**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.



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.

the key to implement recursion: find out the calculation, then repeat, until you find the result, then, use the method:divide and conquer. divide the big question into smaller, then solver it gradually, make it becomes the base case

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:

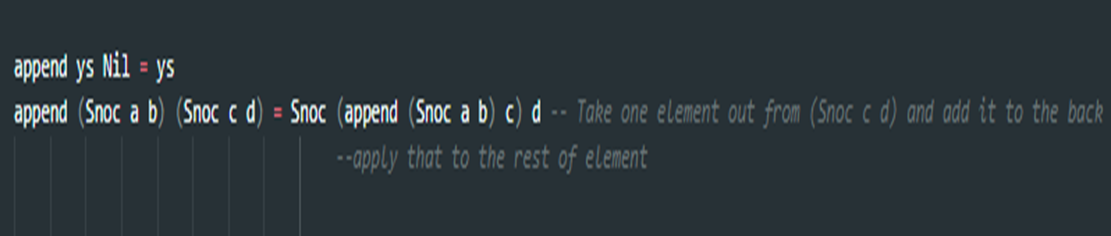
append[1,2] [3,4] 1: append [2] [3,4] -> 1:[2,3,4] -> [1,2,3,4]

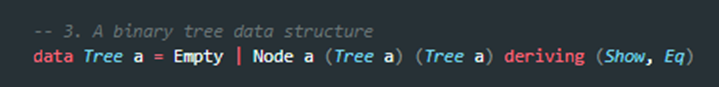
append [2] [3,4] 2:append[ ] [3,4] -> [2] [3,4] -> [2,3,4]

in Blist: we always use “cons” and “snoc” as the opposite

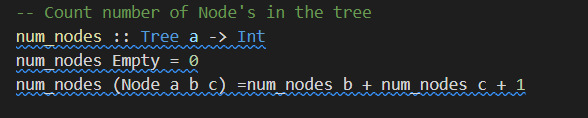
BList adds new elements from behind it.

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



Okay! let’s define a tree in haskell:

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:



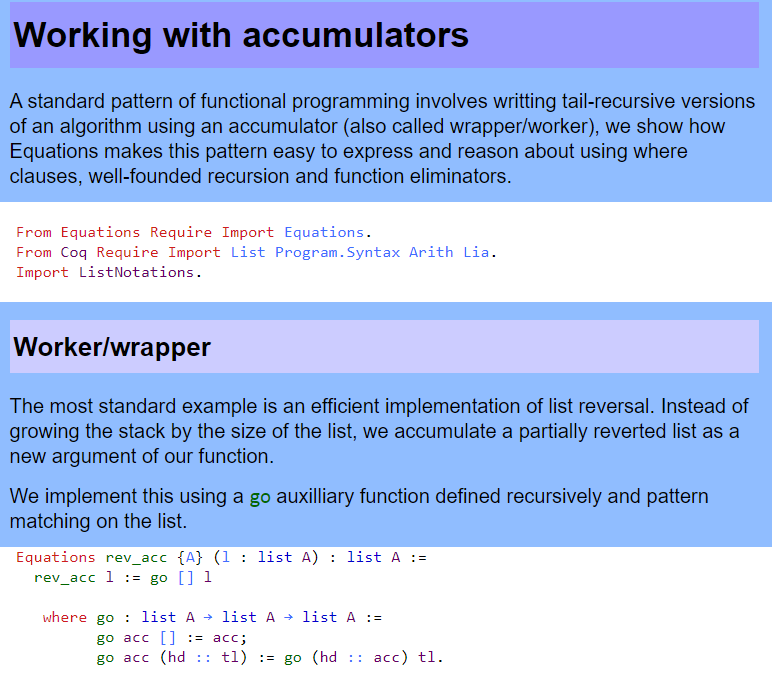
Accumulator is used to the value in the accumulator with each digit, just like what happens in a simple desktop calculator, but it’s more efficient.

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



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.

Let's take our good friend, the **max** function. It looks like it takes two parameters and returns the one that's bigger. Doing **max 4 5** first creates a function that takes a parameter and returns either **4** or that parameter, depending on which is bigger. Then, **5** is applied to that function and that function produces our desired result. That sounds like a mouthful but it's actually a really cool concept. The following two calls are equivalent:

ghci> max 4 5

5

ghci> (max 4) 5

5

Putting a space between two things is simply **function application**. The space is sort of like an operator and it has the highest precedence. Let's examine the type of **max**. It's **max :: (Ord a) => a -> a -> a**. That can also be written as **max :: (Ord a) => a -> (a -> a)**. That could be read as: **max** takes an **a** and returns (that's the **->**) a function that takes an **a** and returns an **a**. That's why the return type and the parameters of functions are all simply separated with arrows.

Then, if we call a function with too few parameters, we get back a **partially applied** function, meaning a function that takes as many parameters as we left out. Using partial applications (calling functions with too few parameters, if you will) is a neat way to create functions on the fly so we can pass them to another function or to seed them with some data.

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**

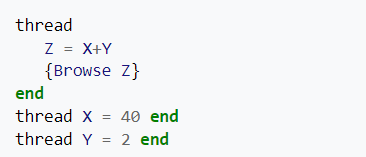
Generally speaking, higher-order function is used to make its arguments accept other functions.

**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:

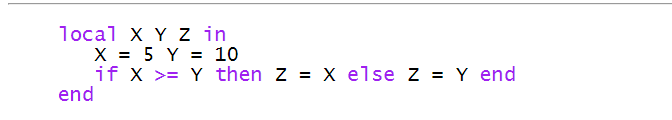


**syntax and semantics**

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

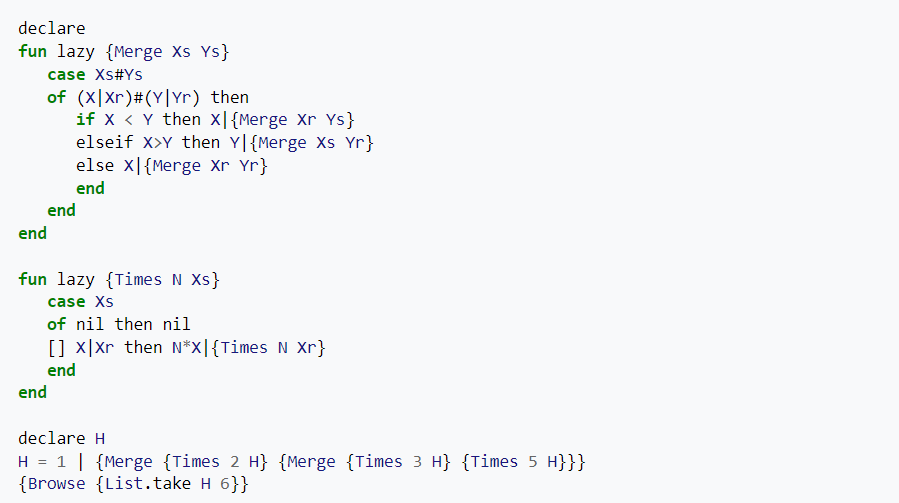
For semantics we can use the if statement.

For example:



It’s very familiarly with the C++.

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



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:

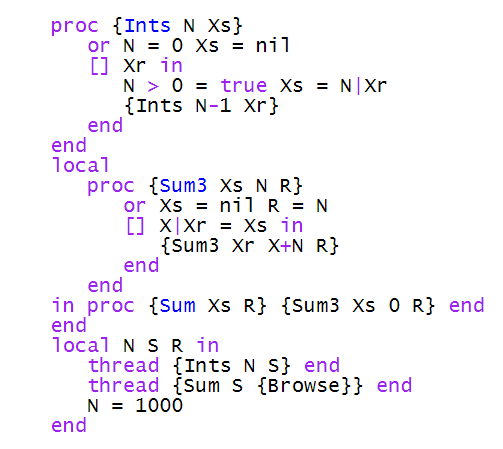
X1= U1… Xn=Un

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.



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



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.

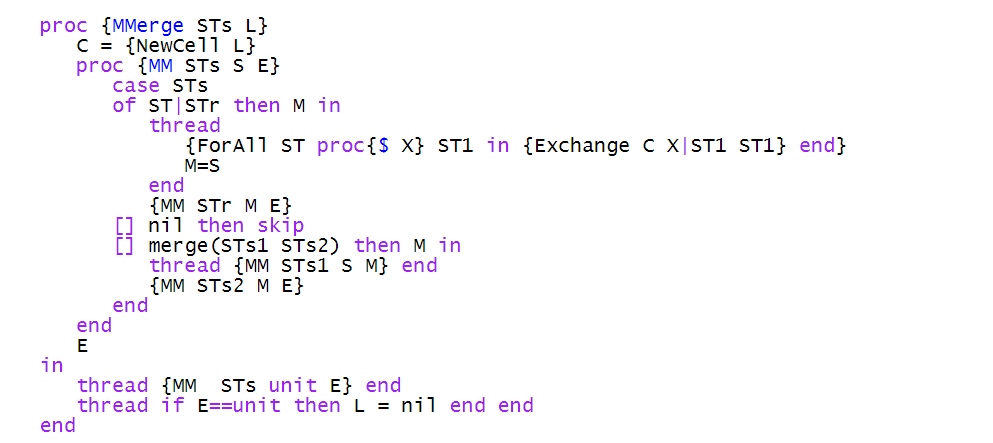
here is an example of deadlock



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.

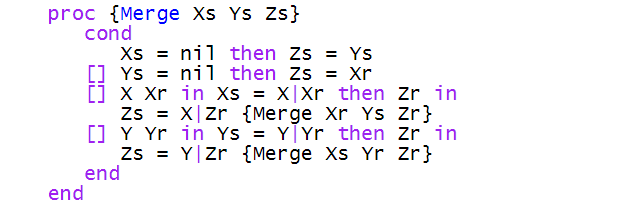
**Determinism**

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



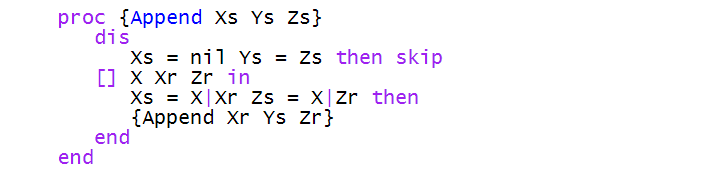
For here, we can simply explain like {MMerge [ X Y] Z}. Because we made a definition about this. So no matter what, the program will keep work like this.

Or we can do it like:



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.



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

