Computer Science I CSCI141
Operations on Strings Lecture (1/2)

08/26/2013

1 Problem Statement

We want to create functions to compute the length of a string, and the reversal of a string.

2 Solution Design and Analysis

2.1 A Logical Perspective

Strings are not terribly complicated. We can identify two distinct choices, or possibilities: either a string is empty or it is non-empty. A non-empty string must start with a character, and whatever comes after that character must also be a string. Thus we have the following formal definition of a string.

Definition 1 *A* string *is one of the following.*

- *An empty string.*
- *A non-empty string, which has the following parts.*
  - *A head, which is a single character, followed by,*
  - *a tail, which is a* string.

When writing functions that process strings, we need to be able to test for the empty string, to extract the head and the tail, and to construct strings. The following code names those operations[1].

```
def strIsEmpty(s):
  # return whether or not s is the empty string

def strHead(s):
  # return the head of s

def strTail(s):
  # return the tail of s

def strConcat(s1, s2):
  # return the concatenation of s1 and s2
```

1. There are other reasonable approaches to string construction.

2.2 The Structural Recursive Design Pattern

One reason the logical characterization of a string is useful is because there are design patterns that help us write code for structures defined by choices and by parts.

For structures defined by choices, we need to determine which choice we are dealing with. And so code for strings involves `if` and comparison to the empty string.

For structures defined by parts, we need to compute based on the values of some or all of the parts. Specifically for strings, we have those helper functions to get at the parts. Further, the *structural recursive design pattern* stipulates that a function should be called recursively on the recursive part of the structure.

In particular, if we follow this design pattern, a function on strings will have the following form.

```
def f(s):
   if strIsEmpty(s):
       return ...
   else:
       return ... strHead(s) ... f(strTail(s)) ...
```

We read this template as saying to return some value if the string is empty, and otherwise to return a value that is computed using the head[2] of the string and using a recursive call on the tail of the string. With this template in hand, algorithm construction is simplified. We need only ask a couple of questions to fill in the template: What is the answer for the empty string, and, if we know the answer for the tail, how can we fix it to get the answer for the whole string?

2.3 Introduction to Python Strings

A Python *string* is a sequence of characters; the sequence can be written explicitly between two single quotation marks. Here are some examples.

```
>>> 'Hello World'
'Hello World'
>>> ''
''
>>> 'a'
'a'
>>> 'abc'
'abc'
```

Notice that the empty string is a valid string.

Like numbers, we can name strings using variables.

---

2. Strictly speaking, the template allows for the possibility that the head of the string is omitted from the computation. We will see that in the upcoming example.

It is also possible to get at various parts, or components, of a string. One way to do that is by *indexing*, which is written as s`[`n`]`, where s is an expression that denotes a string and n is an expression that denotes an integer. The value of s`[`n`]` is the string that is the character at position n in string s.

```
>>> s = 'Hello World'
>>> s[0]
'H'
>>> s[1]
'e'
>>> s[2]
'l'
>>> s[5]
' '
```

Another way to get at the parts of the string is by *slicing*, which is written s`[`m`:`n`]`, where s is an expression that denotes a string, and m and n are expressions that denote integers; the integers are both optional. The value of s`[`m`:`n`]` is the sub-string that starts with the character at position m and continues up until but not including the character at position n. If m is omitted, the starting character is the first character; if n is omitted, it is the sub-string that goes all the way to the end of the string.

```
>>> s[1:4]
'ell'
>>> s[:5]
'Hello'
>>> s[1:]
'ello World'
```

It is also possible to concatenate strings, or put two together. String concatenation is written using a plus-sign (`+`).

```
>>> 'Hello' + ' World'
'Hello World'
>>> 'a' + 'bc'
'abc'
>>> 'ab' + 'c'
'abc'
>>> 'a' + 'b' + 'c'
'abc'
```

Note that strings are *immutable*; strings cannot be changed using the assignment operator.

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[0] = 'z'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

We can concretely realize the logical perspective in Python. We'll implement the `strIsEmpty` using the double-equal operator (`==`), and we'll implement `strConcat` using the plus operator (`+`).

```
def strHead(s):
    """strHead: String -> String
       Purpose: Compute the head of a string
    """
    return s[0]

def strTail(s):
    """strTail: String -> String
       Purpose: Compute the tail of a string
    """
    return s[1:]
```

2.4 Computing Length

We would like to compute the length of a string[3]. Consider the following example.

```
>>> length('abc')
3
```

Let's start with the template. We need to fill in code for the empty string case. What is the length of the empty string?

To fill in the second case, let's take for granted the recursive call works on the tail. Suppose we're computing the length of the string `'abc'`, and we already know that the length of the tail, `'bc'`, is 2, what can we do with that result to get the length of the whole string?

The answers to these questions lead us to the following code.

```
def length(s):
    """length: String -> Number
       Purpose: Compute the length of a string
    """
    if s == '':
        return 0
    else:
        return 1 + length(strTail(s))
```

3. In fact, Python has a function to do this. Nevertheless, we will write our own.

2.5 Computing the Reversal

We would like to compute the reversal of a string. Consider the following example.

```
>>> reverse('abc')
'cba'
```

Let's start with the template. We need to fill in code for the empty string case. What is the reversal of the empty string?

To fill in the second case, let's take for granted the recursive call works on the tail. Suppose we're computing the reversal of the string `'abc'`, and we already know that the reversal of the tail, `'bc'`, is `'cb'`, what can we do with that result to get the reversal of the whole string?

The answers to these questions lead us to the following code.

```
def reverse(s):
    """reverse: String -> String
       Purpose: Compute the reversal of a string
    """
    if s == '':
        return ''
    else:
        return reverse(strTail(s)) + strHead(s)
```

2.6 Substitution Traces

2.6.1 Example: Length

$$\begin{aligned}
\texttt{length('abc')} &= 1 + \texttt{length('bc')} \\
&= 1 + (1 + \texttt{length('c')}) \\
&= 1 + (1 + (1 + \texttt{length(''')})) \\
&= 1 + (1 + (1 + 0)) \\
&= 1 + (1 + 1) \\
&= 1 + 2 \\
&= 3
\end{aligned}$$

2.6.2 Example: Reverse

$$\begin{aligned}
\texttt{reverse('abc')} &= \texttt{reverse('bc')} + \texttt{'a'} \\
&= (\texttt{reverse('c')} + \texttt{'b'}) + \texttt{'a'} \\
&= ((\texttt{reverse('')} + \texttt{'c'}) + \texttt{'b'}) + \texttt{'a'} \\
&= ((\texttt{''} + \texttt{'c'}) + \texttt{'b'}) + \texttt{'a'} \\
&= (\texttt{'c'} + \texttt{'b'}) + \texttt{'a'} \\
&= \texttt{'cb'} + \texttt{'a'} \\
&= \texttt{'cba'}
\end{aligned}$$

2.7 An Alternative Recursive Approach: Accumulative Recursion

Accumulative recursion is a technique that involves introducing an additional parameter, the accumulator, so as to be able to write the recursive code somewhat differently — typically in a tail recursive style. Introducing this new parameter requires finding a suitable generalization of the original function; it is often difficult to find such a function. We will see such generalizations and their utility for the string functions we've been considering.

2.7.1 Accumulative Length

The generalization for length is the function $\texttt{length2}(s, a) = \texttt{length}(s) + a$. The significance of this equation is that it tells us a lot about how to implement `length2`. Of course, we could simply define `length2` in terms of `length`. Instead, we'll use this equation to directly define `length2` using structural tail recursion. Thus we wish to fill in the following outline.

```
def length2(s, a):
    """length2: String * Number -> Number
       Purpose: Compute length(s) + a
    """
    if s == '':
        return ...
    else:
        return length2(strTail(s), ... strHead(s) ... a ...)
```

For the empty string case, we can plug into the equation to see what the result should be.

$$\texttt{length2}(\texttt{''}, a) = \texttt{length}(\texttt{''}) + a = 0 + a = a$$

For the non-empty string case, the string has the form $h + t$, where h is the head and t is the tail of the string. Now using the generalization definition, we get the following.

$$\begin{aligned}\texttt{length2}(h+t,a) &= \texttt{length}(h+t)+a\\ &= (1+\texttt{length}(t))+a\\ &= (\texttt{length}(t)+1)+a\\ &= \texttt{length}(t)+(1+a)\\ &= \texttt{length2}(t,1+a)\end{aligned}$$

Putting these facts together, we can fill in the outline.

```
def length2(s, a):
    """length2: String * Number -> Number
       Purpose: Compute length(s) + a
    """
    if s == '':
        return a
    else:
        return length2(strTail(s), 1 + a)
```

Further, now we can use `length2` to compute `length`. The generalization definition tells us how: $\texttt{length2}(s,0) = \texttt{length}(s)+0 = \texttt{length}(s)$.

```
def length1(s):
    """length1: String -> Number
       Purpose: Compute length(s) tail recursively
    """
    return length2(s, 0)
```

2.7.2 Accumulative Reverse

The generalization for reverse is the function $\texttt{reverse2}(s,a) = \texttt{reverse}(s)+a$. This equation looks very similar to the previous generalization, but there is an important difference. Only strings should be plugged in for the variable a, and the plus sign refers to string concatenation. Note that string concatenation is *not* commutative[4] (i.e. `'a'+'b'` $\neq$ `'b'+'a'`). We will fill in the following outline.

```
def reverse2(s, a):
    """reverse2: String * String -> String
       Purpose: Compute reverse(s) + a
    """
    if s == '':
        return ...
    else:
        return reverse2(strTail(s), ... strHead(s) ... a ...)
```

4. It turns out that lack of commutativity won't hurt us here and may even help; for other problems it can make finding the tail recursive formulation more difficult.

For the empty string case, we can plug into the equation to see what the result should be.

$$\texttt{reverse2}(\texttt{''}, a) = \texttt{reverse}(\texttt{''}) + a = \texttt{''} + a = a$$

For the non-empty string case, the string has the form $h + t$, where h is the head and t is the tail of the string. Now using the generalization definition, we get the following.

$$\begin{aligned}\texttt{reverse2}(h + t, a) &= \texttt{reverse}(h + t) + a \\ &= (\texttt{reverse}(t) + h) + a \\ &= \texttt{reverse}(t) + (h + a) \\ &= \texttt{reverse2}(t, h + a)\end{aligned}$$

Putting these facts together, we can fill in the outline.

```
def reverse2(s, a):
    """reverse2: String * String -> String
       Purpose: Compute reverse(s) + a
    """
    if s == '':
        return a
    else:
        return reverse2(strTail(s), strHead(s) + a)
```

Further, now we can use `reverse2` to compute `reverse`. The generalization definition tells us how: $\texttt{reverse2}(s, \texttt{''}) = \texttt{reverse}(s) + \texttt{''} = \texttt{reverse}(s)$.

```
def reverse1(s):
    """reverse1: String -> String
       Purpose: Compute reverse(s) tail recursively
    """
    return reverse2(s, '')
```

2.8 A Python Iteration Construct to Replace Tail Recursion

Python provides a statement that can be used in place of tail recursion: the `for` loop. It has the following form.

```
for v in s:
    # statements
```

In the above template, v is a variable and s is an expression that denotes a string[5]. This loop iterates through the string executing the statements once for each character. At the first iteration, v is the first character in the string, at the second iteration, v is the second character in the string, and so on.

5. The `for` loop is quite versatile and can iterate through sequences other than strings.

```
>>> for c in 'abc':
...     print(c)
...
a
b
c
```

This loop might seem even too simple to be useful; however, we can mechanically transform any accumulative tail recursive function to a function that uses a `for` loop.

```
def f1(s):
  return f2(s, Init_Exp)

def f2(s, a):
  if s == '':
       return Ret_Exp
   else:
       return f2(strTail(s), Update_Exp)
```

The above code can be transformed into the following code that computes the same result.

```
def fIter(s):
   a = Init_Exp
   for hd in s:
     a = Update_Exp # where strHead(s) is replaced with hd
   return Ret_Exp
```

For the length function we get the following.

```
def lengthIter(s):
   """lengthIter: String -> Number
      Purpose: Compute length(s) with a for loop
   """
   a = 0
   for hd in s:
     a = 1 + a
   return a
```

And for the reverse function we get the following.

```
def reverseIter(s):
   """reverseIter: String -> String
      Purpose: Compute reverse(s) with a for loop
   """
   a = ''
   for hd in s:
      a = hd + a
   return a
```

3 Testing

To simplify debugging, we test each function *as it is implemented*.

For the `length` function, we wish to test that the string is correct for empty and non-empty strings, up to some small size. For example, in our Python implementation, we use these cases: `''`, `'a'`, `'ab'`, and `'abc'`. Looking at the structure of the cases in the length function, once we've determined that the function works correctly for one or two characters, additional characters simply increase the number of recursive calls. Increasing the number of recursive calls is important; sometimes conceptual errors do not manifest with only one or two recursive calls.

Because both `length` and `reverse` each take the same type of input we will reuse our `length` test cases for `reverse`.

The accumulative functions should have tests involving a non-empty accumulator.