Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

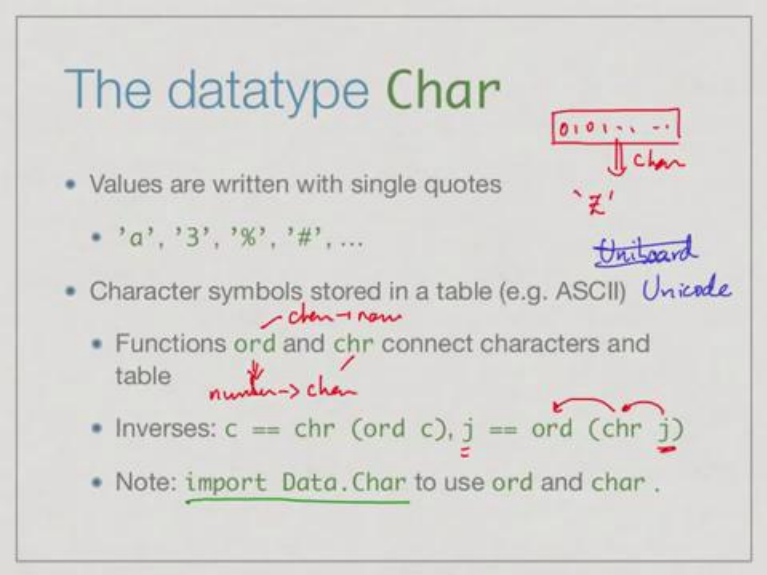
Module # 02

Lecture - 03

Characters and Strings

So, let us turn our attention to a data type we have very looked at very much so far, namely the data type of characters and associated with that the data types strings.

(Refer Slide Time: 00:12)



So, characters are the symbols that we can type from our keyboard for example. So, these are the type of items, which are manipulated in software like word processors, so it is an important part of computational software that people write. So, the way Haskell represents characters in a program is to put the symbol between single quotes, so you have single quote a or single quote percentage and so on.

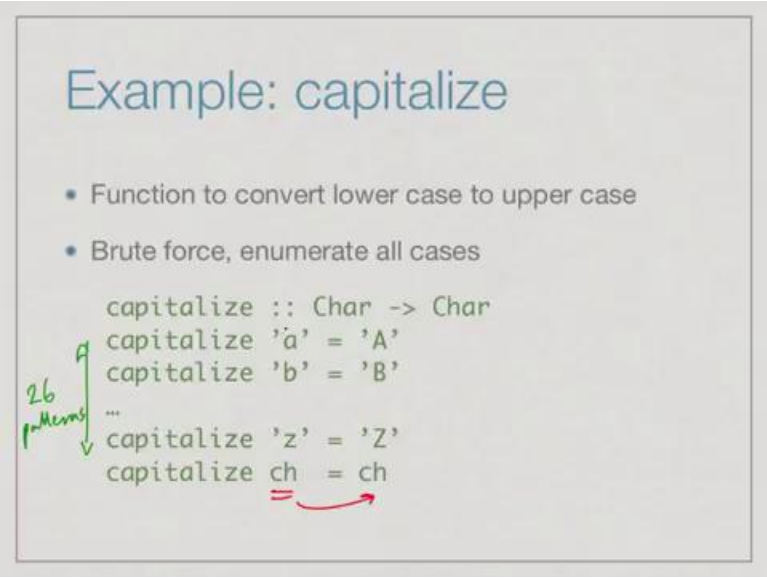
Now, as all of you know actually a character is typically stored in some binary format in the memory. So, there will be a table, which tells us how to interpret a particular bit, so if you want to call this a character, then this represents a characters Z for example. So, there is a table look up, which tells us how to treat a bit string, which is essentially an integer and converted into a number into a character.

So, there are various standards for this, so the whole standard which is very common is called ASCII, a model standard which allows more characters. Let us for example, in non English and especially symbols in languages from say places like India or Asia, which is called a Unicode. So, Unicode is a two byte character representation, ASCII is a one byte thing, basically there is a stable, so now, we need a way to go backwards and forwards.

So, you need to find out the code of a character and you need to find out, what the given code corresponds to. So, these are the two functions that Haskell provides for that, it is called ord and chr, so ord basically takes a number and gives us a character and chr takes a number and ord takes a character and gives us number. So, ord gives us the code for a character and chr gives us the character for a given number, so these are inverses.

So, if I take a character take it is number and then, convert the character back I will get the same character. Similarly, if I take a number, convert it into a character and then, extract it is number or it will get back the same number. So, if you want to use these functions in our Haskell code, they are not included by default, so you have to import a character module. So, you have to use this statement at the top of your dot excess file import data dot chr, if you want to use ord and char.

(Refer Slide Time: 02:45)



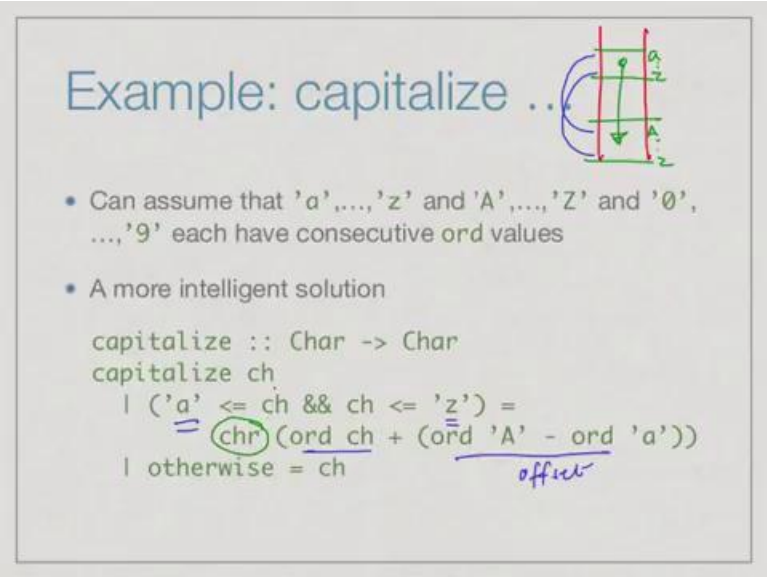
So, let us look at our function that I might want to write, so this is the function we mentioned as an example earlier on. So, supposing we want to write a function that converts all lower case letters to upper case letters and does not do anything for non

letters. So, it should keep numbers as numbers plus a punctuation marks is punctuation marks, but small a to small z will become capital A to capital Z.

So, one way to do this without using any information about how characters are represented is just to make exploit the fact that there are only finitely many characters. Say for instance, we are dealing with the English I want the small properly the Roman alphabet it has only 26 letters a to z, so then we can just write out 26 patterns. So, these are now might 26 patterns, capturing all the given inputs that need to be mapped and finally, we have something which will just copy the input to the output, if it does not match any of these patterns.

So, if I feed it a character which is not a to z, then it will just return. So, percent will come back as percent, question mark is question mark, 9 as 9, but small a will become capital A. So, this of course, is a very Brute force thing and it relies on the fact that we can enumerate of course, this is a very tedious way to write such a program.

(Refer Slide Time: 04:03)

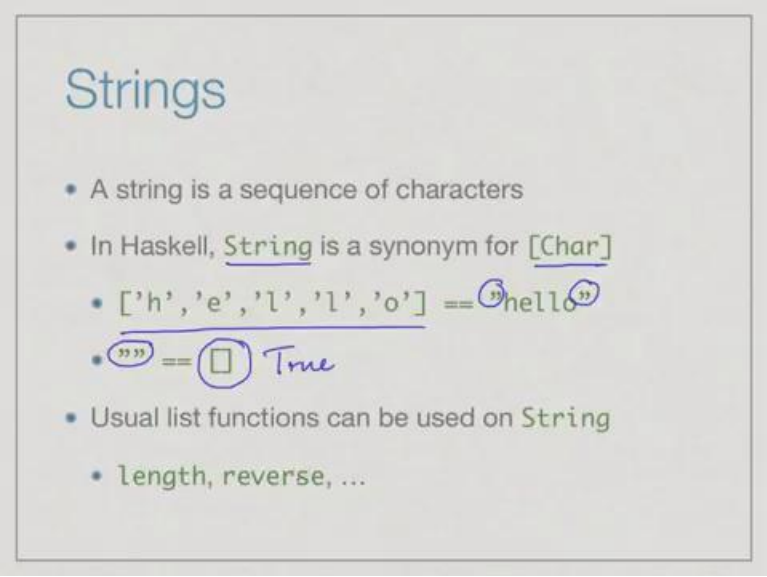


So, what we can do is use the fact that in any of the encodings that Haskell systems will use, the letters are actually in blocks. So, this small letters capital A to small a to small z will all occur consequently in the table, so it will capital A to capital Z, so it will 0 to 9. So, for instance if I have the overall set of encodings, then maybe I have one block, which is small a to small z, maybe I have another block which is capital A to capital Z.

And now, the crucial thing is that the distance from small a to capital A is the same as the distance for small z to capital Z. So, if I look at the encodings they have the same distance apart, so that is on offset. So, what we can do is now we can write this function which says that, if now because the encodings characters can be compared small a is less than small b is less than small c and so on. So, in the given character lies between small a and small z, if ch is bigger than or equal to small a and smaller than or equal to small z, then we add to it is code this is the offset.

So, we add to the code of the given character, the offset which will shifted from this block to this block by an uniform amount, wherever it is, it shift by the same amount and then, we recompute the character back and we get the capital A. So, small a will go to capital A and then, we recompute using the chr function with this. So, this is a way of capitalizing using chr, ord and the additional property that the ord numbering for characters are actually consequent.

(Refer Slide Time: 05:44)



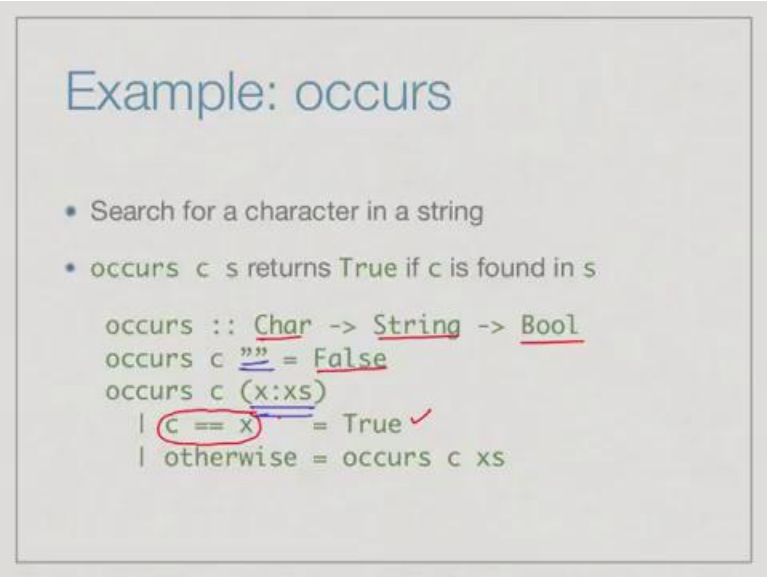
So, usually programs at manipulate character do not manipulate individual characters, but manipulates strings, so string is just a sequence of characters. So, when we have a text processor or something like that, you need a line of input, which will be a sequence of characters or even many lines of input and it will do something do that. So, in Haskell the word string with the capital S is a data type, but it is only a synonym, it is exactly the same in every context as list of char.

So, in other words I can write strings using the familiar double code notation, so I can write a sequence of symbols with double codes. So, this is the string as you would see it in any other programming language. But, internally as for as Haskell is concerned, this notation is identical to having a list with five elements each of which is an individual character, so this is basically how Haskell works.

So, therefore, the empty string, which is just two consecutive double quotes is the same as the empty list, if I say double quote, double quote equal to equals to square bracket open square bracket close, the answer will be true. So, this is just an alternative notation what sometimes programming language people called syntactic sugar. So, this is just the different way of writing the same thing, which is more convenient for the program.

Now, the useful thing is that, because these are all lists, everything that one can do with list directly applies to string. We do not needs separate functions for strings, because we can take length of the string, because length of the string is just the length of the other language. We can reverse the string, because there is some reverse in the other language and so on. So, all the things that we saw for list like a tail, take, drop all these functions work as well on strings as they do want regular lists.

(Refer Slide Time: 07:37)

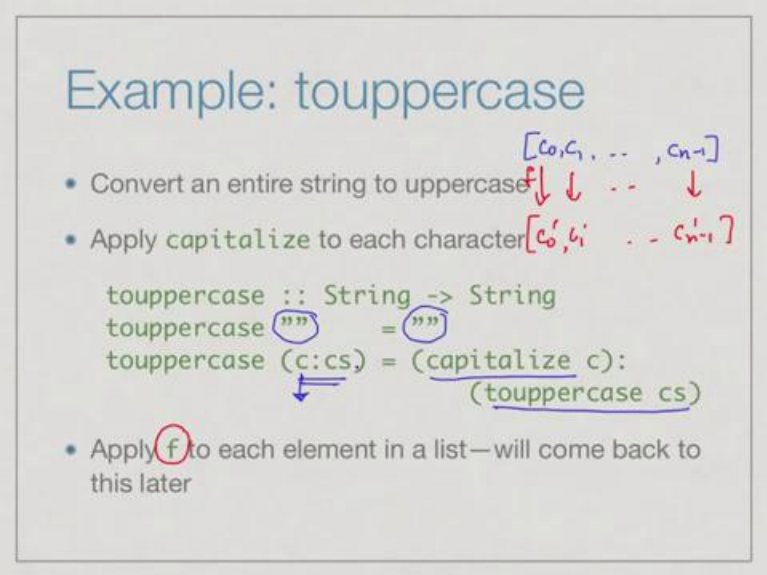


So, similarly if wants to write a function on a string, one would think of it as a list and use induction on the string and then, talk about the function on the empty string as the base case and what to do if you have a string extended a one letter to the left. So, here is

the simple function, we want to check, if a given character occurs in a given string. So, occurs is the function, which takes as input a character a string to search for a character and returns whether or not the characters.

So, the base case such that, if I have a character and search in the empty string, then it is not there the answer must be false. And now, if I search in the non empty string I compare it the first character, so if it is the first character, then I found it if it is not the first character I am a search in the rest. So, I continue my search by looking for occurs c of x s. So, we are doing the familiar induction that we get for list on the string, if we are doing the base case the empty string and the inductive case for the non empty string. So, is exactly like a programming list.

(Refer Slide Time: 08:47)



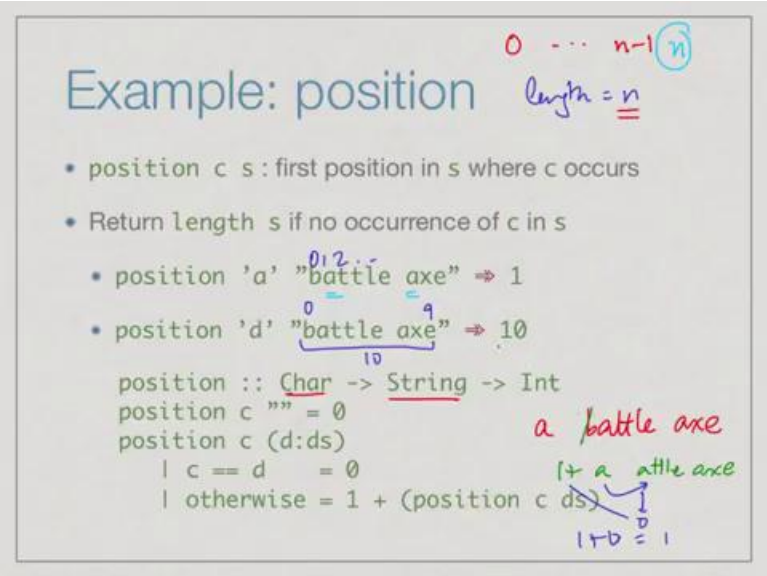
So, here is another function supposing we want to take a string and convert every lower case letter in that string to upper case. Now, we know how to convert one letter, because we already written this function in capitalize. So, again we can apply induction we say that, if you want to convert the empty string to upper case, then nothing happened we just keep the empty string. And now, if we have a non empty string, what we do is we change the first letter to upper case and then, inductively do this the capitalized the first letter and then, we applied to a upper case to the rest.

So, this is actually a situation, where I had say a string, which consists of n characters. And each of these I am going to now apply capitalized to get a new set of characters,

which if they are not capitalized the punctuation mark, so numbers will be the same, otherwise it will capital. So, what we have doing abstractly, if we are taking a function f this case of function f is capitalize and we applying it uniformly to a list, so that we get a new list of the same length we are have instead of c 0 f of 0, which is c 1 f of c 1 and so on.

Now, this is very important operation, which we need to understand for list when we will definitely comeback to this later. But, this is the specific example of it where we have just inductively defined how to capitalizes.

(Refer Slide Time: 10:15)



So, moving on to another example supposing, we want to find the first position of a character in a string remember that in any list the positions of a list of length n are from 0 to n minus 1. So, if I take the length as n, then the valid positions are 0 to n minus 1, so any position, which is outside this range is invalid position. So, we could for example, treat n has the default answer, if the string is not present.

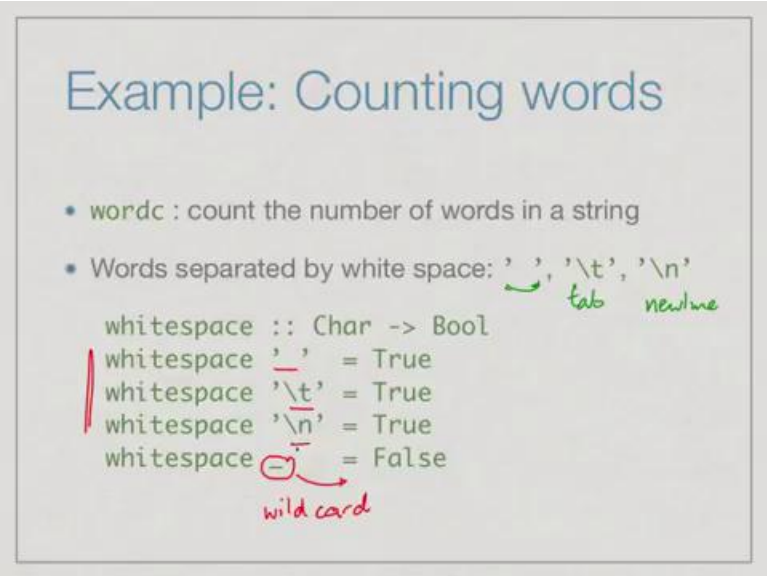
So, either it should give as the first position in s, where c occurs or if there is no occurrence c in s gives us the default value, which is the length, which is outside, so this is beyond the last position. So, for example, if I take this string battle axe, then the two occurrences of a, here and here or because of numbering scheme this is 0 1 2 so on. So, the number we should return is 1, because position 1 is the first occurrence.

On the other hand letter d does not occur in battle axe the length of this string is 10 the positions are numbered 0 to 9, so I return a value of ten saying it is not found. So, how do we write position is quite a straight forward inductive function again, so we take a character take a string and return an integer. So, if it is not found if it is an empty string is definitely not found remember that the length of the empty string is 0, so I should returns 0 by axe specification and this is correct.

Because, if it were a valid position 0, then we have actually one element in the string. So it has no elements in the string, so they were no valid positions, so 0 is the first invalid position. And on the other hand, if I take a non empty string as before I will check, whether it is the first position if it is first position, then we suspect to this term is to 0, if it not founded, then I will accumulated the positions are skip. So, for if I do not find it I add one to the position of the character in the remaining part.

So, if I find for example, battle axe I would first check a with battle axe and I would say this does not match. So, in 1 plus the position of a in battle axe and now comes back with the 0, so I get 1 plus 0 is equal to 1, which is a correct task. So, 1 has to the reason, we are going through all these functions, which is some get practice in working with these inductive definition on list string, which are all basically different versions of the same.

(Refer Slide Time: 12:49)



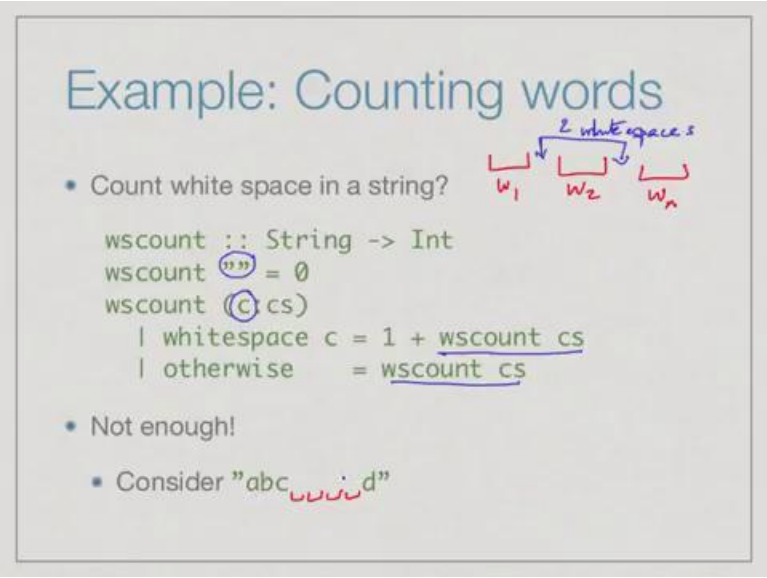
So, as a final example using strings, let us look at something little more complex. So, supposing, we want to found count the number of words in the string, so we have to

define, what a word is. So, word by convention is the sequences of characters will does not have a blank space, so blank space could be either a real blank. So, this represents a blank character this is a tab. So, it can press the tab, but known your keyboard and you get some member of spaces depending on all the tabs are same and this backslash n is a new line.

So, this is the familiar that encoding that maybe programming language backslash it is stands for tab backslash n stands for new line. So, these are whitespace character, so we can first define a function, which classifies a single character whitespace or not. So, all it does it for a three cases, which we had define whitespace we can we might want to later on add cases, but right now we are only saying space tab a new line or whitespace everything else is not whitespace and notice, that we do not need this value here.

So, we have this wildcard pattern he mention lastly if you do not require the input the output, we can just ignore the name and this uses underscore as a wildcard pattern. So, this is everything which is other than these three characters is not punctuations.

(Refer Slide Time: 14:12)

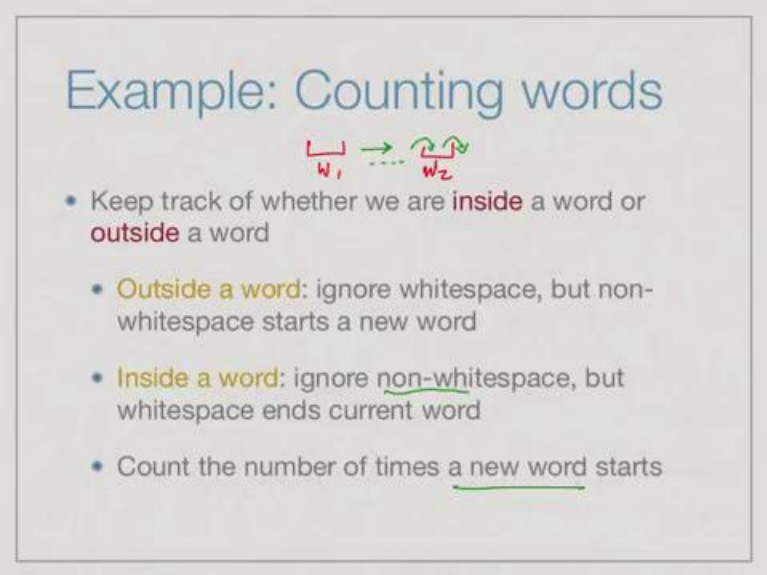


So, the first thing that we could do we assume that, if I have word 1 word 2 and so on. Then, they will each be separated by white space I will have this space two white spaces, so if I can count the number of white spaces, if I add 1 I should get the number of words. So, the first attempt to be to just count the white space, so if I count the white space in the empty string it is 0, if I count the white space in a non empty string, then depending

on other first is a white space or not I add 1 or I just look at the list.

So, if it is the first character if c is a white space, then I add 1 and I continue inductively count the remaining white spaces, otherwise I do not count this, because this is not the white space, then I just recount ((Refer Time: 15:01)). So, this is a very straight forward count the white spaces, but unfortunately this would give as a wrong answer. Because, then a function is next thing like this, where I have maybe four blanks between towards we could say there are 4 white spaces and therefore, this constitutes 4 or 5 words and actually there are towards. So, actually, what is important is not the number of white spaces, but how many times one goes some a word outside and back.

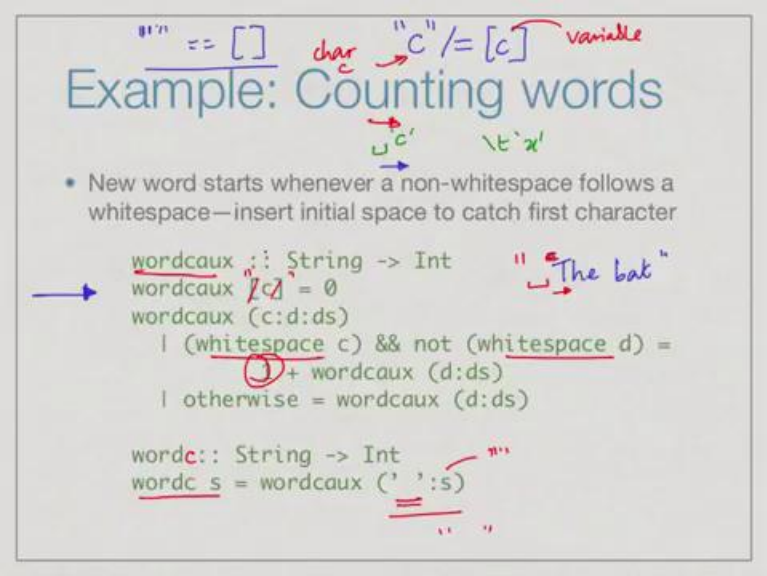
(Refer Slide Time: 15:27)



So, one could write a program with check whether we are inside the word or outside a word. If we are outside a word, so we have a word and then, we have a other word, so in the outside of word the maybe many white spaces, but we do not care. Now, when we hide a non white space we transfer to the inside a word, now inside a word we will ignore non white space, but when we hide the white space we will leave the word.

So, we can use this idea to count words, so actually it is enough to either check, how many times we enter a word, how many times we leave a word, So, in this case we will count the number of times a new word starts.

(Refer Slide Time: 16:12)



So, new words starts whenever you go from a white space, so from a space to say the characters c or from a tag to the character x. So, whenever we transform one position have a white space to a position does not have a white space then by definition new words starts. So, of course, now if I have a sentence like the bat, then I must make sure that I count this as the beginning of the words I must say that I go from a non white space white space, but I start directly with the letter t.

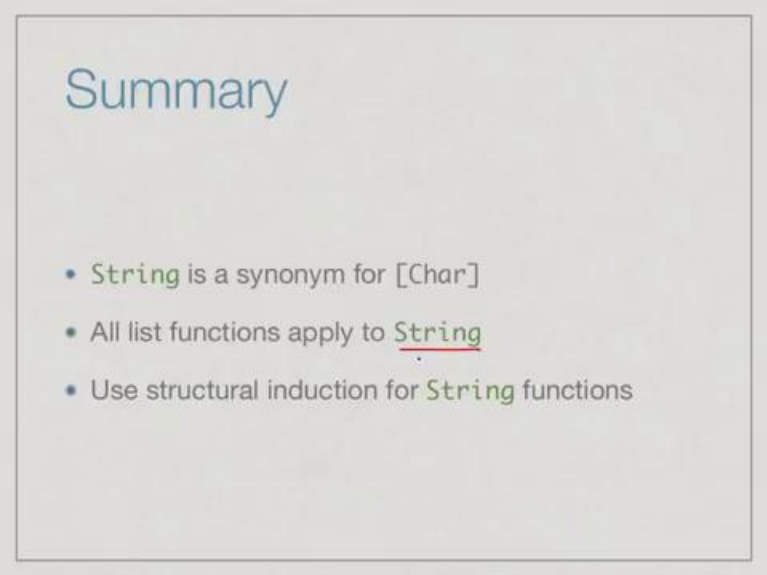
So, what I will do is I will always take whatever string I have and insert a space before, so the very first words get counted correctly. So, the word count, so this should be word count, so the function word count, which I am going to write on given string we call this auxiliary function by first adding a blank space before this string. So, now, this make sure that the very first word is counted correct, now we are fine, what this says is that, if I have just the single character, then I cannot make a transition.

So, I say there are no words, but if have more, then one character see notice that I will always call word count aux with the non empty string, because even if this is empty this whole thing is the string containing the one blank. So, I will never call this the word c aux with an empty string, so I do not need a pattern match case for the empty string. So, if I have one character I will declare it is has no words, if it has two characters, because if I have only one character, then cannot be a word cannot be the starting point of the word because I would flagrant of the character before.

If have two characters check whether the current characters white space and the next characters not white space this is exactly transition, if so I add one to the word come, otherwise continue counting words as before. One important thing here to notice is that when we two empty string we said that the double code double code is equal to this is an equality. However, double code c is not in general equal to this, because when I write double codes c this is the characters c, where as here this is a variable.

So, when I write a function definition like this I need that this c stands for any character, if instant or mistake I has return double code it say that this pattern matches precisely when I have a one element character a single c. So, therefore, one should be careful in some situation you have to use this notation even though you are dealing with strings.

(Refer Slide Time: 19:07)



So, him summaries we have look at the character data type and, what we have observe is that a string, which is a usual unit, in which work assumes manipulates characters is just a sequence of characters. And hence, Haskell provides the types string as a synonym for a list of char and because it is a list all the usual list functions can we apply to string like length or reverse or take or drop and so on.

And also, because these are just list we can write function, that manipulate list manipulate string using the structure induction we exactly this same way, that we use structural induction for a list by looking at the base case the empty string and the inductive case a non empty string were we separate out first character on the list.