**Functional techniques**:

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

List is a general data type in the Haskell, it can contain the same type of data (numbers or characters), or it is empty.

A list of numbers:



A list of characters:



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



Empty list:



In Haskell, we can write list in such way (h:t) to present a list. The **‘h’** is representing the first element in the list, and **‘t’** is representing the rest of elements in the list. For example, let’s say

l = [1,2,3,4], if we write (h:t) = l, then h is going to be number 1, and t is going to be a list that contains rest of numbers [2,3,4].

Text

Description automatically generated

That is very useful when we want to write some operators for the list, such as length or reverse or append…. etc.

One way we use to implement those operators is recursion.

To determine whether recursion is a good idea, you first must decide if you can divide the question into many small parts and imply the same calculation on each of them.

For example, if we try to find a length of a list. What we will do is we go through the list, starting with 0 and add 1 to the result every time when we meet an element. In this case, we apply the same calculation on every element in the list, which is length of list + 1.

Therefore, we can divide the list by the elements in it, and then apply length of list +1 on each of them. Then, we must determine where to stop our calculation.

Absolutely, we should stop when we got to the end of the list. That is when there is no element in the list. (An empty list.). It also called base case. When it comes to the base case, there is not any calculation need, we can get the result directly.

Therefore, the points to implement recursion is, find out the calculation that we need to do again and again before we got the result. Then, find the base case and slice the questions, make it to be smaller and smaller until it becomes the base case.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Now, let’s try appending two lists. Because list adding new element from its from, so when the list in front is empty, we are done. So, the base case is whenever the list in front is empty, then we return the second list. The calculation we need to do is, we go through the first list, and every time when we see an element, we put it in front of the second list. So, we can divide the first list by its element, and apply the calculation to every element, until there is no more element in the first list.

Text, letter

Description automatically generated

A picture containing text, whiteboard

Description automatically generated

Not only list, in Haskell, users even allow to define their own data type.

We know normal list add new element from front

[1,2,3] = 1: [2,3] = 1: (2: [3]) = 1: (2: (3: []))

A picture containing text, whiteboard

Description automatically generated

We call “: “cons and “[]” nil, now there is a data type that opposite to list, which is BList.

As its name, BList adds new element from its back.

In the BList, we use snoc (the reverse version of cons) instead of cons. Therefore,

In the BList, [1,2,3] = Snoc (Snoc (Snoc Nil 1) 2) 3

A picture containing text, whiteboard

Description automatically generated

Differences

A picture containing text, whiteboard

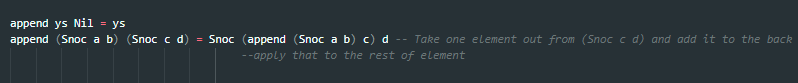
Description automatically generated

Now, how to implement length function on BList by using recursion?

First, we need to determine what is the base case. Not like list, BList adding new element from it’s back. Therefore, its base case should be when the second list is empty, then we are done.

So, every time we take one element out from the second list, and add it to the back, until the second list is empty.

That is the code



What about length? The same idea as list. Base case is when that is Nil, then the length is zero, we are done. When the list is not empty, every time we take one element out and add 1 to the l to the result, until the list is empty.

A picture containing text, whiteboard

Description automatically generated

There is one more data type, we call tree. People use it very often in the programming.

That is how a tree looks like. It starts with a node, and it can be only a node on the side

(We call leaf) and it can be another tree on the side (we call subtree).

Tree

Subtree Leaf

In Haskell, we define a tree as

A screenshot of a computer

Description automatically generated with medium confidence

Here is the question, how do we count how many nodes are there in a tree? The question is kind of the same as the length of list. When there is nothing in a tree, we get 0 node, then we are done, that is the base case. When the tree is not empty, then every time, when we travel through a node, we add 1 to the result. Therefore

Text

Description automatically generated

What if we want to get the sum of all nodes? First, we need to determine the base case. Absolutely, when the tree is empty, the sum will be 0, and then we are done. If the tree is not empty, then we add the number to the result every time we travel through a node. Therefore

Text

Description automatically generated

Our recursion above is not very efficient, because we have pending on every step of the recursion. There is a more efficient way by using an extra argument called accumulators. An accumulator is an additional argument that added to a function. We store the result of every step of the process in the accumulator.

For example, lets do length of list by using an accumulator. First, lets initialize an accumulator. An accumulator is where we store our result. Therefore, in this case, we initialize it with 0. Then, like we do above, every time we take an element out of the list, we add 1 to the result. So, we add 1 to the accumulator, every time we take an element out of the list. When the list is empty, we return the accumulator.

Graphical user interface, application, website

Description automatically generated

Recursion using accumulator

A whiteboard with writing on it

Description automatically generated with medium confidence

General recursion

Text, letter

Description automatically generated

Absolutely, using an accumulator is more efficient than the general recursion. It does not need to wait and jump back and forth.

With the idea of accumulator, we can optimize the recursion that we wrote earlier.

There is a new concept called tail call, which is the function gets everything done at the end.

Like the Len function with accumulator, we wrote above, there is no pending or anything else at the end of the function, it done its job at the ending. But in the general recursion, it got pending at the ending, it must wait the next function to return the result.

Tail call recursion

Graphical user interface, application, website

Description automatically generated

General recursion

A screenshot of a video game

Description automatically generated

How do we convert general recursion to iteration? With the idea of accumulator and tail call, it will be very easy. What we do is just move the operation from the end of the function to the accumulator and then wrap it as a helper function.

For example, let’s write a length function in iterative way. First step, determine the base case, and initialize the accumulator. The base case is when the list is empty, we return the accumulator. So, we initialize the accumulator to be 0. Second step, every time when we take an element out of the list, we add 1 to the accumulator, until the list is empty. Therefore, we get

Graphical user interface, text

Description automatically generated

Now, we call the helper function on our length function, and then initialize the accumulator.

Graphical user interface, text

Description automatically generated

Then we done with converting general recursion to iteration!

All in all, 3 steps:

Step 1. Determine the base case, the helper function should return the accumulator on the base case.

Step 2. Make calculation on the accumulator.

Step 3. Call the helper function in your main function with an appropriate initial value.

**Invariants**

Invariant is something that is always true within the loop or recursion. It ensures that the cycle continues correctly. Its essence is mathematical induction. You'll see it in any function that exists in a loop or recursion.

Let’s make the len function we wrote above as an example. Absolutely, what is always true in the recursion is.

In the base case, what is always true is:

Length of [] is 0. That is always true.

In the inductive case, we have

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

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

With this proof, we can be sure that our recursion is correct and that it will eventually return the correct result.

That is what invariants is. In each loop, the algorithm transfers the data from one state to another state, and the current state will be the starting state for the next loop (unless there's no next loop, or it will immediately end after this loop ends). So, you have to ensure that it conforms to a "feature" makes the algorithm can operate correctly.

**Higher order function**

What is a higher-order function? A function that takes another function as an argument or returned it as a result value is called higher-order function.

Let’s make the built-in function “abs” as an example. What it does is it take a number as argument and return its absolute value.



However, the words “abs” is only a valuable name instead of a function. We can call it is because it is pointing to a function that can calculate absolute value. If you do not believe, we can assign it to another valuable name, and see what will happen

Text

Description automatically generated  
It works! Now, what if we assign another value to “abs”?

Text

Description automatically generated

Now, abs will not work as a function anymore. Because “abs” is no longer pointing to an absolute value function but to an integer.

Since variables can point to functions and arguments to functions can accept variables, a function can accept arguments to another function, which is called a higher-order function.

Let’s try to write a high-order function



If we run add (-1) 2 abs, x will be -1, y will be 2 and f will be abs, then we will get

Add (-1) 2 abs = abs (-1) + abs (2) = 1 + 2 = 3



All in all, to write a higher-order function is to make its arguments accept other functions.