

19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Recursion in Haskell

RECURSION

- Recursion is actually a way of defining functions in which the function is applied inside its own definition
- Recursion is important to Haskell because unlike imperative languages, computations are done in Haskell by declaring what something is instead of declaring how to get it.
- There are no while loops or for loops in Haskell and instead we many times have to use recursion to declare what something is.

EXAMPLE –MAXIMUM FUNCTION USING RECURSION

--maxrec.hs

maxrec :: (Ord a) => [a] -> a

maxrec [] = error "maximum of empty list"

maxrec [x] = x

maxrec (x:xs)

| x > maxTail = x

| otherwise = maxTail

where maxTail = maxrec xs

Check if the head is greater than the maximum of the rest of the list. If it is, return the head. Otherwise, return the maximum of the rest of the list.

```
ghci> maxrec [5,2,3,1,4]
```

```
5
```

```
ghci> maxrec [454,678,989,12,1]
```

```
989
```

```
ghci> maxrec []
```

```
*** Exception: maximum of empty list
```

```
CallStack (from HasCallStack):
```

```
  error, called at maxrec.hs:3:13 in main:Main
```

```
ghci> maxrec [1]
```

```
1
```

SAMPLE EVALUATION OF MAXREC

- Let's take an example list of numbers and check out how this would work on them: [2,5,1].
- If we call maximum' on that, the first two patterns won't match.
- The third one will and the list is split into 2 and [5,1].
- The where clause wants to know the maximum of [5,1], so we follow that route.
- It matches the third pattern again and [5,1] is split into 5 and [1].
- Again, the where clause wants to know the maximum of [1] .
- Because that's the edge condition, it returns 1.
- Finally! So going up one step, comparing 5 to the maximum of [1] (which is 1), we obviously get back 5.
- So now we know that the maximum of [5,1] is 5.
- We go up one step again where we had 2 and [5,1] .
- Comparing 2 with the maximum of [5,1], which is 5, we choose 5.

IMPERATIVE VERSION OF MAXIMUM ELEMENT IN A LIST

```
#include <stdio.h>

int main()
{
    int size, i, largest;
    printf("\n Enter the size of the array: ");
    scanf("%d", &size);
    int array[size];
    printf("\n Enter %d elements of the array: \n", size);
    for (i = 0; i < size; i++)
    {
        scanf("%d", &array[i]);
    }
    largest = array[0];
    for (i = 1; i < size; i++)
    {
        if (largest < array[i])
            largest = array[i];
    }
}
```

Algorithm !

- Set up a variable to hold the maximum value so far
- Loop through the elements of a list
- If an element is bigger than then the current maximum value,
 - Replace it with that element.
- The maximum value that remains at the end is the result

A CLEANER VERSION OF MAXREC USING MAX

```
--maxrec1.hs
```

```
maxrec1 :: (Ord a) => [a] -> a
```

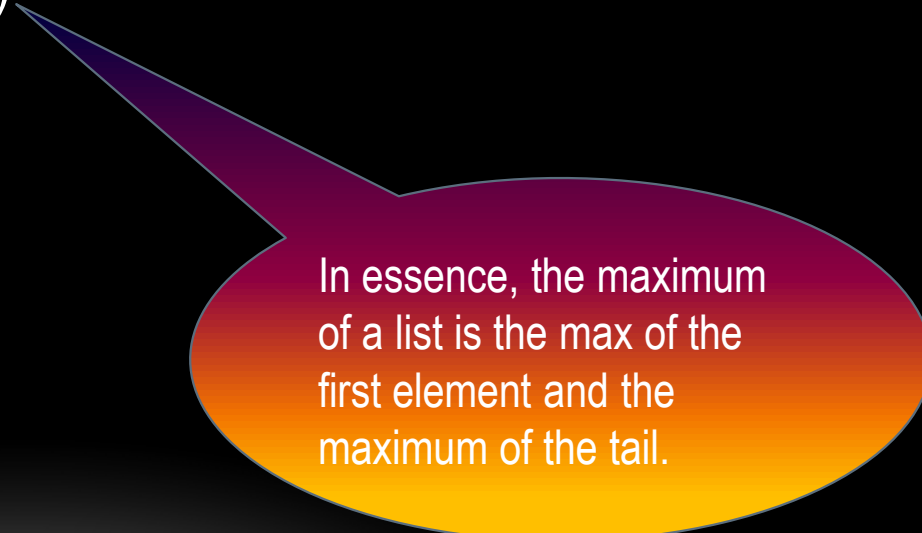
```
maxrec1 [] = error "maximum of empty list"
```

```
maxrec1 [x] = x
```

```
maxrec1 (x:xs) = max x (maxrec1 xs)
```

```
ghci> maxrec1 [2,5,1]
```

```
5
```



In essence, the maximum of a list is the max of the first element and the maximum of the tail.

EXAMPLE - REPLICATE

```
--myreplicate.hs
```

```
myreplicate :: (Num i, Ord i) => i -> a -> [a]
```

```
myreplicate n x
```

```
  | n <= 0 = []
```

```
  | otherwise = x:myreplicate (n-1) x
```

```
ghci> myreplicate 3 5
```

```
[5,5,5]
```

```
ghci> myreplicate 0 5
```

```
[]
```

Note: Num is not a subclass of Ord which means that what constitutes for a number doesn't really have to adhere to an ordering. Hence we have to specify both the Num and Ord class constraints when doing addition, subtraction or comparison.

Replicate takes an Int and some element and returns a list that has several repetitions of the same element.

- Guards are used here instead of patterns as we're testing for a boolean condition.
- If n is less than or equal to 0, return an empty list.
- Otherwise return a list that has x as the first element and x replicated n-1 times as the tail.
- Eventually, the (n-1) part will cause our function to reach the edge condition.

EXAMPLE - TAKE

--mytake.hs

mytake :: (Num i, Ord i) => i -> [a] -> [a]

mytake n _

| n <= 0 = []

mytake _ [] = []

mytake n (x:xs) = x : mytake (n-1) xs

Try using a piece of paper to write down how the evaluation would look like if we try to take, say, 3 from [4,3,2,1].

ghci> mytake 3 [5,4,3,2,1]

[5,4,3]

ghci> mytake 0 [5,4,3,2,1]

[]

ghci> mytake 1 []

[]

- The first pattern specifies that trying to take a 0 or negative number of elements, returns an empty list.
- _ is used to match the list because we don't really care what it is in this case.
- a guard is used without an otherwise part. Hence if n turns out to be more than 0, the matching will fall through to the next pattern.
- The second pattern indicates that trying to take anything from an empty list, returns an empty list.
- The third pattern breaks the list into a head and a tail. Taking n elements from a list equals a list that has x as the head and then a list that takes n-1 elements from the tail as a tail.

EXAMPLE - REVERSE

--myreverse.hs

myreverse :: [a] -> [a]

myreverse [] = []

myreverse (x:xs) = myreverse xs ++ [x]

ghci> myreverse [1,2,3,4,5]

[5,4,3,2,1]

An empty list reversed equals the empty list itself !

if we split a list to a head and a tail,
the reversed list is equal to the
reversed tail and then the head at
the end

EXAMPLE - ELEM

--elem'.hs

elem' :: (Eq a) => a -> [a] -> Bool

elem' a [] = False

elem' a (x:xs)

| a == x = True

| otherwise = a `elem'` xs

ghci> elem 2 [1,2,3,4,5]

True

ghci> elem 2 []

False

It takes an element and a list and sees if the element is present in the list.

Empty list contains no elements

If the head isn't the element then we check the tail. If we reach an empty list, the result is False.

EXAMPLE – QUICK SORT

```
quicksort :: (Ord a) => [a] -> [a]
```

```
quicksort [] = []
```

```
quicksort (x:xs) =
```

```
  let smallerSorted = quicksort [a | a <- xs, a <= x]
```

```
      biggerSorted = quicksort [a | a <- xs, a > x]
```

```
  in smallerSorted ++ [x] ++ biggerSorted
```

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
```

```
[1,2,2,3,3,4,4,5,6,7,8,9,10]
```

```
ghci> quicksort "the quick brown fox jumps over the lazy dog"
```

```
" abcdeeeefghhijklmnoooopqrrsttuuvwxyz"
```

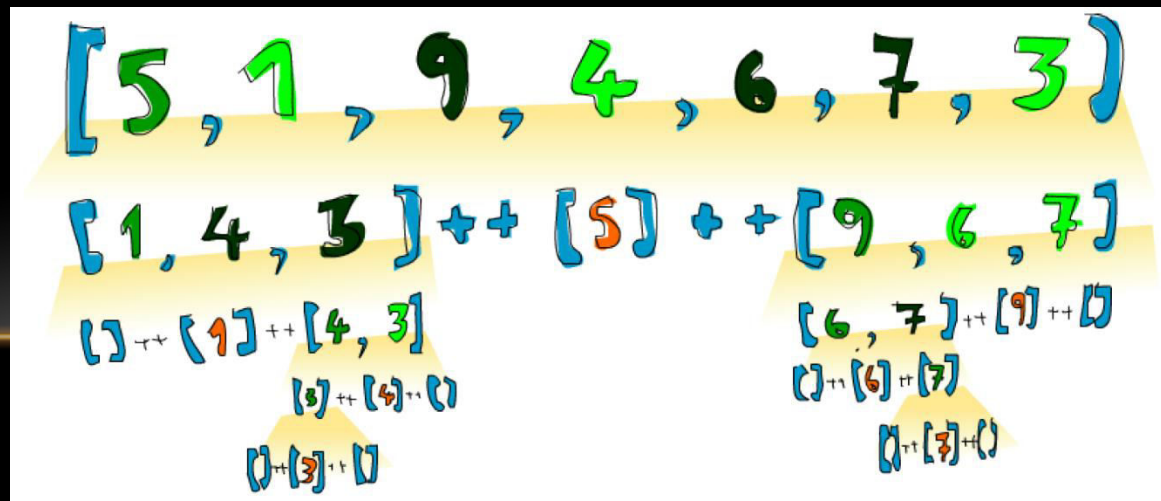
- Type signature - quicksort :: (Ord a) => [a] -> [a].
- Edge condition - Empty list. A sorted empty list is an empty list.
- Main algorithm: a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted).

QUICK SORT EVALUATION

- If we have to sort $[5, 1, 9, 4, 6, 7, 3]$, the algorithm will first take the head, which is 5 and then put it in the middle of two lists that are smaller and bigger than it.
- Hence at a point, you will have $[1, 4, 3] ++ [5] ++ [9, 6, 7]$.
- Once the list is sorted completely, the number 5 will stay in the fourth place since there are 3 numbers lower than it and 3 numbers higher than it.
- Now, if $[1, 4, 3]$ and $[9, 6, 7]$ are sorted, a sorted list is obtained ! The two lists are sorted using the same function.
- Eventually, it reaches empty lists and an empty list is already sorted in a way, by virtue of being empty.

Legend

- Orange - An element that is in place and won't move anymore
- Green – Pivot
- Light green – Elements smaller than pivot
- Dark green – Larger than pivot
- Yellow gradient – A Pass of QS



QUICK SORT IMPERATIVE ('C') IMPLEMENTATION

```
#include<stdio.h>

void quicksort(int number[25],int first,int last){
    int i, j, pivot, temp;

    if(first<last){
        pivot=first;
        i=first;
        j=last;

        while(i<j){
            while(number[i]<=number[pivot]&& i<last)
                i++;
            while(number[j]>number[pivot])
                j--;
            if(i<j){
                temp=number[i];
                number[i]=number[j];
                number[j]=temp;
            }
        }

        temp=number[pivot];
        number[pivot]=number[j];
        number[j]=temp;
        quicksort(number,first,j-1);
        quicksort(number,j+1,last);
    }
}
```

```
    }
}

int main(){
    int i, count, number[25];

    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);

    printf("Enter %d elements: ", count);
    for(i=0;i<count;i++)
        scanf("%d",&number[i]);

    quicksort(number,0,count-1);

    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
        printf(" %d",number[i]);

    return 0;
}
```

THINKING RECURSIVELY – SUMMING UP !

- Pattern Observed:
 - Define an edge case and then
 - define a function that does something between some element and the function applied to the rest.
 - Example: A sum is the first element of a list plus the sum of the rest of the list.
 - A product of a list is the first element of the list times the product of the rest of the list.
 - The length of a list is one plus the length of the tail of the list etc.
- Edge cases: Some scenario where a recursive application doesn't make sense.
 - E.g., For a list - empty list; For a tree – Leaf node
 - Factorial - some number and the function applied to that number modified. Factorial of 0 doesn't make sense and since multiplication, set as 1 (identity for product)
 - For lists, sum of empty lists is 0 (Identity for addition)
 - For quick sort – Edge case and identity are empty lists

THINKING RECURSIVELY – SOME GUIDELINES...

- So when trying to think of a recursive way to solve a problem...
- Try to think of when a recursive solution doesn't apply and see if you can use that as an edge case,
- think about identities and
- think about whether you'll break apart the parameters of the function (for instance, lists are usually broken into a head and a tail via pattern matching) and on which part you'll use the recursive call.

Reference:

Miran Lipovaca - Learn You a Haskell for Great Good!_ A Beginner's Guide-No Starch Press (2011)

THANK YOU