

CMPU 101 §02 · Computer Science I

Building Lists

10 October 2022



Where are we?

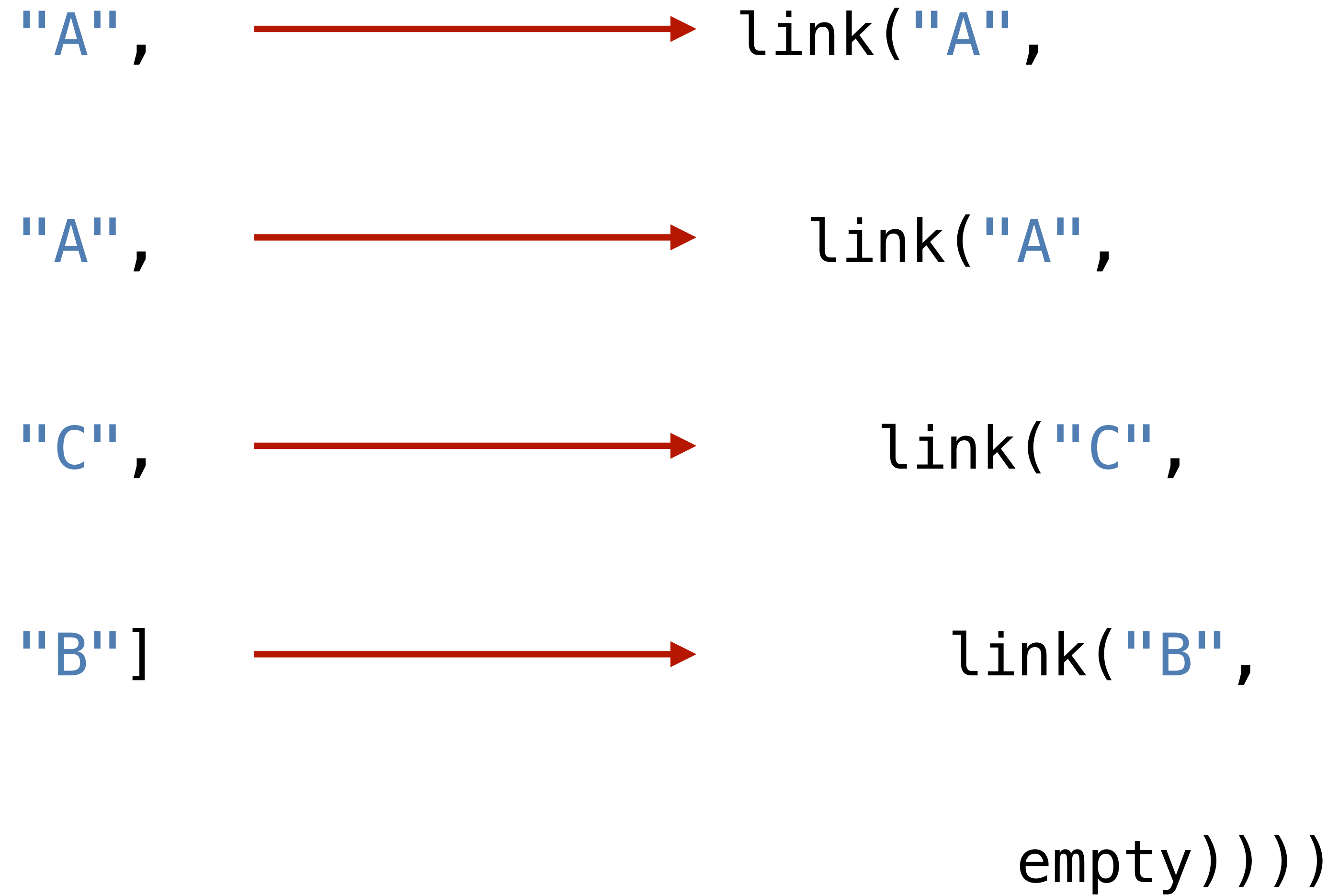
To write our own functions to process a list, item by item, we need to use the true form of a list and think recursively.

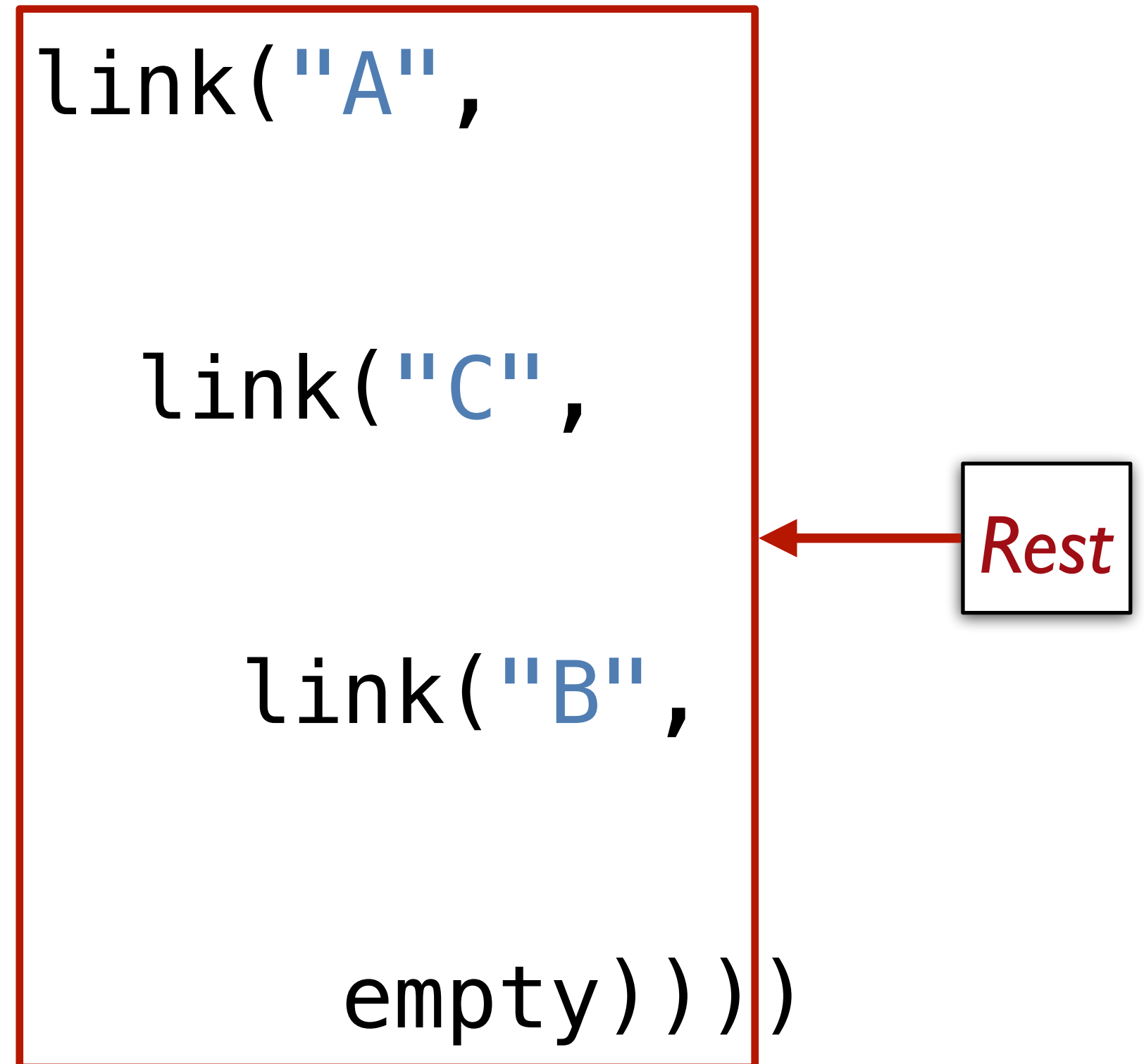
A list is either:

empty

link(*⟨item⟩*, *⟨list⟩*)

[list:





```
cases (List) lst:  
  | empty => ...  
  | link(f, r) => ...  
end
```

The diagram illustrates the variable binding in the second case of the `cases` expression. Two boxes, one labeled *First* and one labeled *Rest*, are positioned below the `link(f, r)` pattern. Red arrows point from the *First* box to the variable `f` and from the *Rest* box to the variable `r`. The variables `f` and `r` in the code are highlighted with red boxes.

All recursive functions have these two parts:

Base case(s):

What's the simplest case to solve?

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*.


```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  cases (List) lst:
    | empty => 0 ← Base case
    | link(f, r) => f + my-sum(r) ← Recursive case
  end
where:
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: ]) is 0
end
```

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Determine whether there are any numbers below 10 in lst"
  cases (List) lst:
    | empty => false
    | link(f, r) =>
      (f < 10) or any-below-10(r)
  end
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
```

```
fun my-any(pred :: Function, lst :: List) -> Boolean:
  doc: "Determine whether any elements of lst satisfy pred"
  cases (List) lst:
    | empty => false
    | link(f, r) => pred(f) or my-any(pred, r)
  end
where:
  my-any(lam(x): x < 10 end, [list: ]) is false
  my-any(lam(x): x < 10 end, [list: 9, 10, 11]) is true
  my-any(lam(x): x < 10 end, [list: 10, 11, 12]) is false
end
```

```
fun my-all(pred :: Function, lst :: List) -> Boolean:
  doc: "Determines whether all elements of lst satisfy pred"
  cases (List) lst:
    | empty => true
    | link(f, r) => pred(f) and my-all(pred, r)
  end
where:
  my-all(lam(x): x < 10 end, [list: ]) is true
  my-all(lam(x): x < 10 end, [list: 7, 8, 9]) is true
  my-all(lam(x): x < 10 end, [list: 9, 10, 11]) is false
end
```

Building lists

add-1-all and **map**

Let's write a function 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"  
  ...  
end
```



```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

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?

The **map** function we've used works identically, except that it takes a function and applies it instead of adding 1 every time.

```
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
```


pos-nums and **filter**

The function **pos-nums** selects only the positive numbers from a list of numbers.

```
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
  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
```

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

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
```

Here are the steps you should take when writing a list function:

- 1 Write the name, inputs, input types, and output type for the function.
- 2 Write some examples of what the function should produce.

The examples should cover all structural cases of the inputs – i.e., empty vs non-empty lists – 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

Code from class:

<https://link/to/come>

