

Introduction To Haskell Programming

Prof. 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 not really looked at very much so far, namely the data type of characters and associated with that the data type of strings.

(Refer Slide Time: 00:12)

The datatype Char

- Values are written with single quotes
 - 'a', '3', '%', '#', ...
- Character symbols stored in a table (e.g. ASCII) Unicode
- Functions `ord` and `chr` connect characters and table
 - Handwritten: char → num*
 - Handwritten: number → char*
- Inverses: `c == chr (ord c)`, `j == ord (chr j)`
- Note: `import Data.Char` to use `ord` and `chr`.

Handwritten annotations:

- A box containing `01011111` with an arrow pointing down to `'z'` and the label `char`.
- The word `Unicode` is written in blue.
- Arrows indicate the mapping between characters and numbers for the `ord` and `chr` functions.

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 that Haskell represents characters in a program is to put the symbol between single quotes, so you have 'a' or '%' and so on.

Now as all of you know 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 character 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 convert it into a character.

So, there are various standards for this, so the old standard which is very common is called ASCII, a modern standard which allows more characters sets, 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 this table, 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 takes a character and gives us a number which is the code for the character and chr gives us the character for a given number, so these are inverses.

So, if I take a character take its number and then, convert to character back I will get the same character. Similarly, if I take a number, convert it into a character and then, extract its number I 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 .hs file, import data.char , if you want to use ord and char.

(Refer Slide Time: 02:45)

Example: capitalize

- Function to convert lower case to upper case
- Brute force, enumerate all cases

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
capitalize ch  = ch
```

Handwritten notes: "26 patterns" with a vertical arrow pointing from 'a' to 'z', and a red arrow pointing to the default case 'ch = ch'.

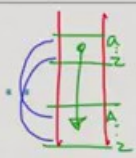
So, let us look at a function that I might want to write, so this is the function we mentioned as an example earlier on. So, supposing we want to write a function that converts all lower case letters to upper case letters and does not do anything for non letters. So, it should keep numbers as numbers plus punctuation marks as punctuation marks, but 'a' to 'z' will become 'A' to 'Z'.

So, one way to do this without using any information about how characters are represented is just to exploit the fact that there are only finitely many characters. Say for instance, we are dealing with the English or what is more properly the Roman alphabet it has only 26 letters a to z, so then we can just write out 26 patterns. So, there are now 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, '%' will come back as '%', '?' as '?', 9 as 9, but small a will become capital A. So, this of course, is a very bruteforce 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)

Example: capitalize ..



- Can assume that 'a', ..., 'z' and 'A', ..., 'Z' and '0', ..., '9' each have consecutive **ord** values
- A more intelligent solution

```

capitalize :: Char -> Char
capitalize ch
  | ('a' <= ch && ch <= 'z') =
    = chr (ord ch + (ord 'A' - ord 'a'))
  | otherwise = ch
    
```

offset

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

And now, the crucial thing is that the distance from 'a' to 'A' is the same as the distance from 'z' to 'Z'. So, if I look at the encodings they have the same distance apart, so that is an offset. So, what we can do is now we can write this function which says that because the encodings of characters can be compared 'a' < 'b' < 'c' and so on. So, if the given character

lies between 'a' and 'z', if $ch \geq 'a'$ and $ch \leq 'z'$, then we add to its code the offset.

So, we add to the code of the given character, the offset which will shift it from this block to this block by an uniform amount, wherever it is, it will shift by the same amount and then, we could recompute the character back and we get the 'A'. So, 'a' will go to '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)

The slide is titled "Strings" in a large blue font. It contains a list of bullet points with handwritten annotations in blue ink:

- A string is a sequence of characters
- In Haskell, String is a synonym for [Char]
 - ['h','e','l','l','o'] == "hello" (The list and the string are circled, and an arrow points from the list to the string.)
 - "" == □ True (The empty string and the empty list are circled, and an arrow points from the empty string to the empty list.)
- Usual list functions can be used on String
 - length, reverse, ...

So, usually programs that manipulate characters do not manipulate individual characters, but manipulate 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. 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 or [Char].

So, in other words I can write strings using the familiar double quote notation, so I can write a sequence of symbols with double quotes. So, this is the string as you would see it in any other programming language. But, internally as far 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, `"" == []`, 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 strings. We do not need separate functions for strings, because we can take length of the string, because length of the string is just the length of the underlying list. We can reverse the string, because we are just reversing the underlying list and so on. So, all the things that we saw for list like head, tail, take, drop all these functions work as well on strings as they do want regular lists.

(Refer Slide Time: 07:37)

Example: occurs

- Search for a character in a string
- `occurs c s` returns True if `c` is found in `s`

```
occurs :: Char -> String -> Bool
occurs c "" = False
occurs c (x:xs)
  | c == x = True ✓
  | otherwise = occurs c xs
```

So, similarly if one 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 with a one letter to the left. So, here is a 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 character is there.

So, the base case says that if I have a character and I search in the empty string, then it is not there, so the answer must be false. And now, if I search in the non empty string, then I compare it with the first character, so if it is the first character, then I found it, if it is not the

first character I must search in the rest. So, I continue my search by looking for occurs c xs. So, we are doing the familiar induction that we did for list on the string, if we are doing the base case for the empty string and the inductive case for the non empty string. So, its exactly like a programming a list.

(Refer Slide Time: 08:47)

Example: touppercase

- Convert an entire string to uppercase $[c_0, c_1, \dots, c_{n-1}]$
- Apply **capitalize** to each character $[c'_0, c'_1, \dots, c'_{n-1}]$

```

touppercase :: String -> String
touppercase "" = ""
touppercase (c:cs) = (capitalize c):
                    (touppercase cs)

```

- Apply **f** to each element in a list—will come back to this later

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 have already written this function capitalize. So, again we can apply induction we say that, if you want to convert the empty string to upper case, then nothing happens, we just get back 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. We capitalized the first letter and then, we applied touppercase to the rest.

So, this is actually a situation, where I have, say a string, which consists of n characters. And each of these I am going to now apply capitalize to get a new set of characters, which if they are not capitalizable, so the punctuation marks, numbers will be the same, otherwise they will be in capital. So, what we are doing abstractly is, if we are taking a function f, this case the function f is capitalize and we are applying it uniformly to a list, so that we get a new list of the same length where I have instead of c_0 $f(c_0)$ which is c'_0 and instead of c_1 we have $f(c_1)$ and so on.

Now, this is a very important operation, which we need to understand for list when we will

definitely come back to this later. But, this is a specific example of it where we have just inductively defined how to capitalize a string.

(Refer Slide Time: 10:15)

Example: position

0 ... n-1 (n)
length = n

- `position c s`: first position in `s` where `c` occurs
- Return `length s` if no occurrence of `c` in `s`

• `position 'a' "battle axe" => 1`

• `position 'd' "battle axe" => 10`

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
  | c == d    = 0
  | otherwise = 1 + (position c ds)
```

a battle axe
(+ a battle axe)
1 + 0 = 1

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 $n-1$. So, if I take the length as n , then the valid positions are 0 to $n-1$, so any position, which is outside this range is an invalid position. So, we could for example, treat n as the default answer, if the string is not present.

So, either it should give us the first position in `s`, where `c` occurs or if there is no occurrence of `c` in `s`, it 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 there are two occurrences of 'a', here and here but because of our 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, the 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 10 saying it is not found. So, how do we write positions 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, it is definitely not found. Remember that the length of the empty string is 0, so I should return 0 by our 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 there 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 check, whether it is the first position. If it is first position, then with respect to this string it is 0, but if not founded, then I would have accumulated the positions I have skipped so far. So, if I do not find it, I add 1 to the position of the character in the remaining part.

So, if I find, for example in 'battle axe' I would first check 'a' with first character of 'battle axe' and I would say that this does not match. So, it will be 1 plus the position of 'a' in 'attle axe' and now it comes back with a 0, so I get 1 plus 0 is equal to 1, which is the correct answer. So the reason we are going through all these functions is we just get some practice in working with these inductive definitions on list strings, which are all basically different versions of the same.

(Refer Slide Time: 12:49)

Example: Counting words

- `wordc` : count the number of words in a string
- Words separated by white space: ' ', '\t', '\n'
 - tab* *newline*

```

whitespace :: Char -> Bool
whitespace ' ' = True
whitespace '\t' = True
whitespace '\n' = True
whitespace _ = False

```

wild card

So, as a final example using strings, let us look at something a little more complex. So, supposing, we want to count the number of words in the string. So we have to define what a word is. So, word by convention is a sequence of characters which 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, you can press the tab button on your keyboard and you get will some number of spaces depending on how the tabs are set and this '\n' is a new line.

So, this is the familiar encoding that many programming languages use. 't' stands for tab, '\n' stands for new line.

'n' stands for new line. So, these are whitespace characters, so we can first define a function, which classifies a single character as whitespace or not. So, all it does is for the three cases which we have defined whitespace we can, we might want to later on add cases, but right now we are only saying ' ', '\t', '\n' as whitespace, everything else is not whitespace and notice that we do not need this value here.

So, we have this wildcard pattern we mentioned last. If you do not require the input in the output, we can just ignore the name and just use this '_' as a wildcard pattern. So, this says everything which is other than these three characters are not punctuations.

(Refer Slide Time: 14:12)

Example: Counting words

- Count white space in a string?

Diagram: A string is divided into words w_1, w_2, \dots, w_n by white spaces. A bracket above w_2 and the space after it is labeled "2 white spaces".

```

wscount :: String -> Int
wscount "" = 0
wscount (c:cs)
  | whitespace c = 1 + wscount cs
  | otherwise    = wscount cs
  
```

- Not enough!
- Consider "abc d"

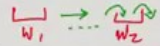
So, the first thing that we could do is assume that, if I have word 1 word 2 and so on. Then, they will each be separated by white space, so I would have in this case, two white spaces, so if I can count the number of white spaces, if I add 1 I should get the number of words. So, the first attempt would 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 whether the first character is a white space or not I add 1 or I just look at the rest.

So, if it is the first character, if c is a white space, then I add 1 and I continue inductively to count the remaining white spaces, otherwise I do not count this, because this is not a white space and I just recount ((Refer Time: 15:01)). So, this is a very straight forward way to count the white spaces, but unfortunately this would give us the wrong answer. Because, when a function does a thing like this, where I have may be four blanks between two words as in "abc d", we could say that there are 4 white spaces and therefore, this constitutes 4 or

5 words when actually there are only two words. So, actually, what is important is not the number of white spaces, but how many times one goes from a word outside and back.

(Refer Slide Time: 15:27)

Example: Counting words



- Keep track of whether we are **inside** a word or **outside** a word
- **Outside a word**: ignore whitespace, but non-whitespace starts a new word
- **Inside a word**: ignore non-whitespace, but whitespace ends current word
- Count the number of times a new word starts

So, one could write a program to 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 another word, so when we are outside a word there maybe many white spaces, but we do not care. Now, when we hit a non white space we transfer to inside a word, now inside a word we will ignore non white space, but when we hit a 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 or 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)

" " == [] char "c" /= [c] variable

Example: Counting words

- New word starts whenever a non-whitespace follows a whitespace—insert initial space to catch first character

```
wordcaux :: String -> Int
wordcaux "" = 0
wordcaux (c:d:ds) =
  | (whitespace c) && not (whitespace d) =
    1 + wordcaux (d:ds)
  | otherwise = wordcaux (d:ds)
```

wordc :: String -> Int
wordc s = wordcaux (' ':s)

Handwritten notes: "The bat" with arrows pointing to 't' and 'b'; "c" with arrows pointing to 'c' and 'x'.

So, new words start whenever you go from a white space, so from a space to say the character c or from a tab to the character x. So, whenever we transfer from one position which have a white space to a position that does not have a white space then by definition a new word 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 a word, so I must say that I go from a non white space to 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 word gets counted correctly. So, the word count, so this should be wordcount, so the function wordcount, which I am going to write on a given string will call this auxiliary function by first adding a blank space before this string. So, now, this makes sure that the very first word is counted correctly, now we are fine, what this says is that, if I have just a single character, then I cannot make a transition.

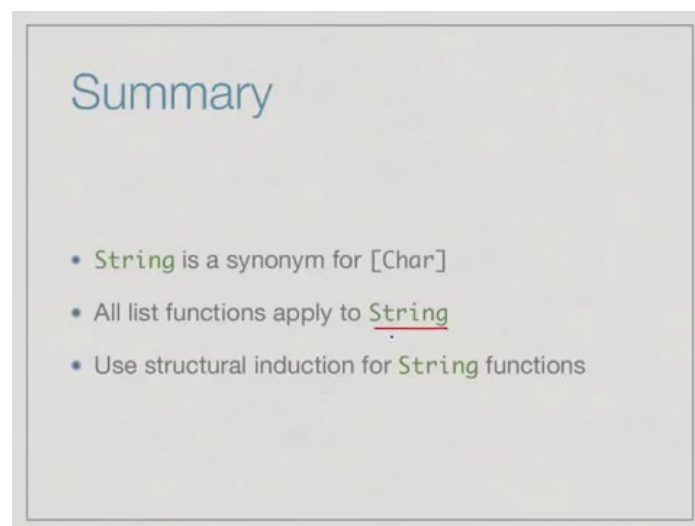
So, I say there are no words, but if have more than one character see notice that I will always call word count aux ‘wordcaux’ with a non empty string, because even if this is empty this whole thing is the string containing one blank. So, I will never call this the wordcaux with an empty string, so I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

If I have two characters I check whether the current character is a white space and the next

character is not white space, this is exactly this transition, if so I add one to the word count, otherwise I continue counting words as before. One important thing here to notice is that when the two empty strings, we said that the `"" == []`. This is an equality. However, `"c"` is not in general equal to `c`, because when I write `"c"` this is the character `c`, where as in `[c]`, `c` is a variable.

So, when I write a function definition like this I need that this `c` stands for any character, if instead by mistake I had written double quote, it would say that this pattern matches precisely when I have a one element character `c`, a single `'c'`. So, therefore, one should be careful in some situations you have to use the list notation even though you are dealing with strings.

(Refer Slide Time: 19:07)



So, to summarize we have looked at the character data type and, what we have observed is that a string, which is a usual unit in which one consumes and manipulate characters is just a sequence of characters. And hence, Haskell provides the type string as a synonym for a list of char and because it is a list all the usual list functions can be applied to string like length or reverse or take or drop and so on.

And also, because these are just lists we can write functions that manipulate lists or manipulate strings using the structural induction, exactly the same way that we use structural induction for a list by looking at the base case which is the empty string and the inductive case which is a non empty string where we separate out first character and the rest.