CMPU 101 § 04/05 · Problem-Solving and Abstraction

Recursive Functions

13 October 2021

Where are we?

number-grade	letter-grade
98	"A"
100	"A"
74	"C"
84	"B"

```
[list:
  "A",
  "A",
  "C",
  "B"]
```

A list is either:

```
empty
link(⟨item⟩, ⟨list⟩)
```

A list of one item, e.g.,

```
[list: "A"],
```

is really a link between an item and the empty list:

```
link("A", empty)
```

```
[list:
  "A",          link("A",
  "A",            link("A",
  "C",              link("C",
  "B"]                link("B",
                        empty))))
```

Is **`link(3, 4)`** a valid list?

We've seen convenient functions we can use to work with lists:

```
>>> import lists as L

>>> lst = [list: "a", "b", "c"]

>>> L.map(lam(i): "item-" + i end, lst)
[list: "item-a", "item-b", "item-c"]

>>> L.filter(lam(i): not(i == "a") end, lst)
[list: "b", "c"]

>>> L.any(lam(i): i == "a" end, lst)
true

>>> L.all(lam(i): i == "a" end, lst)
false
```

But 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*.

Recursion is a technique that involves defining a solution or structure using itself as part of the definition.

```
fun my-sum(lst :: List<Number>) -> Number:
  ...



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 my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  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 my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  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
```

__cases__ is a special form of conditional that we use to ask "which __shape__ of data do I have?"

Denotes the output of a function

```
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  end
where:
  my-s        1, 4]) is 3 + my-sum([list: 1, 4])
  my-s        1, 4]) is 1 + my-sum([list:    4])
  my-s           4]) is 4 + my-sum([list:     ])
  my-s            ]) is 0
end
```

Marks the expression to evaluate if the data has the shape on the left.

```
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  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
```

When we call this function, it evaluates as:

```
my-sum(link(3, link(1, link(4, empty))))
3 + my-sum(link(1, link(4, empty)))
3 + 1 + my-sum(link(4, empty))
3 + 1 + 4 + my-sum(empty)
3 + 1 + 4 + 0
```

Practice designing recursive functions

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

We've already seen a higher-order function that lets us do this easily:

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  L.any(lam(x): x < 10 end, lst)
end
```

This is how you *should* write this function – higher-order functions like **any** are great!

We'll implement it using recursion just for practice. After we've done that, we'll be able to see how **any** is actually implemented!

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  ...
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
```

What goes here?

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  ...
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
```

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  ...
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
```

```
fun any-below-10(lst :: List<Number>) -> Boolean:
  ...
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 any-below-10(lst :: List<Number>) -> Boolean:
  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
```

Now that we've seen how to write **`any-below-10`**, we can use the same pattern to implement our own version of **`any`**.

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

```
fun my-all(pred, lst :: List) -> Boolean:
  cases (List) lst:
    | empty => true
    | link(fst, rst) => pred(fst) and my-all(pred, rst)
  end
end
```

Thinking recursively

Any time a problem is structured such that the solution on larger inputs can be built from the solution on smaller inputs, recursion is appropriate.

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 recursive-function(lst :: List) -> ...:
  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ... recursive-function(r) ...

  end
end
```

Base case

Recursive case

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

If your input is a list, you pass the ***rest*** of the list to the recursive call.

```
link("A",
```

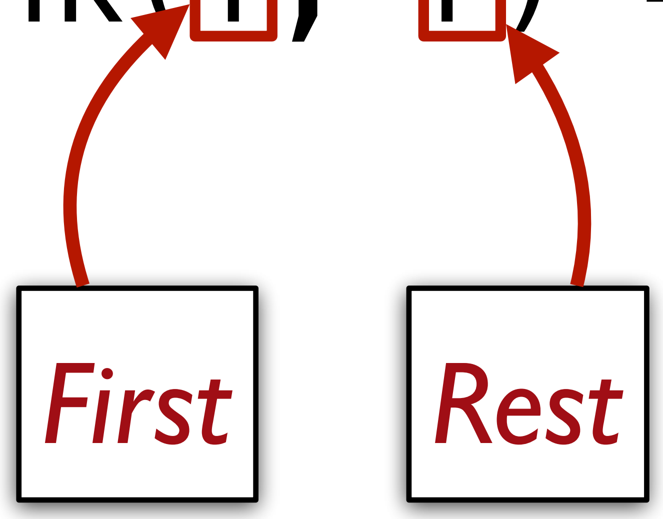
First

```
link("A",
  link("C",
    link("B",
      empty))))
```

Rest

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

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



What happens if we *don't* make the input smaller?

```
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  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 my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(lst)
  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
```

Recursive call on the original input list

When we call this function, it evaluates as:

```
my-sum(link(3, link(1, link(4, empty))))
3 + my-sum(link(3, link(1, link(4, empty))))
3 + 3 + my-sum(link(3, link(1, link(4, empty))))
3 + 3 + 3 + my-sum(link(3, link(1, link(4, empty))))
3 + 3 + 3 + 3 + my-sum(link(3, link(1, link(4, empty))))
...
```

This isn't going to end well.

When a recursive function never stops calling itself, it's called *infinite recursion*.

Wrap-up practice

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

Final note

Lists, recursion, and **cases** syntax are not easy concepts to grasp separately, much less all together in a short time.

Don't feel frustrated if it takes a little while for these to make sense. Give yourself time, be sure to practice working in Pyret, and ask questions.

Acknowledgments

This lecture incorporates material from:

Jonathan Gordon, Vassar College

Kathi Fisler, Brown University

Doug Woos, Brown University