Recursion (continued)

CMPU 101 – Problem Solving and Abstraction

Peter Lemieszewski

Picking up from last week: When is a recursive solution appropriate?

Any time a problem is structured such that

- the solution on larger inputs can be built from the solution on smaller inputs, then
- recursion is appropriate.

The two cases we need to solve

All recursive functions have these two parts:

Base case(s):

What's the simplest case to solve?

(Usually, the "empty" or "null" or "zero" case)

Recursive case(s):

What's the relationship between the current case and the answer to a slightly smaller case?

You should be calling the function you're defining here; this is referred to as a *recursive call*.

Each time you make a recursive call, you must make the input smaller.

Otherwise, we would have a "GNU" case (i.e. endless recursion)!

If your input is a list, you do this by passing the *rest* of the list to the recursive call.

Splitting up a list recursively: First and Rest

```
››› lst = [list: "item 1", "and", "so", "on"]
››› lst.first
"item 1"
››› lst.rest
[list: "and", "so", "on"]
```

First/Rest in my-sum

link(f, r) => f + my-sum(r)

- first of the list is… f
- rest of the list is… my-sum(r)

What if…

… we made a recursive call on the original input list?

link(f, r) => f + my-sum(lst)

- first of the list is… f
- rest of the list is… my-sum(lst)

Let’s try writing another recursive function

Given: a list of numbers…

The function **any-below-10** should return true if any member of the list is less than 10 and false otherwise.

Writing any-below-10

#Start with the test cases first!

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  ...
where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
  any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
  any-below-10([list: 4]) is (4 < 10)
  any-below-10([list: ]) is ...
end
```

Writing any-below-10: base case test case

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  ...
where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
  any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
  any-below-10([list: 4]) is (4 < 10)
  any-below-10([list: ]) is false
end
```

Writing any-below-10: rewrite the recursive tests

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  ...
where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
  any-below-10([list:    1, 4]) is (1 < 10) or any-below-10([list: 4])
  any-below-10([list:       4]) is (4 < 10) or any-below-10([list: ])
  any-below-10([list: ]) is false
end
```

Writing any-below-10: rewrite the recursive tests

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  ...
where:
 any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
 any-below-10([list:    1, 4]) is (1 < 10) or any-below-10([list: 4])
 any-below-10([list:       4]) is (4 < 10) or any-below-10([list: ])
 any-below-10([list: ]) is false
end
```

Writing any-below-10: lastly, the function itself

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10, think of link as meaning detach"
  cases (List) lst:
    | empty => false
    | link(f, r) => (f < 10) or any-below-10(r)

where:
 any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
 any-below-10([list:    1, 4]) is (1 < 10) or any-below-10([list: 4])
 any-below-10([list:       4]) is (4 < 10) or any-below-10([list: ])
 any-below-10([list: ]) is false
end
```

Writing a Recursive Predicate

- Now that we've seen how to write **any-below-10**, we can use the same pattern to implement a higher-order function where we can ask if any item in a list satisfies *some predicate*.
  - "Some predicate": meaning some kind of "generalized or, helper, function"

Writing my-any

```
fun my-any(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for any item in the given list."
  cases (List) lst:
    | empty => false
    | link(f, r) => fn(f) or my-any(fn, r)
  end
End
#Compare with "any-below-10"
```

Compare with "any-below-10"

```
fun my-any(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for any item in the
given list."
  cases (List) lst:
    | empty => false
    | link(f, r) => fn(f) or my-any(fn, r)
  end
End
#Compare with "any-below-10"
```

```
    | empty => false
    | link(f, r) => (f < 10) or any-below-10(r)
```

Writing my-all

```
fun my-all(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for every item
in the given list."
  cases (List) lst:
    | empty => true
    | link(f, r) => fn(f) and my-all(fn, rst)
  end
end
```

Let’s try some practice examples together

BTW This stuff can be *adjective!*

adjectives =

[list: “difficult", “funky"]

Practice Makes ______

```
fun list-len(lst :: List) -> Number:
  doc: "Compute the length of a list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => 1 + list-len(____)
  end
end
```

Practice Makes Perfect

```
fun list-len(lst :: List) -> Number:
  doc: "Compute the length of a list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => 1 + list-len(r)
  end
end
```

Practice Makes ______

```
fun list-product(lst :: List<Number>) ->
Number:
  doc: "Compute the product of all the
numbers in lst"
  cases (List) lst:
    | empty => 1
    | link(f, r) => ____ * list-product(r)
  end
end
```

Practice Makes Perfect

```
fun list-product(lst :: List<Number>) ->
Number:
  doc: "Compute the product of all the
numbers in lst"
  cases (List) lst:
    | empty => 1
    | link(f, r) => f * list-product(r)
  end
end
```

Practice Makes _______

```
fun is-member(lst :: List, item) -> Boolean:
  doc: "Return true if item is a member of lst"
  cases (List) lst:
    | empty => ______
    | link(f, r) =>
      (f == ______) or (is-member(______, ______)
  end
end
```

Practice Makes Perfect

```
fun is-member(lst :: List, item) -> Boolean:
 doc: "Return true if item is a member of lst"
 cases (List) lst:
   | empty => false
   | link(f, r) =>
    (f == item) or (is-member(r, item)
 end
end
```

Next up: fn that adds 1 to every number in a list.

```
fun add-1-all(lst :: List<Number>) -> List<Number>:
  doc: "Add one to every number in the list"
  ...
where: #are all of the tests??!?
end
```

add 1 to every number in a list: test cases

```
fun add-1-all(lst :: List<Number>) -> List<Number>:
  doc: "Add one to every number in the list"
  ...
where:
    add-1-all([list: 3, 1, 4])
      is [list: 4, 2, 5]
     add-1-all([list: 1, 4])
      is [list: 2, 5]
     add-1-all([list: 4])
      is [list: 5]
     add-1-all([list: ]) is [list: ]

end
```

add 1 to every number in a list: alternate format

```
fun add-1-all(lst :: List<Number>) -> List<Number>:
  doc: "Add one to every number in the list"
  ...
where:
  add-1-all(link(3, link(1, link(4, empty))))
             is link(4, link(2, link(5, empty)))
  add-1-all(link(1, link(4, empty)))
             is link(2, link(5, empty))
  add-1-all(link(4, empty))
             is link(5, empty)
  add-1-all(empty) is empty

end
```

add 1 to every number in a list: mod’ed test cases

```
fun add-1-all(lst :: List<Number>) -> List<Number>:
  doc: "Add one to every number in the list"
  ...
where:
add-1-all([list: 3, 1, 4])
    is link(4, add-1-all([list: 1, 4]))
  add-1-all([list: 1, 4])
    is link(2, add-1-all([list: 4]))
  add-1-all([list: 4])
    is link(5, add-1-all([list: ]))
  add-1-all([list: ]) is [list: ]

end
```

add 1 to every number in a list: code

```
fun add-1-all(lst :: List<Number>) -> List<Number>:
  doc: "Add one to every number in the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) => link(f + 1, add-1-all(r))
  end
where:
add-1-all([list: 3, 1, 4])
   is link(4, add-1-all([list: 1, 4]))
  add-1-all([list: 1, 4])
   is link(2, add-1-all([list: 4]))
  add-1-all([list: 4])
   is link(5, add-1-all([list: ]))
  add-1-all([list: ]) is [list: ]

end
```

diff

Something that often trips people up when writing functions like this is the difference between

```
link(x, y)
```

and

```
[list: x, y]
```

What happens if we change the former to the latter?

add 1 to every number in a list: code

```
fun add-1-all(lst :: List<Number>) -> List<Number>:
  doc: "Add one to every number in the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) => link(f + 1, add-1-all(r))
  end
where:
add-1-all([list: 3, 1, 4])
   is link(4, add-1-all([list: 1, 4]))
  add-1-all([list: 1, 4])
   is link(2, add-1-all([list: 4]))
  add-1-all([list: 4])
   is link(5, add-1-all([list: ]))
  add-1-all([list: ]) is [list: ]

end
```

The **map** function we've used works identically, except that it takes a function and applies this function, instead of simply adding 1 to every item in the list.

my-map function:

```
fun my-map(fn :: Function, lst :: List) -> List:
  doc: "Return a list of the results of running fn on every element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) => link(fn(f), my-map(fn, r))
  end
where:
  my-map(lam(i): i + 1 end, [list: 1, 4])
    is [list: 2, 5]
  my-map(lam(i): i + 1 end, [list: 4])
    is [list: 5]
  my-map(lam(i): i + 1 end, [list: ])
    is [list: ]
end
```

Pattern

- We've seen examples of recursive functions and
  - Made them generic by introducing a predicate (function)
- Let's do the same by developing functions:
  - **pos-nums** that returns/selects only positive numbers from a list of numbers.
    - A specific recursive function that we can generalize as…
  - **filter** that returns a list of items where some *predicate* returns true
    - Essentially a "my-filter" recursive function

pos-nums

```
fun pos-nums(lst :: List<Number>) -> List<Number>:
  doc: "Select the positive numbers from lst"
  cases (List) lst:
    | empty => empty
    | link(n, rst) =>
      if n > 0:
        link(n, pos-nums(rst))
      else:
        pos-nums(rst)
      end
  end
where:
  pos-nums([list: ]) is [list: ]
  pos-nums([list: 1]) is [list: 1]
  pos-nums([list: -1]) is [list: ]
  pos-nums([list: 1, -2]) is [list: 1]
  pos-nums([list: -1, 2]) is [list: 2]
  pos-nums([list: 1, -2, -3, -4]) is [list: 1]
  pos-nums([list: -1, 2, -3, -4]) is [list: 2]
  pos-nums([list: 1, -2, 3, 4]) is [list: 1, 3, 4]
end
```

My-filter: with generic predicate (1)

```
fun my-filter(predicate :: Function, lst :: List<Number>) -> List<Number>:
  doc: "Filter a list to only items where predicate returns true"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      if predicate(f):
        link(f, my-filter(predicate, r))
      else:
        my-filter(predicate, r)
      end
  end
where:
 # we can define the predicate in our test case. Let's replicate pos-nums functionality

end
```

My-filter: with generic predicate (2)

```
fun my-filter(predicate :: Function, lst :: List<Number>) -> List<Number>:
  doc: "Filter a list to only items where predicate returns true"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      if predicate(f):
        link(f, my-filter(predicate, r))
      else:
        my-filter(predicate, r)
      end
  end
where:
 # we can define the predicate in our test case. Let's replicate pos-nums functionality
 # we can use lambda for this purpose too: format: lam(x): ??? end
end
```

My-filter: with generic predicate (3)

```
fun my-filter(predicate :: Function, lst :: List<Number>) -> List<Number>:
  doc: "Filter a list to only items where predicate returns true"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      if predicate(f):
        link(f, my-filter(predicate, r))
      else:
        my-filter(predicate, r)
      end
  end
where:
  my-filter(lam(x): x > 0 end, [list: 1, -2, 3, 4]) is [list: 1, 3, 4]
end
```

Even more generic: The List Aggregation Pattern

```
fun ⟨function-name⟩(⟨arguments, incl. lst⟩) -> ⟨return type⟩:
  cases (List) lst:
    | empty => ⟨empty case⟩
    | link(f, r) =>
     ⟨some processing on f⟩
       ⟨combined with⟩
       function-name(r)
  end
end
```

Writing your own recursive list functions

- Here are the procedures for writing your list functions:

1. Write the name, inputs, input types, & output type for the function.
2. Write some examples of what the function should produce and should cover all structural cases:
   a. i.e., empty vs non-empty lists
   b. as well as *interesting* scenarios within the problem.
3. Write out the list aggregation template
4. Implement the function so that it handles the examples correctly

Writing your own recursive list functions

- Here are the procedures for writing your list functions:

1. Write the name, inputs, input types, & output type for the function.
2. Write some examples of what the function should produce and should cover all structural cases:
   a. i.e., empty vs non-empty lists
   b. as well as *interesting* scenarios within the problem.
3. Write out the list aggregation template
4. Implement the function so that it handles the examples correctly

One final recommendation: Don’t skip steps!

Link to code

- pos-nums
- add-1-all
- my-filter

- And, lecture 11 code (any-below-10, any-in-list, all-in-list)

Acknowledgements

- This lecture incorporates material from:
- Kathi Fisler, Brown University,
- Marc Smith, Vassar College
- And, Jonathan Gordon, Vassar College