own pc.

CH is written in Lua, so Lua interpretter is required. (maybe required)

CH

- Expressions
-- Definition

} two main elements of CH.

List

[ ] : [ ]   perfectly acceptable.

↑ : ↑

item   list

also [ ] : [ ] : [ ] is ok.

$$\underbrace{3 : 7 : 5}_{\text{3 components}} : [\,]$$

$$\underbrace{[\,] \quad [\,]}_{\text{2 components}} : [\,]$$

To decompose a [ ] we use head and tail

head of (2 : 7 : 5 : [ ]) = 2
tail of (2 : 7 : 5 : [ ]) = 7 : 5 : [ ]

to get the second item we would.

head (tail (2 : 7 : 5 : [ ])) = 7
i.e. you take the head of the tail.

Functions

\ <NAME> --> <EXPRESSION>

$\boxed{\backslash = \lambda \quad \text{Lambda}}$

example

$$\backslash \; \underset{\text{Argument}}{\underset{\downarrow}{n}} \longrightarrow \underset{\text{Result.}}{\underset{\downarrow}{n * n}}$$

$\boxed{\text{n is not a variable it is a parameter.}}$

$(\backslash\ n \rightarrow n * n)\ 2$

If you want to get the square root of a number

$(\backslash\ \underbrace{n \rightarrow n * n)\ 2}_{\text{link these two}}$

i.e set n to 2 and the evaluate

$(\backslash\ 2 \rightarrow \underbrace{2 * 2}_{4})\ 2 = 4$

All you doing is simplifying an expression as (\2 -> 2 * 2) 2 is an answer but 4 is a simpler way of expressing it.

CS4620 Wednesday 25th September 2013

16 Combinations allowed (4, 4, 4, 4)
The function takes only 1 argument. (ONLY!)

To add two ~~nuubers~~ numbers (for example)

(\ ns -> head ns + head (tail ns))

eg (2 : 3 : [ ]) => 5

A much more eligant way (and more powerful) is :
first bind the number to parameter

$$(\backslash\ n1 \to \underbrace{(\backslash\ n2 \to n1 + n2))}\ 2$$

$$(\backslash\ n2 \to \underbrace{2 + n2})\ \ 3$$

$$\backslash\ 3 \to \underbrace{2 + 3}$$

$$\downarrow$$

$$5$$

OR

$$(\backslash\ n1 \to (\backslash\ n2 \to n1 + n2))\ \ 2\ \ 3$$

$$2 \to (\backslash\ n2 \to \underbrace{2 + n2})$$

$$\downarrow$$

$$\underbrace{2 + 3}$$

$$5$$

n2 does not get evaluated straight away *
n1 eval first and then n2

Curried Functions
(founded by Haskell Curry)
That is where the previous functions got their name from

Relational Operators (== or /=)

$$\underbrace{1 + 1 \;==\; 2}_{\substack{\downarrow \\ \text{TRUE}}}$$

In haskell you are only allowed to compare items of the same type (int == int , boolean == boolean etc).

1 : 2 : [ ] == 8 - 7 : 1+1 : [ ] => TRUE

⬇

1 : 2 : [ ] == 1 : 2 : [ ]

1 and 8 - 7 are the same thing but represented a different way, likewise with 2 and 1+1.

Important : The order of the items matters

You cannot compare two functions to see if they are the same.

if < condition > then < EXP - 1 > else < EXP - 2 >

An expression is something that denotes an item. The value it expresses is either val 1 or val 2 depending on the condition ie boolean TRUE or FALSE.

```
\n -> if n >= 0 then n else -n
```

CS4620 Monday 30th September 2013

howsperweek = 24 * 7 (168)
(name = expression)

This is not an assignment but rather a definition. What we are doing is taking an item and attaching an expression to it.

We can use howsperweek or 24 * 7 or 168. It is always the same value, so it is not a variable as a variable can change but the amount of hours per week does not change. A variable for example n could equal to 1 at the start but be 3 later on.

You can not type a definition into an interactive interpreter. You must store it in a file and load it in. It only allows expressions. The files are ordinary txt files but must finish with .hs eg demo.hs Inside CH you type > :load demo.hs , then you can start using the definitions (you can leave out the .hs in the CH interpreter eg > :load demo.

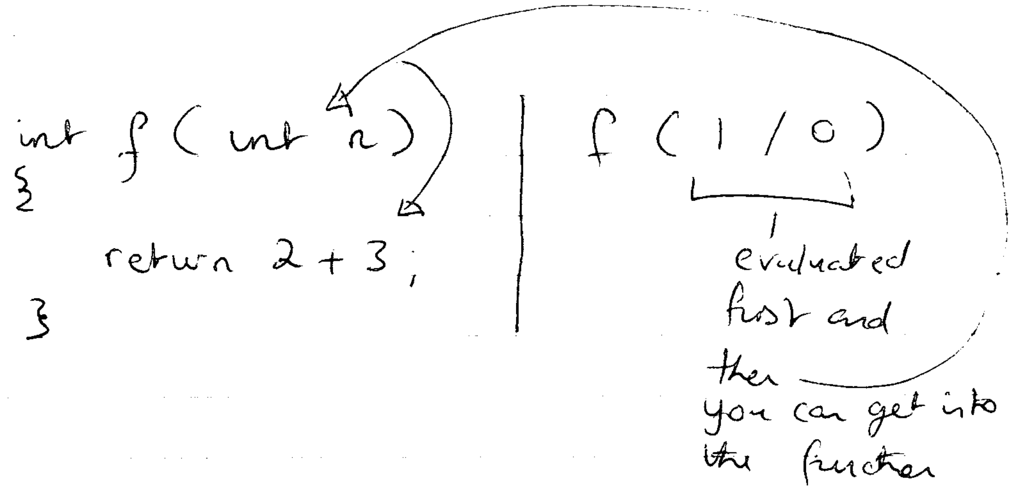
★ Operations are not performed unless/until their results are actually needed.

◆ Call-By-Need - I need this right now!

Call a function

eg $(\backslash\ n \to 2 + 3)\ (\underbrace{\text{div } 1\ 0}_{\text{in haskell no error messages}})$

In C;



```
int f ( int n )
{
    return 2 + 3;
}
```

f (1 / 0)

evaluated first and then you can get into the function

In Haskell there is no need to evaluate (1/0) because it is not <u>needed</u>!
But if the function was : (\n -> n + 3)(div 10
then the (div 1 0) is needed because it is part of the expression so 1 would be divided by 0 here.

```
a = b + 1
   :
b = 4
```

This is ok in haskell, as haskell doesn't evalute a say that b is not defined when it finds b = 4 later then it is able to evalutate

Lists -> there is the potential for infinite lists.

not = \x -> if x then False else True

- Precidence same as other languages (4 - 5 * 3)
  5 * 3 first.

2 : 3 : 4 : [ ]

But with : it is reversed it works Right to LeFT.

[ ] -> 4 -> 3 -> 2

Names must be in lowercase letter (begin with)
eg hELLO or iI43KaeKb and after that
any variation of letter or digit.
Uppercase are used for other reasons True / False
for example

```
\ x -> 2 + ((\ x -> x * x ) 5) + x
```

these
x's are
different from eachother

The most local thing (x) applies.

CS4620   Monday 7th October 2013

```
length = \ xs -> if null xs then 0
                 else 1+ length (-tail xs
```

◆ Higher Order Functions

A function is higher order if it takes a function as argument and/or returns a function as result.

◦ doublelist (4 : 1 : 3 : []) => 8 : 2 : 6 : []

```
→ doublelist = \ns -> if null ns then
                      []
                      else
                      2 * (head ns) : double list (-tail ns
```

◦ fliplist (4 : 1 3 : []) => 3 : 1 : 4 : []

```
→ fliplist = \ ns -> if null ns then
                     []
                     else
                     not (head ns) : fliplist (tail ns)
```

same thing but shorter

map => takes a function & a list.

```
map = \f -> \xs -> if null xs then
                   []
                   else
                   f(head xs): map f (tail xs)
```

```
→ doublelist = \xs -> map (\n -> 2 * n) xs
```

```
fliplist = \bs -> map (\b -> not b) bs
```

(the above is the same / gives the same result as the fliplist on the previous page)

◇ MAP = Higher Order Function

Every function in Haskell takes in one function and gives back one result.

```
Doublelist (4 : 1 : 3 : [ ])
(\xs -> map (\n -> 2 * n ) xs) (4 : 1 : 3 : []
```

<u>Assignment</u>

```
$ submit -cs4620 a1.hs
```

Max 80 Char length.

CS4620 Monday 14th October 2013

plus = \m -> \n -> m + n

plus 3 4 => 7

Increment a number by 1

inc = \n -> 1 + n

inc 5 => 6

<u>or</u>

inc = plus 1

$\underbrace{\text{inc}}_{\downarrow}$ 5 => 6

plus 1 5 => 6

sum = \ns -> if null ns then

$\boxed{0}$

else

plus (head ns) (sum (tail ns))

<u>or</u> (head ns) $\boxed{+}$ (sum (tail ns))

and = \bs -> if null bs then

$\boxed{\text{True}}$

else

(head bs) $\boxed{\&\&}$ (and (tail bs))

or = \bs -> if null bs then

$\boxed{\text{FALSE}}$

else

(head bs) || (and tail bs))

Higher Order Function - FOLDR

foldr

```
sum  =  foldr  plus  0
and  =  foldr  (\b1 -> \b2 -> b1 && b2)
```

```
foldr  =  \f   ->   \z   ->  \xs   ->   . . .
           ↑         ↑          ↖
       function   zero (empty list)   list being
                                      processed
```

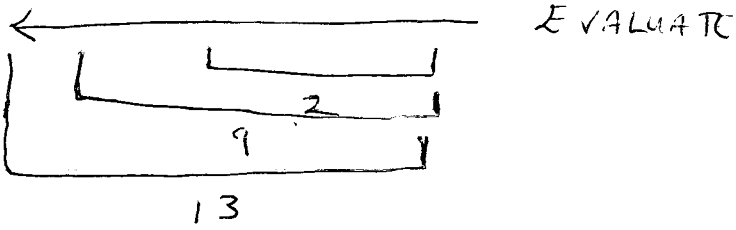
```
                                        (0)
. . .   if   null  xs   then  z
        else f  (head  xs) (foldr /f /z (tail xs))
          (plus)
```

```
foldr  =  \f  ->  \z  ->  \xs  ->
           if   null   xs   then z
           else  f (head  xs) (foldr  /f /z (tail xs))
```

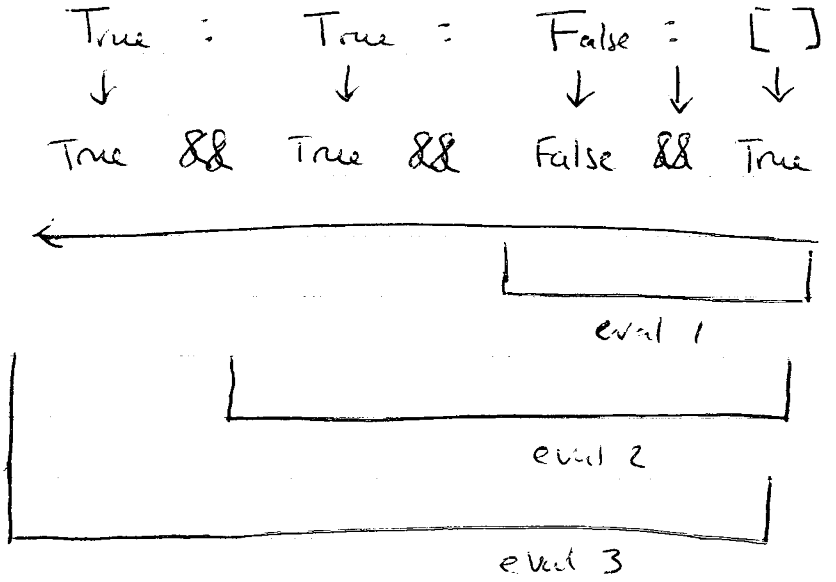
Example : sum (4 : 7 : 2 : [])

```
                          4 : 7 : 2 : []
 :  ->  +                 ↓   ↓   ↓   ↓
 [] ->  0                 4 + 7 + 2 + 0
```

folding the list = fold
from the right = r
foldr

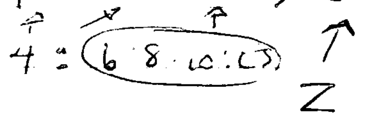


and (True : True : False : [])



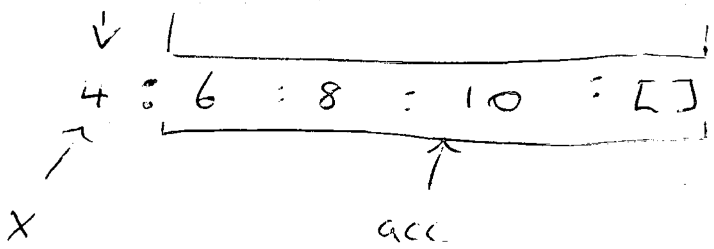
---

map = \f -> foldr (\x -> \acc -> f x : acc) []



<u>example</u>

doublelist : 1 : 2 : 3 : 4 : 5 : []



(acc = accumulator)

for wednesday : filter

CS4620 Wednesday 16th Septober 2013

$$\text{foldr} \quad \underset{\underset{\text{function}}{\nearrow}}{f} \quad \underset{\underset{\text{exp.}}{\uparrow}}{z} \quad \underset{\underset{\text{list}}{\uparrow}}{xs}$$

Collapses the list xs down to a single value.

[ ] = z

: = f

$$\text{sum} = \text{foldr}\ (\backslash n1 \to \backslash n2 \to \overbrace{n1 \underset{\underset{f}{\uparrow}}{+} n2})\ \underset{\underset{z}{\uparrow}}{0}$$

$$4 : 7 : 3 : 1 : 2 : [\,]$$

$$4 + 7 + 3 + 1 + 2 + 0$$

$$\xleftarrow{\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$\textcircled{17}$$

$$\underset{(\text{head})}{4} + \underset{\underset{(\text{tail})}{\boxed{13}}}{\underline{\qquad\qquad\qquad\qquad}}$$

parameter $\boxed{\begin{matrix}\text{head}\\ n1\end{matrix}} \begin{matrix}+\\+\end{matrix} \boxed{\begin{matrix}\text{tail}\\ n2\end{matrix}}$

```
if null xs then z
else f (head xs) (foldr f z (tail xs))
```

★ map = \f -> fold (\x -> \acc -> f x : acc) [

z
(if null xs)
then z.

map takes a function and a list

```
x
1 : 2 : 3 : 4 : []
×2   4 : 6   8 : []
          acc
2 : 4 : 6 : 8 : []
```

4 processed tail of list - acc

(p - parameter)
2 parameters
x + acc

◇ filte -> predicate applied to a list

★ filte = \p -> foldr (\x -> \acc -> if p x then x : acc else acc) [

```
x
3 : 4 : -2 : 6 : -3 : []
      4 : 6 : []
         acc
3 : 4 : 6 : []
```

filter

select positive numbers

if -3 = x
not acc to list &

if positive = IN
if not keep out

else acc - see func above

```
length ⇒ ~~7~~ 2 8 : 4 6 : []
         etc.  +2  +1  +0
   x
   ~~7~~ :    4 (acc)
   Head  (5)  tail
```

You can ignore the head value, just inc the acc by 1.

★ length = foldr (\x -> \acc -> 1 + acc) 0

~~DESIGN OF THE SYSTEM~~

CS4620 (16/10/13 continued)

◇ ALL takes a predicate (T or F) and a list, does every element of the list match it.

$$\text{all } (\backslash n \to n > 0)\ (1 : 2 : 3 : [\,]) \Rightarrow \underbrace{\text{TRUE}}_{\text{Answer}}$$

$$\text{all } (\backslash n \to n > 0)\ (1 : -2 : 3 : [\,]) \Rightarrow \textbf{FALSE}.$$

✱ $\text{all} = \backslash p \to \backslash xs \to \underbrace{\text{and (}}_{\text{can define}}\text{map } p \text{ } xs)$

<u>and</u> or <u>map</u> in terms of foldr.

This is ineffecient - checking all, we should just look for one FALSE and then we are done. <u>But</u> this function does this, it finds one that does not match and then it stops.

ANY

CS4620 Monday 21st October 2013

take has two parameters a number and a list

$$\text{take } 3 \; (5 : 7 : 4 : 6 : 8 : [\,]) \Rightarrow \underbrace{5 : 7 : 4 : [\,]}$$

~~(5 : 7 : 4 : [ ])~~

Gives back the first 3 elements of the list

$$\text{take } 0 \; (5 : 7 : 4 : 6 : 8 : [\,]) \Rightarrow [\,]$$

$$\text{take } 6 \; (\underbrace{5 : 7 : 4 : 6 : 8 : [\,]}_{\text{Only 5 Elements.}})$$

- * No facility for declaring error
- * When you define a function you can do what you wish.
- * CH gives you back only what can be given back i.e. 5 elements because there is no 6th element.

$$\text{take } -2 \; (\underbrace{5 : 7 : 4 : 6 : 8 : [\,]}_{\text{Gives back an empty list}})$$

because we have no function for error.

★ take definition (basic - ie no Higher order functions)

take <= 0 .. we return an empty list (Base case)

take 3 .. 5

$$5 : \underbrace{7 : 4 : 6 : 8}_{\text{take 2 of this list}} : [\,]$$

$\downarrow$

5 :

```
● take = \n -> \xs -> if n <= 0 then []
                      else head ns  take (n-1) (tail xs
```

Example

take 3 (4 : []) => 4 : []

Our function will take the head ie 4 plus take 2 of empty list.

because the list of numbers is only one long and we try to take two more we have an error, but because there are no errors in CH we need to add another base case.

```
take = \n -> \xs -> if n <= 0 then []
                    else if null xs then []
                    else head xs : take (n-1) (tai
                                                xs
```

more efficent:

```
★ take = \n -> \xs -> if n <= 0 || null xs then [
                      else head xs : take (n-1) (tail xs)
```

△ HIGHER ORDER FUNCTION for take

There is none!!! So you can't use a higher order function for take. This is because if you look at map, filter, foldr:

map f xs } see next page
filter p xs
foldr f z xs : this cannot be applied to an infinite list. WHY?

foldr goes to the end of the list add an z and work backwards.

if the list does not have an end, then fold will not even get started. So foldr cannot be used for take. Foldr lists must be finite.

map & filter - using map will be strange, map returns the same length list as the orignal.
filter selects some in the list so it is not practical.

take - works on infinite lists

DROP - opposite of take, take takes the certain numbers from a list and discards the rest, drop discards the certain numbers at the start and keeps the rest

```
drop 3 ( 7 : 1 : 4 : 6 : 2 : 9 : 8 : [] )
   => ( 6 : 2 : 9 : 8 : [] )

drop 20 ( . . . )
   => []

drop 0 => ( orignal list returned )

* drop = \n -> \xs -> if n <= 0 || null xs then x
                      else drop (n-1) (tail xs)
```

TAKE WHILE

takeWhile - takes two parameters, the first is not a number but a predicate (p) this time
takeWhile p xs

Keep gathering until they dont match the predicate p.

takeWhile (\n -> n > 0)(7 : 2 : 4 : -3 : 8 : 1 : [

All positive

p

list

xs

=> (~~7 : 2 : 4~~) 7 : 2 : 4 : [ ]

The list had a -3 so when this didnt match the predicate the function stopped, disregarding all numbers beyond the -3 also, i.e. 8 : 1.

DROPWHILE

dropWhile (\n -> n > 0)(7 : 2 : 4 : -3 : 8 : 1 : [ ])

All positive

p

list xs

xs

=>] - 3 : 8 : 1 : [ ]

as soon as the predicate was matched the numbers were dropped. when -3 was found it started returning the list.

takeWhile definition

```
takeWhile = \p -> \xs -> if not(p (head xs))
                         || null xs then []
                    else head xs : taikeWhile p (tail xs
```

include head

continue on through the tail until p is not met.

The above is not correct!!!
Supposing xs was empty, not (p (head xs) would be called before null xs. So what we need to write instead is ?

```
takeWhile = \p -> \xs -> if null xs // not (p (head xs)
                         then []
                         . . .
```

this way if the list is empty we never even need to look at the: not (p (head xs)), because this is <u>if x or y</u> then are false. and if the list is empty then we get a False so []

CS4620 Wednesday 23rd October 2013

J.Manning Office - 1.80

```
zipWith (\n1 -> \n2 -> n1 + n2)
        (3 : 7 : 4 : 2 : [ ]) (-5 : 6 : 4 : 0 [ ])
```

Run down two (both) lists in parallel

```
⇒ (-2  :  13  :  8  :  2  :  [ ])
    ↑      ↑     ↑     ↑      ↓
  3+-5  :  7+6 : 4+4 : 2+0  : [ ]
```

```
If we had (3 : 7 : 4 : [ ])
          (-5 : 6 : 4 : 0 : [ ])
```

we would stop and ignore

so -2 : 13 : 8

```
★ zipWith - \f -> \xs -> \ys -> if null xs || null ys
                                  then [ ]
                                  else
                                  f (head xs) (head ys)
                                  : zipWith f (tail xs)
                                              (tail ys)
```

Infinite lists are possible – haskell because it is lazy, it will not do anything unless it has to.

```
from = \n -> n   from (n+1)

from 1 = 1 : 2 : 3 : 4 : ...
```

(Ctrl C to stop this from running)

△ Fibonacci Numbers

0   1   ← first two numbers

0 1 1 2 3 5 8 13 . . .   sum of previous two numbers, so $0+1 = 1$

$1+1 = 2$

$1+2 = 3$

★ $$f(n) = \begin{cases} 0, & \text{if } n = 1 \\ 1, & \text{if } n = 2 \\ f(n-1) + f(n-2) & \text{if } n > 0 \end{cases}$$

0	1	1	2	3	5	8	13	. . .
f(1)	f(2)	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)	. .

```
○ f = \n ->
        if n == 1 then 0
        if n == 2 then 1
        else f (n-1) + f (n-2)
```

so if we type :   f(4)   we would get 2.

★ fibs = map f (from 1)

↑

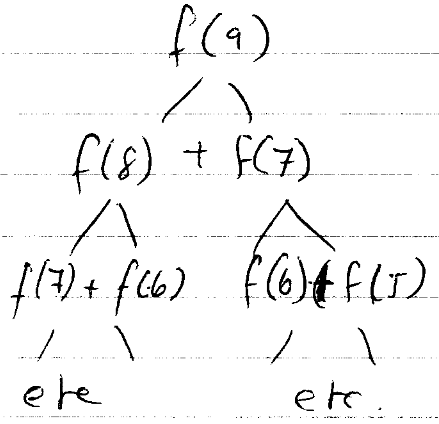
normally we would define a function i.e. \n ->
but here we are using a list ie. map.

fibs on its own would go on for ever, so instead
we type take 10 fibs (first 10 numbers)
or takeWhile (\x -> x <= 1000) fibs

↓

all numbers up to 1000 from fibs

But, with this method if we wanted f(9)
we would have to calculate all the numbers to get
there : ie



This is quite inefficient !!!

(\n -> n * n) (2 + 3)

↓ ↘ this was bound to 2+3 but now because it is bounded to 5 it no longer sees 2+3

2+3

↓

5 →

The Interpretter can remember things but not everything, remember in the early lectures!

Another case is :

big = 100 * 100

big + big

↓ So

100 * 100 *

↓ 10,000

10,000 → this now becomes big, i.e big = ~~100 * 100~~ 10000

We can not expect the interppretter to remember everything it has done (i.e. every evaluation).

CS4620 Wednesday 30th October 2013

fibs map f (from 1)

fibs =>	0	1	1	2	3	5	8	11
fibs tail =>	1	1	2	3	5	8	11	
01	1	2	3	5	8	13	19	

★ fibs = 0 : 1 : zipWith (\f1 ->\f2 -> f1 + f2)
fibs (tail fibs)

take 32 fibs - Map => Millions of evals (10 min)
- zipwith => 2 thousand evals (instant)

Generating Prime Numbers.

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

1-> divisible by 2 - cross out. - not prime.
2-> 2 is a prime ②, so is ③ so delete any number divisble by ③
3-> ⑤ likewise, ⑦ etc . . .
4-> What you are left with is 2, 3, 5, 7, 11, 13, 17, 19.
these are our prime numbers.

This method is called The Sieve of Eratostheaes

★ all one {

primes = sieve (from 2)

sieve = \ns -> head ns : dropMultiples (head ns)(tail ns *

dropMultiples = \d -> ~~\ns~~ -> filter (\n mod n d /=0) ~~ns~~

Sieve is a ~~f~~ very sophisticated filter

\ns = list of ns (ns = list of numbers)

head ns = first number in list - we keep this.

\d - takes a number from a

\ns - list of numbers

\filter - that numbers - keep if it gives you a non-zero remainder

* ⇒ all it will do at this stage is give us back
a list of 2 and every odd number after 2.
what about 3 5 7 etc. we need to include ↲

Sieve - where the ↙* is.
ie. sieve (dropMultiples (head ns)(tail ns)

(2) ie Clean Coder ?
(1) Clean Code by Robert C Martin

Refactoring - Martin ?

- Java, SQLServer, Oracle, Javascript.
- NoSQL new
- Apps -
- Thin Client - (web, intra/ext-net.)

The pragmatic Programmer
by David Thomas
Andy Hunt

- Agile Manefestor
  - Individuals & Interactions over process and tools.
  - Working Software over comprehensive docs
  - Customer Collaboration over contract negotiation.

Webx - share desktop.

Version Control

easy mock - mokito

- Agile Key Values
  - Communication (Scrum - standing meetings)
  - Simplicity (simple thing now than a couplex thing) adapt code over time - re-engineering.
  - Feedback (standup meetings etc)
  - Courage (regression tests - change code without fear)

Test Driven Development.
- Write a test
- See it fail
- Write production code to make it pass (+ maybe refactor)
- Refactor to remove duplication and to keep design simple.

CruiseControl Dashboard ⇒ Jenkins.
Sonar (plugins)

Specific by Example - Gojko Adzic
Growing O-O Software - Steve Freeman.
Concordia Acceptance TDD - www.concordie.org

Terry MacSweeney
www.fexco.com

tmacsweeney@fexco.com

Spring framework
hibernate "
* Jira - track requirement
Eclipse
* TDD ***

CS4620 Monday 5th November 2012.

fibs = 0 : 1 : zipWith (...) fibs (tail fibs)
fibs = head fibs : tail fibs.

The seconds one is true but computationally wrong / useless.

primes = ...
take 16 primes
2 : 3 : 5 : 7 : ... : 53 : []   = 9,371 evals

When you make a definition you dont to any computation, but once you use it the computation is stored so the next time the evals will be faster (i.e. take 16 primes is now = 583 evals)

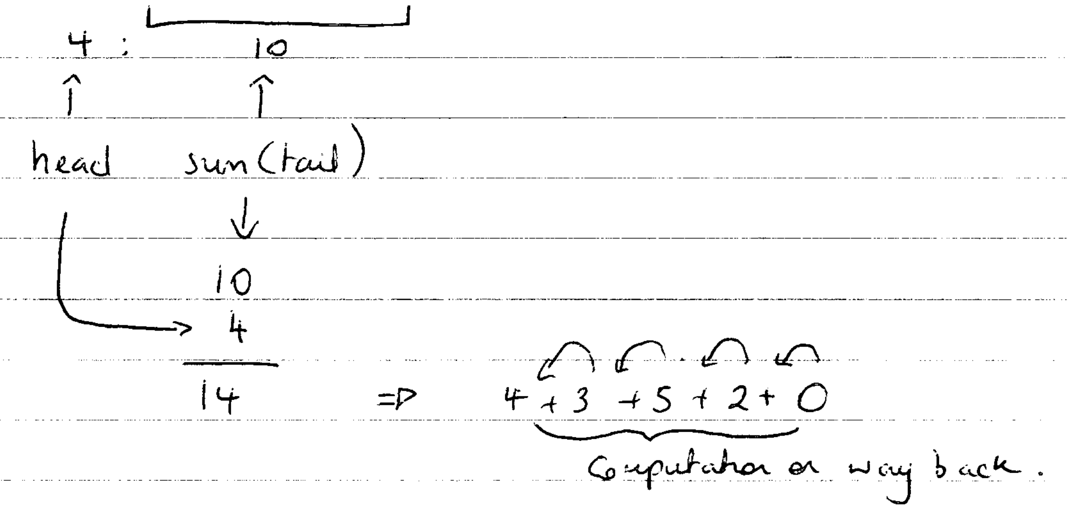
ie. primes = sieve (from 2)
~~as~~ when used = 2 : 3 : 5 ... : 53 : []
so the next time it is used instead of primes = sieve ..
it is equal = 2 : 3 : 5 .. : 53 : [] .

takeWhile (\p -> p <= 40) primes
head (drop 99 (primes))
head (drop while (\p -> p <= 1000) primes)

Accumulators

```
sum = \ns -> if null ns then 0 else head ns +
                                     sum (tail ns)
```

sum (4 : 3 : 5 : 2 : [ ]) => 14



Alternative version of sum:

```
* sum  = \ns -> sum' 0 ns
  sum' = \sumSoFar -> \ns ->
           if null ns then SumSoFar
         else sum'(sumSoFar + head(ns)) (tail ns)
```

' use ~~can~~ use a single quote as an identifier in CH

SumSoFar	ns
0	4 3 5 2
4	3 5 2
7	5 2
12	2
14	[ ]

when you get to the empty list the answer is what is in SumSoFar.

The purpose of this alternative is to add (sum) items in order of which they appear in the list, not in reverse as the orginal list (sum).

CS4620 Wednesday 6th November 2013

Helpers + Definitions - use clear comments in assignments.

★
```
sum = sum' 0
sum = \ns -> sum' 0 ns
```

ns not really needed and can be dropped hence the first definition.

```
a) sum' = \sumSoFar -> \ns
        -> if null ns then
           sumSoFar
           else sum' (sumSoFar + head ns) (tail ns)

b) sum = \ns -> if null ns then 0 else
                head ns + sum (tail ns)
```

Tail Recursion

△ MAXIMUM (OF A LIST)

maximum (5 : 2 : 7 : 1 : 8 : 3 : [])

● Another way to do fibs.

```
                        ------->
fibs = 0  1  1  2  3 |  5    8    13
                     |  f1   f2

                           ---------->
fibs = 0  1  1  2  3  5 |  8    13
                        |  f1   f2
```

The lot of fibonacci numbers starting at 5 and 8 to get 13 then the ... starting at 8 and 13 etc

This is recursive ; a recursive process.

13 = f1 + f2

★ fibs STARTING AT f1, f2 = f1 : fibs STARTING AT f2, f1 + f2.

▲ fibs' = \f1 -> \f2 -> f1 : fibs' f2 (f1 + f2)

The above is renamed to fibs' as its the helper!

Fibs is therefore =

● ▲ ★ fib = fibs' 0 1 $\boxed{\text{V.I.P. - Learn this}}$

take 16 fibs {
- zipWith : 1403 evals
- ACCUMULATORS : 750 evals