Buildings Lists

13 October 2021
Computer Science Department Open House
TODAY @ 6:30pm GRAD HILL TENT
Drop by to meet the Major's Committee members, CS faculty, and other interested students for some pizza and snacks. Learn about the department, course offerings, and more! Open to any students with an interest in Computer Science.
THERE WILL BE PIZZA!!
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:
  "A",
  link("A",
  "A",
  link("A",
  "C",
  link("C",
  "B"

[link("B", empty)])))]
link("A",
  link("A",
    link("C",
      link("B",
        empty))))

First

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

First  Rest
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: "Compute the sum of elements in lst"
    cases (List) lst:
        | empty => 0 base case
        | link(fst, rst) => fst + my-sum(rst) 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
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(fst, rst) =>
      (fst < 10) or any-below-10(rst)
  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, lst :: List) -> Boolean:

doc: "Determine whether any elements of lst satisfy pred"

cases (List) lst:
  | empty => false
  | link(fst, rst) => pred(fst) or my–any(pred, rst)
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, lst :: List) -> Boolean:

doc: "Determines whether all elements of lst satisfy pred"
cases (List) lst:
  | empty => true
  | link(fst, rst) => pred(fst) and my-any(pred, rst)
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
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: "Adds one to every element of the list"
    ...
end
fun add-1-all(lst :: List<Number>) -> List<Number>:
    doc: "Adds one to every element of 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: "Adds one to every element of 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: "Adds one to every element of 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: "Adds one to every element of 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: "Adds one to every element of the list"
    cases (List) lst:
        | empty => empty
        | link(fst, rst) => link(fst + 1, add-1-all(rst))
    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: "Calls function fn on every member of the list"
  cases (List) lst:
    | empty => empty
    | link(fst, rst) => link(fn(fst), my-map(fn, rst))
  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: "Selects 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
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(fst, rst) =>
            if pred(fst):
                link(fst, my-filter(pred, rst))
            else:
                my-filter(pred, rst)
        end
    end
    where:
        my-filter(lam(x): x > 0 end, [list: 1, -2, 3, 4]) is [list: 1, 3, 4]
The list aggregation pattern
fun \langle \textit{function-name} \rangle(\langle \textit{arguments, incl. lst} \rangle) \rightarrow \langle \textit{return type} \rangle:

cases (\textit{List}) \textit{lst}:
  | \textit{empty} => \langle \textit{empty case} \rangle
  | \textit{link}(\textit{fst}, \textit{rst}) =>
    \langle \textit{some processing on} \, \textit{fst} \rangle
    \langle \textit{combined with} \rangle
    \textit{function-name}(\textit{rst})
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
Approaches to list functions
import lists as L
import math as M

fun sum-of-squares(lst :: List<Number>) -> Number:
    cases (List) lst:
        | empty => 0
        | link(fst, rst) =>
            (fst * fst) + sum-of-squares(rst)
    end
end

fun sum-of-squares(lst :: List<Number>) -> Number:
    M.sum(L.map(lam(x): x * x end, lst))
end
Note that sometimes trying to write a single recursive function is *not* the right approach!

```plaintext
fun my-avg(lst :: List<Number>) -> Number:
  doc: "Computes the average of the numbers in lst"
...
where:
  my-avg([list: 1, 2, 3, 4]) is 10/4
  my-avg([list: 2, 3, 4]) is 9/3
  my-avg([list: 3, 4]) is 7/2
  my-avg([list: 4]) is 4/1
end
```
Note that sometimes trying to write a single recursive function is *not* the right approach!

```python
fun my-avg(lst :: List<Number>) -> Number:
    doc: "Computes the average of the numbers in lst"
    M.sum(lst) / L.length(lst)
where:
    my-avg([list: 1, 2, 3, 4]) is 10/4
    my-avg([list: 2, 3, 4]) is 9/3
    my-avg([list: 3, 4]) is 7/2
    my-avg([list: 4]) is 4/1
end
```
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Marc Smith, Vassar College
Doug Woos, Brown University