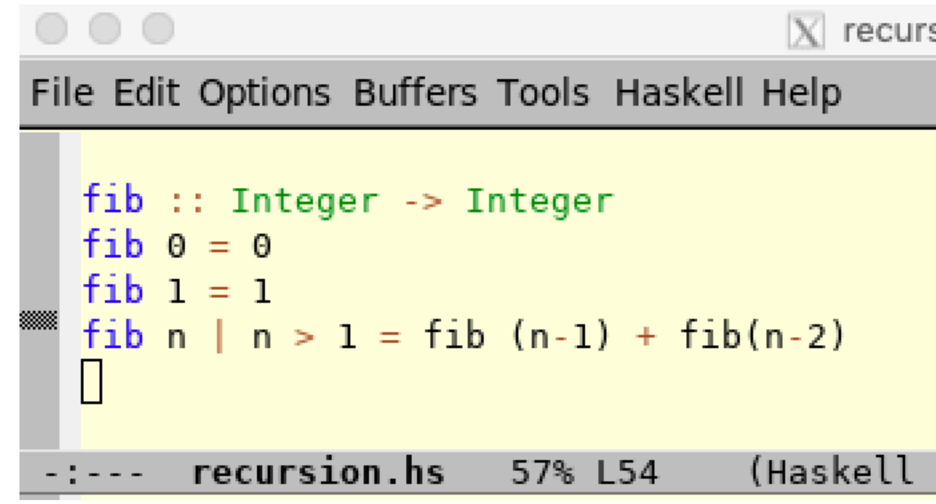
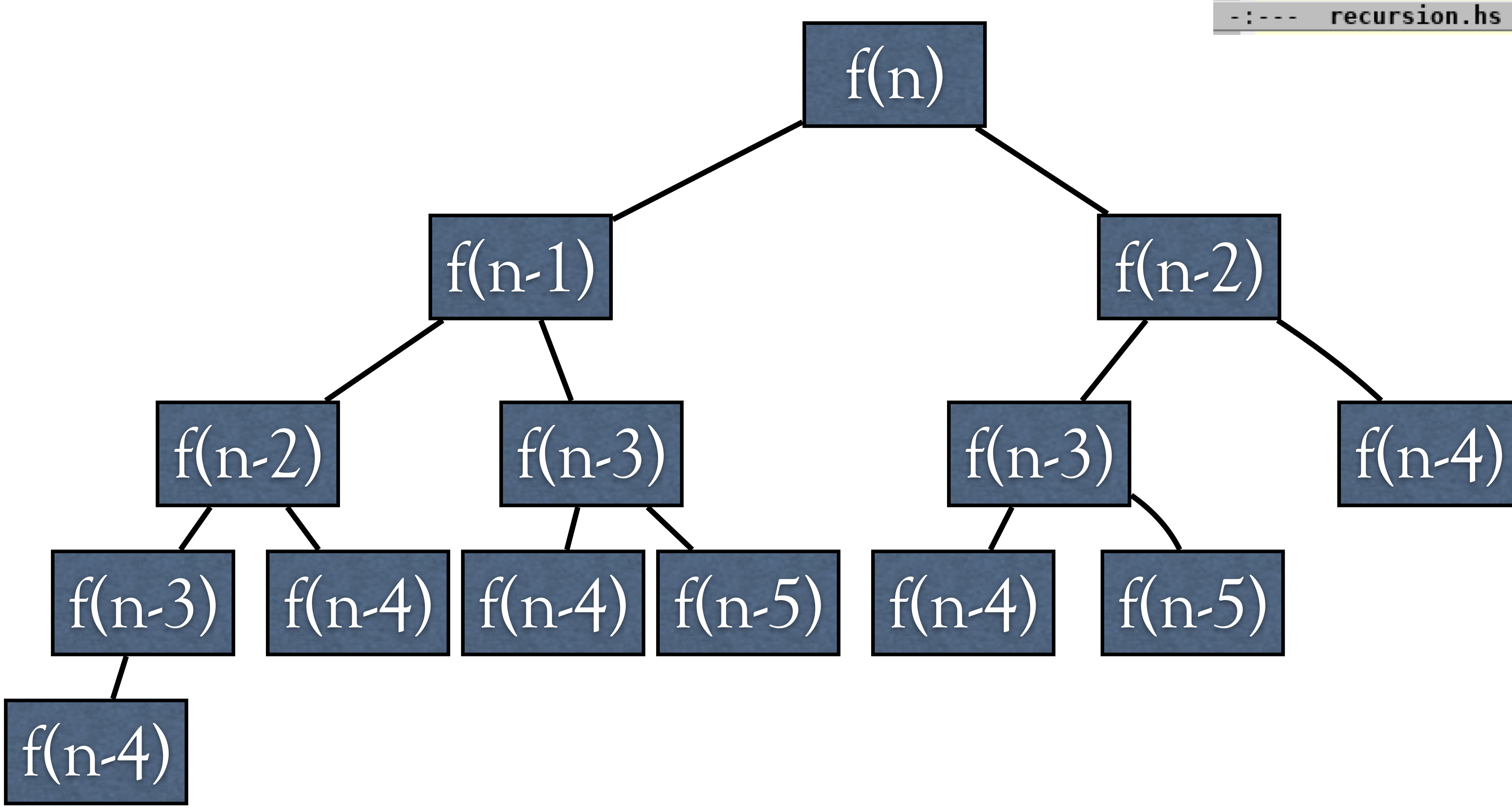
Recursion,
Lists

Fibonacci, again

- The straight forward implementation of the fibonacci sequence proved to be very slow, even for rather modest arguments.
- The computation will have a tree like structure
- Note the repeated nodes





Fibonacci, again

0, 1,

- Build sequence from the start and move along
  - Keep a tuple with the next two values
  - Start with (0, 1)
- When we move one step, add both values and shift the second value to the first
  - (k, m) => (m, m+k)

Note modest times!

recursion.hs

File Edit Options Buffers Tools Haskell Help

```haskell
afib :: Integer -> Integer
afib n | n >= 0 =
         let fib 0 (m, _) = m
             fib k (m, next) =
               fib (k-1) (next, m+next)
         in
           fib n (0, 1)
```

```
-:---  recursion.hs    63% L57    (Haskell Fixme ElDoc)
λ> fib 10
55
it :: Integer
(0.00 secs, 108,376 bytes)
λ> afib 10
55
it :: Integer
(0.00 secs, 73,784 bytes)
λ> fib 20
6765
it :: Integer
(0.29 secs, 4,841,080 bytes)
λ> afib 20
6765
it :: Integer
(0.04 secs, 78,776 bytes)
λ> fib 30
832040
it :: Integer
(1.68 secs, 586,745,720 bytes)
λ> afib 30
832040
it :: Integer
(0.00 secs, 83,768 bytes)
λ> afib 100
354224848179261915075
it :: Integer
(0.01 secs, 120,584 bytes)
```

Primes

- Determine whether a given natural number N is a prime number
- No immediate "smaller" problem in terms of N-1
- N is prime if it is *not* divisible by any of the numbers 2, 3, .. (N-1), (N-1)
- Solve problem of divisibility instead!
- Determine whether N has no dividers in the *range* K..M
  - Note: K is included in the range, but M is not

- Base case
  - range is empty, i.e., K == M
  - no dividers present
- Inductive case
  - decrease size of range, i.e., K+1..M, leads to recursive call
  - sub problem left:
    - check divisibility of N with K

Primes

```
-- indivisable low high n - True if there is no divider in the
-- interval low (included) to high - 1
indivisable :: Integer -> Integer -> Integer -> Bool
indivisable low high n | low == high = True
indivisable low high n =
  (mod n low /= 0)
  && indivisable (low + 1) high n

-- Is the argument a prime number?
isPrime :: Integer -> Bool
isPrime n =
   n > 1 && indivisable 2 n n
```

- The variant for `indivisible` is the *size* of the range, which decreases for each recursive call
- We test divisibility from the low end of the range
- Due to the lazy evaluation of `&&` we will stop as soon as we find a divisor
- We could do better
  - no need to test even numbers above 2
  - no need to test anything above square root of N
  - no need to test with anything but primes

Mutual Recursion

- We can have two, or more, functions that call each other recursively
- Indirect recursion - several function calls before calling the first function again
- Decide whether a natural number is even with only subtraction
- Base case
  - 0 is an even number
- Inductive case
  - N, N>0 is even if N-1 is odd
    - so we test for being odd as well

```
-- Mutual recursion
isEven :: Integer -> Bool
isEven 0 = True
isEven n = isOdd (n-1)

isOdd :: Integer -> Bool
isOdd 0 = False
isOdd n = isEven (n-1
```

Nested Recursion

- Recursion can take many forms
- It is perfectly ok to have a nested call, as in the third clause for `acker`
  - Note that `acker` grows very fast, `acker 4 2` has 19,729 digits
  - Ackermann function
- Does `collatz` terminate for every natural number?
  - hint: this is still undecided
  - Collatz Conjecture, 1937

```
-- Nested recursion
acker 0 n = n+1
acker m 0 = acker (m-1) 1
acker m n = acker (m-1) (acker m (n-1))

collatz 1 = 1
collatz n | mod n 2 == 0 = collatz (div n 2)
collatz n = collatz (3*n + 1)
```

Collecting Items

- Up to now we have used tuples to *group* related items
  - `("lecture", 8) :: (String, Integer)`
- We can write functions on tuples, e.g.,
  - `max :: (Integer, Integer) -> Integer`
- Tuples cannot collect an *arbitrary* number of items
  - Given N integers, we cannot write a function that computes the max of the N integers, without knowing N beforehand
- Tuples are used to group a *fixed* number of items of possibly different types
- It is easy to find situations where want to collect an arbitrary number of items
  - TODO list
  - shopping list
  - book collection
  - a list of TODO lists
  - birds seen
  - ...

Lists of items

- Haskell has the possibility to collect items of the *same type* into a *list*
- A list can be empty, i.e., contain no elements
- The *order* of elements is important
- The same element can occur multiple times
- Mathematically this is an *ordered sequence*, not a *set*
  - elements in a set have no order
  - elements in a set are unique
- We have already seen lists in disguise
  - A string is a list of characters
    - the order of characters is important
    - the same character can occur multiple times
  - A great deal of operations on strings are actually operations on lists

Lists

- Lists are written $[e_0,\ e_1,\ ..\ e_N]$
  - if all e_i :: <type>, then $[e_0,\ e_1,\ ..\ e_N]$ :: [<type>]
- Note type of string: `[Char]`
- `take` operates on lists
- a list of integers
- a list of strings is actually
  - a list of lists of characters
- we can have a list of functions
  - must be of same type
- enter a list of characters
  - prints as a string!
- What is the type of the empty list?
  - a list of *anything*
  - `[] :: [a]`
  - The empty list is *polymorphic*

```
-:---  recursion.hs    Bot L106    (Haskell Fixme ElDoc)
λ> "Lisp is another language"
"Lisp is another language"
it :: [Char]
λ> take 4 it
"Lisp"
it :: [Char]
λ> [2,3,5,7,11,13,17,19,23,29]
[2,3,5,7,11,13,17,19,23,29]
it :: Num a => [a]
λ> ["Tea", "Biscuits", "Fish"]
["Tea","Biscuits","Fish"]
it :: [[Char]]
λ> [True, True, False]
[True,True,False]
it :: [Bool]
λ> [isEven, isPrime]
[isEven, isPrime] :: [Integer -> Bool]
λ> [fac, fib, collatz]
[fac, fib, collatz] :: [Integer -> Integer]
λ> ['f', 'o', 'o']
"foo"
it :: [Char]
λ> []
[]
it :: [a]
λ> 
-:**-  *haskell*       Bot L546    (Interactive-Haskell Fixme)
Mark set
```

List Construction

- When Haskell sees `[e0, e1, .. eN]` ($[e_0, e_1, .. e_N]$)
  - each element is evaluated and a list is constructed
- `[]` constructs the empty list
- `x : xs` constructs a new list from an element and an existing list
  - `x` is called the head of the list
  - `xs` (a list) is called the tail of the list
  - The type of x must be of the same type as the elements of `xs`
- $[e_0, e_1, .. e_N]$ is shorthand for
  - $e_0 : e_1 : .. e_N : []$
- Lists can be compared for equality with `==`
  - Two lists are equal if they contain the same elements in the same order
  - As the empty list can "contain" elements of any type it can be compared for equality with any list
  - Lists of different types cannot be compared for equality
- Note the similarity with a recursive function; base and inductive case

Deconstructing Lists

- We can take a list part with two convenient functions
  - `head :: [a] -> a`
    - fails for the empty list
  - `tail :: [a] -> [a]`
    - fails for the empty list
- These are functions on *polymorphic* lists - they don't care about the type of the contents of the list
- Note symmetry with `x : xs` (`x` is the head and `xs` is the tail)
- Note different types of `head` and `tail`

```
-:---  recursion.hs   62% L94    (Haskell Fixme
λ> tail [4711, 42, 19]
[42,19]
it :: Num a => [a]
λ> head [4711, 42, 19]
4711
it :: Num a => a
λ> tail [4711, 42, 19]
[42,19]
it :: Num a => [a]
λ> tail []
*** Exception: Prelude.tail: empty list
λ> head "ship"
's'
it :: Char
λ> tail "ship"
"hip"
it :: [Char]
λ> head ["tall", "ships", "race"]
"tall"
it :: [Char]
λ> tail ["tall", "ships", "race"]
["ships","race"]
it :: [[Char]]
λ> 
-:**-  *haskell*      Bot L586   (Interactive-H
Mark set
```

Functions on Lists

- A list has two constructors: `[]` and `:`
  - Also, the shorthand `[x, y, z, ..]`
- We can take lists apart with `head` and `tail`
- This is all you need to work with lists
- As for tuples, we can use constructors in patterns to take lists apart
  - in practice that means we use pattern matching much more than `head` and `tail`
  - we can define `head` and `tail` using pattern matching

Functions on Lists

Determine whether a list is empty

```haskell
-- With comparison
isEmpty :: [a] -> Bool
isEmpty list = list == []
```

```haskell
-- With pattern matching
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty  _ = False
```

Extract parts of a list

```haskell
-- Obtain head of list
head :: [a] -> a
head (x:_) = x
```

```haskell
-- Obtain tail of list
tail :: [a] -> a
tail (_:xs) = xs
```

Length of List

- How can we compute the length of a list?
- We can only extract one list element at a time
  - this is similar to only being able to subtract one
- Base case: empty list
  - length is zero
- Inductive case
  - compute length of tail of list
  - add 1
- Note the similarity to recursion of integers, subtracting one in each step
- The difference is that we are doing on a *structure*

```
-- Length of list
length :: [a] -> Integer
length [] = 0
length (_:xs) = 1 + length xs
```

Get Last Element

- We can get the first element directly using `head` or pattern matching
- How can we get the last element?
  - We have to move down the list removing one element at a time
  - The empty list has no last (or first) element
- Base case: list of one element
  - the single element is what we want!
- Inductive case: list with more than one element
  - find last element of tail

```
-- Last element of list
last :: [a] -> a
last [x] = x
last (_:xs) = last xs
```

Take first N elements

- Given a list, return first N elements (or all of the list if shorter than N)
- We have two base cases
  - N is zero - no elements to take
    - return empty list
  - list is empty - no elements to take
    - return empty list
- Inductive case
  - Take N-1 elements from tail of list
  - Add head with list constructor

```
-- Take first N elements of list
take :: Integer -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

Add two lists

- Given two lists [1,2,3] and [4,5,6], how can we construct the list [1,2,3,4,5,6]?
- Again, we can only remove and add one element at a time.
- Base case: first list is empty
  - result is second list
- Inductive case: first list is non empty
  - add tail of first list with second list
  - add head to the result

```
-- Add two lists together
append :: [a] -> [a] -> [a]
append [] list = list
append (x:xs) list =
  x : append xs list
```

Existing List Functions

- Understanding simple list functions is key to understanding more general recursive functions
- There are a lot of simple list functions already defined, but writing and understanding them gives a great deal of value
- Good exercise to define them by yourself

```
null :: [a] -> Bool
head :: [a] -> a
tail :: [a] -> [a]
init :: [a] -> [a]
last :: [a] -> a
length :: [a] -> Int
```

```
reverse :: [a] -> [a]
++ :: [a] -> [a] -> [a]
elem :: a -> [a] -> Bool
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
replicate :: Int -> a -> [a]
```