Functional techniques:

recursion on inductive data (lists, blists, trees, etc.), implicit state, accumulators, converting general recursion to iteration, invariants, higher-order functions

First of all, here is is some general data type in the haskell, some of them probably contain the same type of data, or empty.

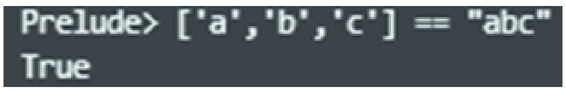
A list of numbers:

[1,2,3,4]

A list of characters:

[‘a’,’b’,’c,’]

By the way, in Haskell, a string is a list of characters.



```
Prelude> ['a','b','c'] == "abc"
True
```

Empty list:

[ ]

In haskell, we always write like (h:t) to present a list like this. For me, ‘h’ is like the first element of the list, and t is representing the rest of those.

For example L=[3,5,6], (h:t)= L, then h in this list would represent number ‘3’, and t will represent 5 and 6. we will write like [5,6].

This is very useful for people who start learning about Haskell and oz language.

Implicit:

The literal meaning is that some details are not explained, but there is no need to explain, because there is enough information available to analyze the collected information.
For example: When using Twitter in daily life, it usually contains obvious and hidden data, and will push some related tweets according to your preferences when you read or watch. In programming, these words are usually used to describe how variables are declared.

How to achieve recursion? We need to turn a big problem into many small problems and solve them one by one.
example:

```
            length[]= 0
            length(x:xs) = 1+ length xs;
```

xs is going to do the same thing for the rest numbers,   x: take one element out

```
 or we can do it like : length [1,2,3,4] -> 1+length [2,3,4]
                        length [2,3,4] -> 1+length[3,4]
                        length[ 3,4] -> 1+ length[4]
                       length[4] -> 1+length[]
                       length[] -> 0
```

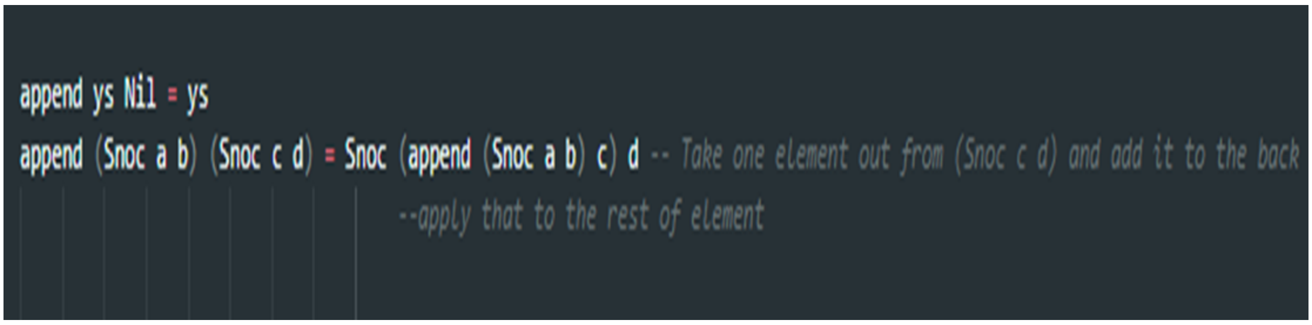
now, we can try to appending two lists:

```
append[] ys= ys    (here is base case)
append(x:xs) ys= x: append xs ys
```

(x:xs) means we take x or one element out of the list xs
x:append xs ys means: we take x in the front of the second list, then apply it to the rest of the element.
for example:

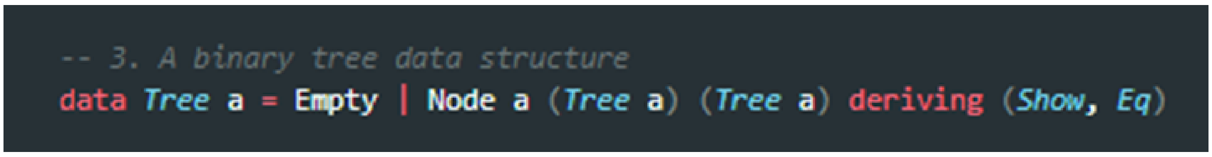
append[1,2] [3,4]   1: append [2] [3,4] -> 1:[2,3,4] -> [1,2,3,4]
append [2] [3,4]   2:append[ ] [3,4] -> [2] [3,4] -> [2,3,4]

in Blist: we always use “cons” and “snoc” as the opposite
BList adds new elements from behind it.
In BList, we use snoc (the reverse version of cons) instead of cons. so,
In BList, [1,2,3] = Snoc(Snoc(Snoc Nil 1) 2) 3



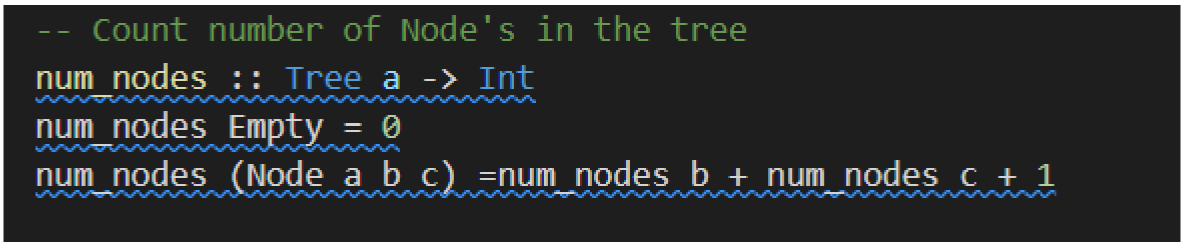
```haskell
append ys Nil = ys
append (Snoc a b) (Snoc c d) = Snoc (append (Snoc a b) c) d -- Take one element out from (Snoc c d) and add it to the back
                                                            --apply that to the rest of element
```

Okay! let’s define a tree in haskell:



```haskell
-- 3. A binary tree data structure
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Eq)
```

How do we count how many nodes are in a tree? This problem is somewhat the same as the length of the list. When there is nothing in the tree, we get no nodes. This is the basic situation. When the tree is not empty, then every time we pass a node, we will trying to add 1 to the result:

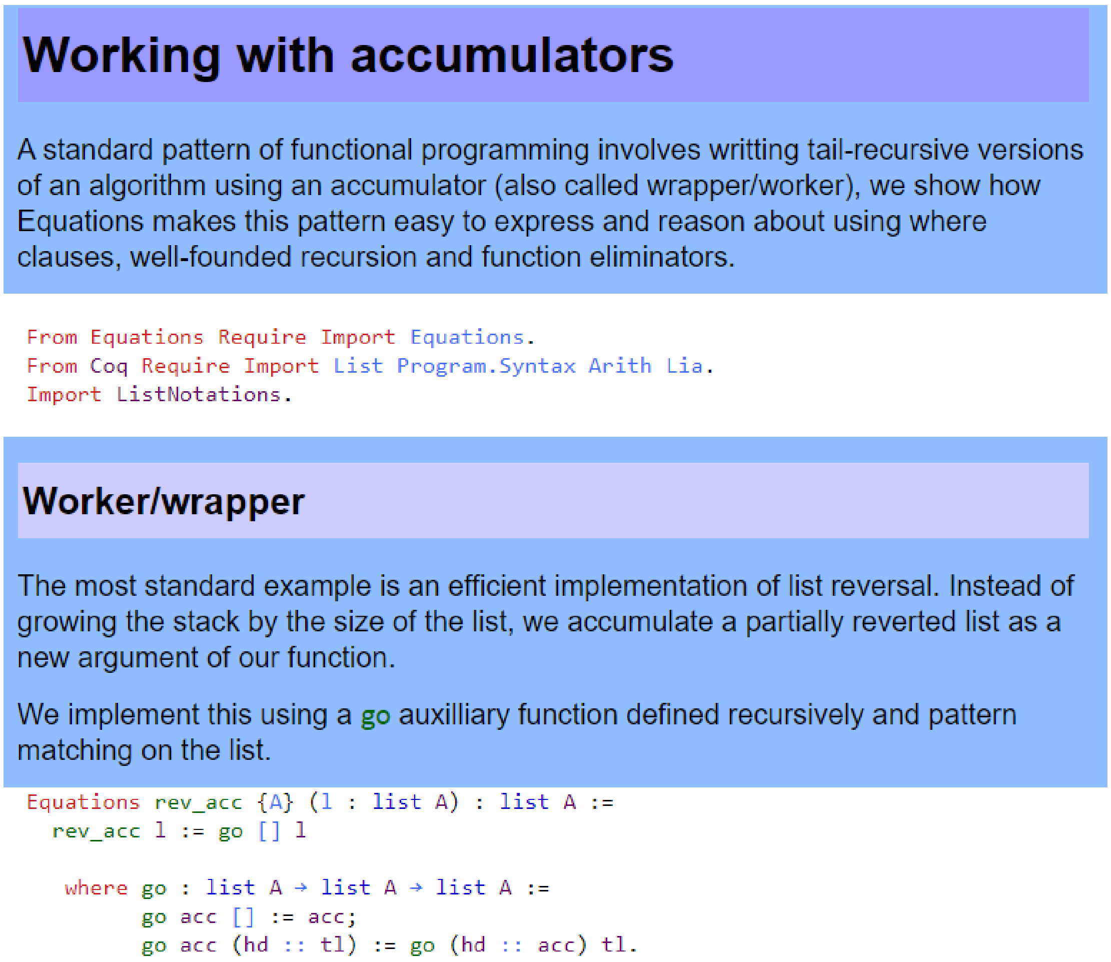


```haskell
-- Count number of Node's in the tree
num_nodes :: Tree a -> Int
num_nodes Empty = 0
num_nodes (Node a b c) =num_nodes b + num_nodes c + 1
```

Accumulator is used to the value in the accumulator with each digit, just like what happens in a simple desktop calculator, but it’s more efficient.

here is a example :

len [1,2,3]	0	
len [2,3]	0+1	
len [3]	0+1+1	
len[ ]	0+1+1+1	so the length is 3



Working with accumulators

A standard pattern of functional programming involves writting tail-recursive versions of an algorithm using an accumulator (also called wrapper/worker), we show how Equations makes this pattern easy to express and reason about using where clauses, well-founded recursion and function eliminators.

```
From Equations Require Import Equations.
From Coq Require Import List Program.Syntax Arith Lia.
Import ListNotations.
```

Worker/wrapper

The most standard example is an efficient implementation of list reversal. Instead of growing the stack by the size of the list, we accumulate a partially reverted list as a new argument of our function.

We implement this using a go auxilliary function defined recursively and pattern matching on the list.

```
Equations rev_acc {A} (l : list A) : list A :=
  rev_acc l := go [] l

   where go : list A → list A → list A :=
         go acc [] := acc;
         go acc (hd :: tl) := go (hd :: acc) tl.
```

When using an accumulator, it’s more efficient to calculate the recursion, we don't need to jump back.

Regarding converting general recursion to iteration:

How do we convert a general recursion to iteration? regarding the idea of accumulator and the tail call, it’s easy. the only thing we need to do is move the operation from the end of the function to accumulator then convert it as a helper function.

let’s make a example:

```
len :: [a] -> int -> int
len [] = 0
len(x:xs)  = 1 + len xs
```

then how do we convert it to the iteration?
read the sentences above
we can use accumulator’s logic, then convert it as a helper

```
helper [ ] a = a
helper (x:xs) a = helper xs (a+1)
```

But, if you know tail recursion, you probably think helper function is kind of the same with it.

```
len [ ]a = a
len (x:xs) a = len xs (a+1)
```

Ok, let's keep working on helper function
we can call the helper function and initialize the accumulator.

```
len xs = helper xs 0 where
        helper [ ] a = 0
        helper (x:xs) a = helper xs (a+1)
```

Thats it !

Three steps of iteration:

step1. Determine the base case, then helper function should return the accumulator to the base case,

step2. Do calculate in the accumulator.

step3. call your helper function in the main function with the proper initial value.

People may ask why we should convert recursion to iteration?

invariants
base case:
length of [] is 0, true.
then we can use :

Length of (x:xs) = 1 + length of xs or we can say

    len [x] + len xs = 1 + len xs

    we know that length of [x] = 1

    len [x] + len xs = 1 + len xs

Invariant is always true in loop or recursion, no matter what. It used to make sure the cycle continues correctly, like an induction in math.

let’s make some examples to prove it.
1:
length of [ ] is 0. always true
length (x:xs) = 1+ length of xs
or
length [x] + length xs = 1 + length xs

Higher order functions

It’s a function that take another function as an argument and return it as a result. we call it higher-order function.

Take the max function as an example. It needs two parameters to return and get a larger one. So use max 4 5 as an example to complete

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Like we write C++, remember to use spaces before writing a parameter.
Now let's look at max. According to the following example, it can be understood as: max accepts a function of a, returns, returns, and returns. For us, the arrow is used to better understand the meaning of function.
If we use too few functions, we can't get the specific meaning, which means we can only use part of the function

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

lets do `multThree 3 5 9` or `((multThree 3) 5) 9`

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

Generally speaking, higher-order function is used to make its arguments accept other functions.

Part 2 Oz syntactic sugar:

First of all, we need to know what is the thread in Oz language?

The threads are like dataflow variables and declarative concurrency, like we need to write a thread then do the definition.

example:

```
thread
   Z = X+Y
   {Browse Z}
end
thread X = 40 end
thread Y = 2 end
```

syntax and semantics

Syntax is like the structure, how you want to write this program easuky. semantics is the meaning of the code. we can say like: If B is bound to true, S2 is executed

For semantics we can use the if statement.

For example:

```
local X Y Z in
   X = 5 Y = 10
   if X >= Y then Z = X else Z = Y end
end
```

It’s very familiarly with the C++.

Also, we can use other example to make you guys understand

```
declare
fun lazy {Merge Xs Ys}
   case Xs#Ys
   of (X|Xr)#(Y|Yr) then
      if X < Y then X|{Merge Xr Ys}
      elseif X>Y then Y|{Merge Xs Yr}
      else X|{Merge Xr Yr}
      end
   end
end

fun lazy {Times N Xs}
   case Xs
   of nil then nil
   [] X|Xr then N*X|{Times N Xr}
   end
end

declare H
H = 1 | {Merge {Times 2 H} {Merge {Times 3 H} {Times 5 H}}}
{Browse {List.take H 6}}
```

In the above example, we used if elseif statement, to make our syntax and semantic more easier to understand. if X smaller than Y, then X merge Xr Ys. else if X>Y, we do the opposite thing , merge Y. they are kind same like the syntax in C++.

Scheduling

Oz language is the first high-level constraint language, because it provided a interface, so we can use the way like C++ to create a new constraint.

for the storing of constraint included record, numbers, name and variable.

For example, we make a constraint stores:

X1= U1… Xn=Un

for here, X1 are variables, and U1 are Oz entities or variables totally speaking, constraint is a relationship between variables, we can use it to limit the variables. kind of same like other languages.

```
V1  < V2
V2  < V3
   ...
V14 < V15
```

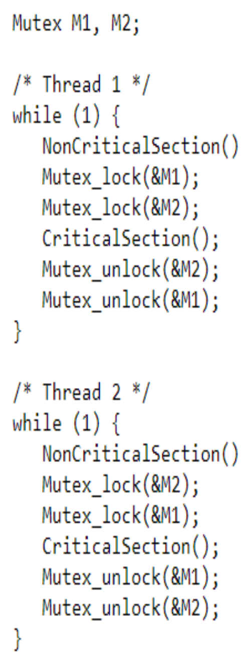
It's like a math constraint, we only used < to constraint those variables.

```
proc {Ints N Xs}
   or N = 0 Xs = nil
   [] Xr in
      N > 0 = true Xs = N|Xr
      {Ints N-1 Xr}
   end
end
local
   proc {Sum3 Xs N R}
      or Xs = nil R = N
      [] X|Xr = Xs in
         {Sum3 Xr X+N R}
      end
   end
in proc {Sum Xs R} {Sum3 Xs 0 R} end
end
local N S R in
   thread {Ints N S} end
   thread {Sum S {Browse}} end
   N = 1000
end
```

For this one, it's kind of harder, because the thread executed Ints, until N is known, itherwise it cannot decide on which disjunct to choose. Also, sum3 is waiting for sum to be declared.
This shows super clear that how constraint works

Deadlock
First of all, deadlock is a situation when a set of threads are blocked, because there are other threads holding and waiting for another one. For

example, in real life, we need to take a flight to another country, but there are other planes waiting to take off. Which means we need to wait until the flight in front of us takes off, then we can take off.
here is an example of deadlock



```
Mutex M1, M2;

/* Thread 1 */
while (1) {
   NonCriticalSection()
   Mutex_lock(&M1);
   Mutex_lock(&M2);
   CriticalSection();
   Mutex_unlock(&M2);
   Mutex_unlock(&M1);
}

/* Thread 2 */
while (1) {
   NonCriticalSection()
   Mutex_lock(&M2);
   Mutex_lock(&M1);
   CriticalSection();
   Mutex_unlock(&M1);
   Mutex_unlock(&M2);
}
```

let's say thread 1 is running, so it needs to lock M1, but it is interrupted before it can lock M2. when thread 2 starts running , it locks M2, when it tries to lock M1. So, it is blocked, because M1 already locked by thread1.

Determinism

It's a algorithm, when given a particular input, it will output the same result

```
proc {MMerge STs L}
   C = {NewCell L}
   proc {MM STs S E}
      case STs
      of ST|STr then M in
         thread
            {ForAll ST proc{$ X} ST1 in {Exchange C X|ST1 ST1} end}
            M=S
         end
         {MM STr M E}
      [] nil then skip
      [] merge(STs1 STs2) then M in
         thread {MM STs1 S M} end
         {MM STs2 M E}
      end
   end
   E
in
   thread {MM  STs unit E} end
   thread if E==unit then L = nil end end
end
```

For here, we can simply explain like {MMerge [X Y] Z}. Because we made a definition about this. So no matter what, the program will keep work like this.

Or we can do it like:

```
proc {Merge Xs Ys Zs}
   cond
      Xs = nil then Zs = Ys
   [] Ys = nil then Zs = Xr
   [] X Xr in Xs = X|Xr then Zr in
      Zs = X|Zr {Merge Xr Ys Zr}
   [] Y Yr in Ys = Y|Yr then Zr in
      Zs = Y|Zr {Merge Xs Yr Zr}
   end
end
```

Also, there are a lot of non-deterministic, like prolog, but a little different. For Oz language, there is a specified structure, we specify the search strategy.
dis construct.

```
proc {Append Xs Ys Zs}
   dis
      Xs = nil Ys = Zs then skip
   [] X Xr Zr in
      Xs = X|Xr Zs = X|Zr then
      {Append Xr Ys Zr}
   end
end
```

Here is the example in Oz mode. We can use the dis construct to create a choice point when we need it.

Also, there is one example for Prolog

```
append([], Ys, Ys).
append([X|Xr], Ys, [X|Zr]) :- append(Xr, Yr, Zr).
```