Problem Solving and Abstraction (CMPU 101)

Tom Ellman

Lecture 12

The Secret Nature of Lists

Writing our input as [list: 3, 1, 4] hides the truth.

It’s just a shorthand for the real structure of a list.

In its secret heart, Pyret knows there are only two ways of making a list:

The value: **empty**.

Using the **link** function to add an element to the beginning.

When we write an expression like:

```
[list: 3, 1, 4]
```

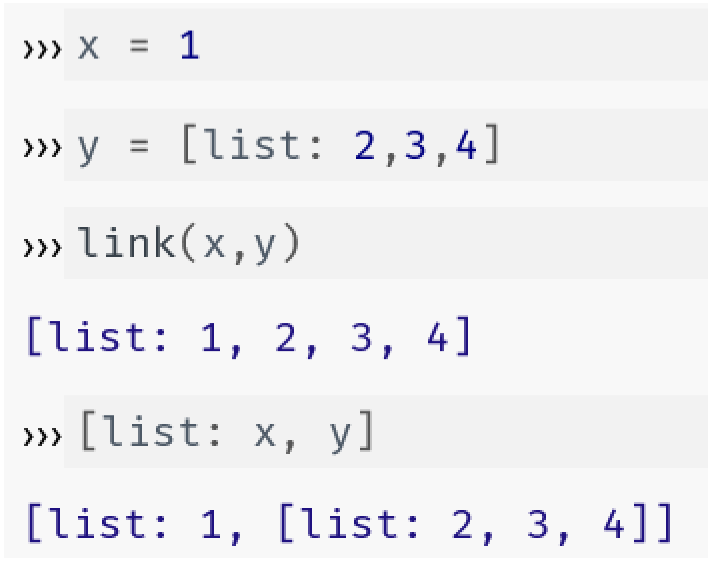
Pyret translates it into this:

```
link(3,
  link(1,
    link(4, empty)))
```

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

link(x, y)
and
[list: x, y]

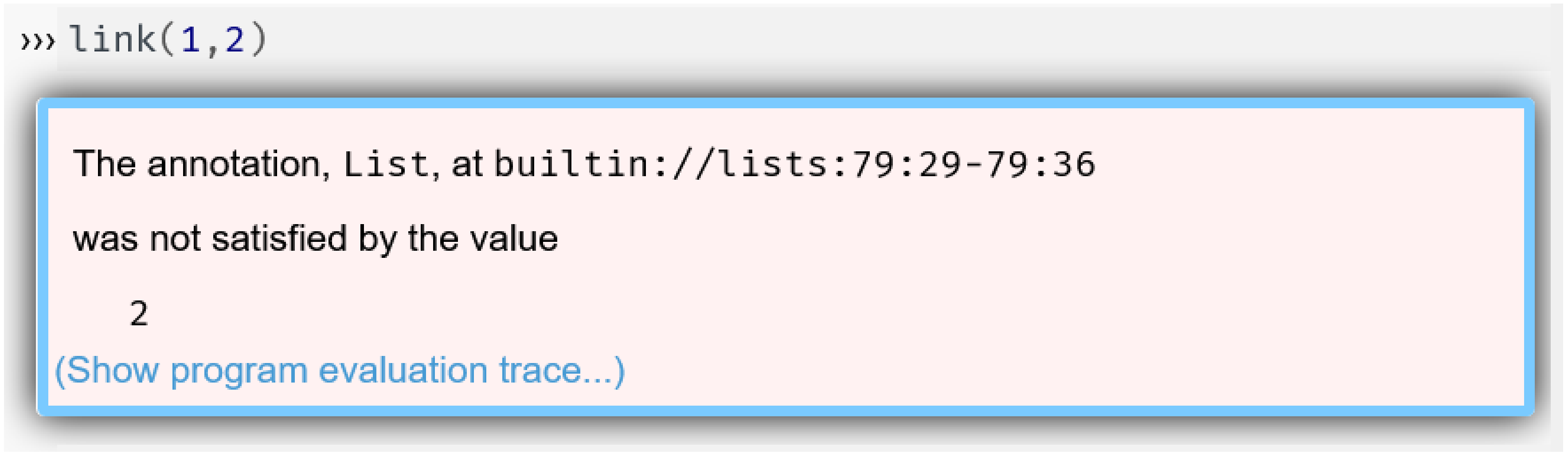
What happens if we change the former to the latter?



```
››› x = 1
››› y = [list: 2,3,4]
››› link(x,y)
[list: 1, 2, 3, 4]
››› [list: x, y]
[list: 1, [list: 2, 3, 4]]
```

The second argument of link must be a (possibly empty) list.

link(<anything>, <list>)



```
››› link(1,2)
```

The annotation, `List`, at `builtin://lists:79:29-79:36`

was not satisfied by the value

    2

(Show program evaluation trace...)

Thinking Recursively

Recursion is appropriate, any time a problem can be split into parts, one of which is a smaller version of the original problem.

```
sum(link(3,
         link(1,
           link(4, empty)))
         ⇩

3 + sum(link(1,
             link(4, empty)))
```

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

Recursion

- 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 problem and the answer to a slightly smaller problem?
  - You should be calling the function you’re defining here; this is referred to as a recursive call.

```
fun rec-fun(lst :: List<Number>) -> Number:
 cases (List) lst:
  | empty =>   <------------------------------------  Base Case

        # Expression giving value of rec-fun(empty)


 | link(first, rest) =>   <-------------------------  Recursive
                                                       Case
        # Expression giving value of rec-fun(link(first,rest))
        # in terms of first and rec-fun(rest).


 end
end
```

all-below

- Given:
  - a list (lst) of numbers
  - a number (bnd)
- Return:
  - true if all numbers on lst are below bnd
  - false otherwise.

Examples

```
fun all-below(lst :: List<Number>, bnd :: Number) -> Boolean:

  ...?...

where:
  all-below(empty, 0) is true
  all-below([list: 1,2,3,4,5], 5) is false
  all-below([list: 1,2,3,4,5], 6) is true
end
```

Notice that all-below(empty,0) is true.   Why?

```
fun all-below(lst :: List<Number>, bnd :: Number) -> Boolean:
 cases (List) lst:
    | empty => true
    | link(first, rest) => (first < bnd) and all-below(rest,bnd)
 end
where:
  all-below(empty, 0) is true
  all-below([list: 1,2,3,4,5], 5) is false
  all-below([list: 1,2,3,4,5], 6) is true
end
```

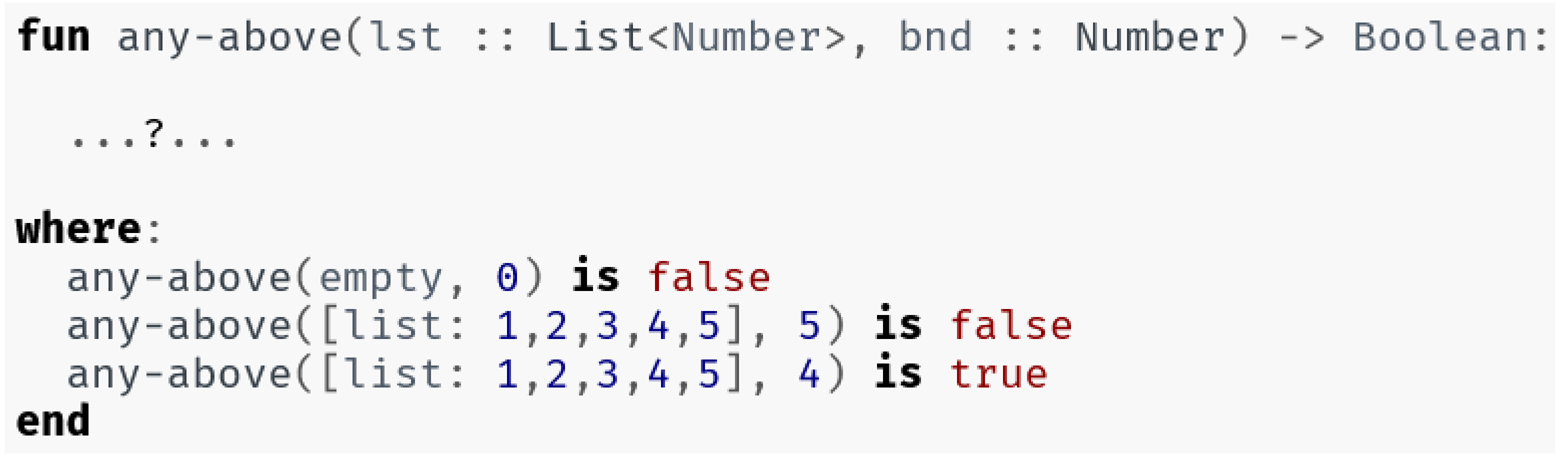
Notice that all-below is true if lst is empty, regardless of bnd.

In this case we cay that all-below is vacuously true!

any-above

- Given:
  - a list (lst) of numbers
  - a number (bnd)
- Return:
  - true if at least one number on lst is above bnd.
  - false otherwise.

Examples

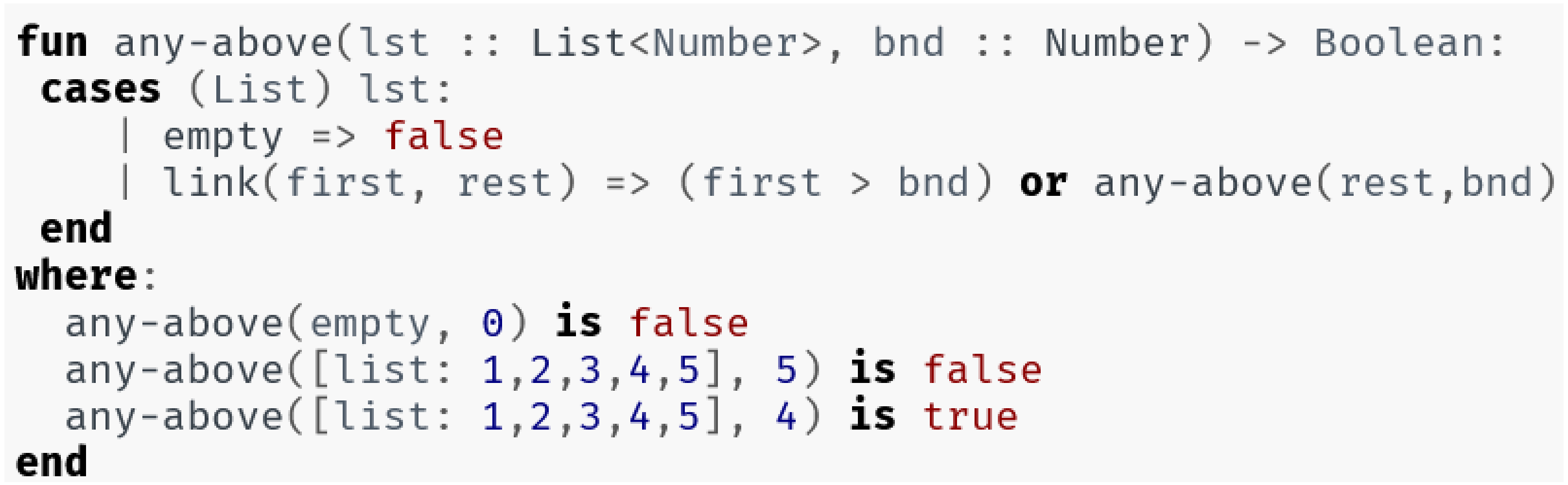


```
fun any-above(lst :: List<Number>, bnd :: Number) -> Boolean:

  ...?...

where:
  any-above(empty, 0) is false
  any-above([list: 1,2,3,4,5], 5) is false
  any-above([list: 1,2,3,4,5], 4) is true
end
```

Notice that any-above(empty, 0) is false. Why?



```
fun any-above(lst :: List<Number>, bnd :: Number) -> Boolean:
 cases (List) lst:
    | empty => false
    | link(first, rest) => (first > bnd) or any-above(rest,bnd)
 end
where:
  any-above(empty, 0) is false
  any-above([list: 1,2,3,4,5], 5) is false
  any-above([list: 1,2,3,4,5], 4) is true
end
```

Notice that any-above is false if lst is empty, regardless of bnd.

In this case we say that any-above is vacuously false.

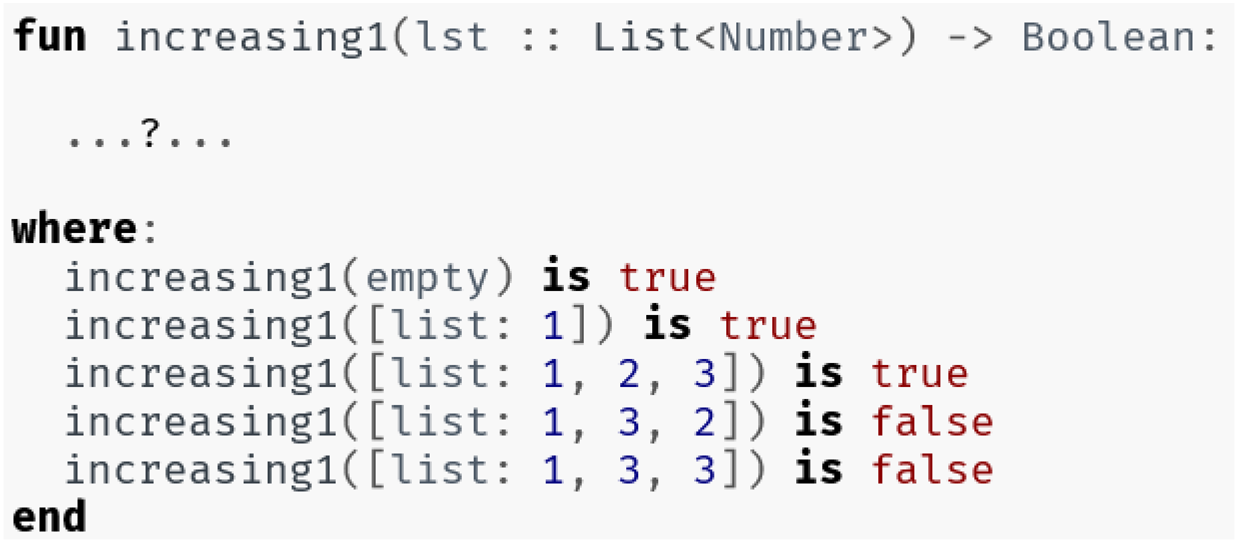
increasing

- Given a list of numbers:

- Return:
  - true if each number with a predecessor is larger than the its predecessor.

  - false if at least one number has a predecessor and is not larger than its predecessor.

Examples



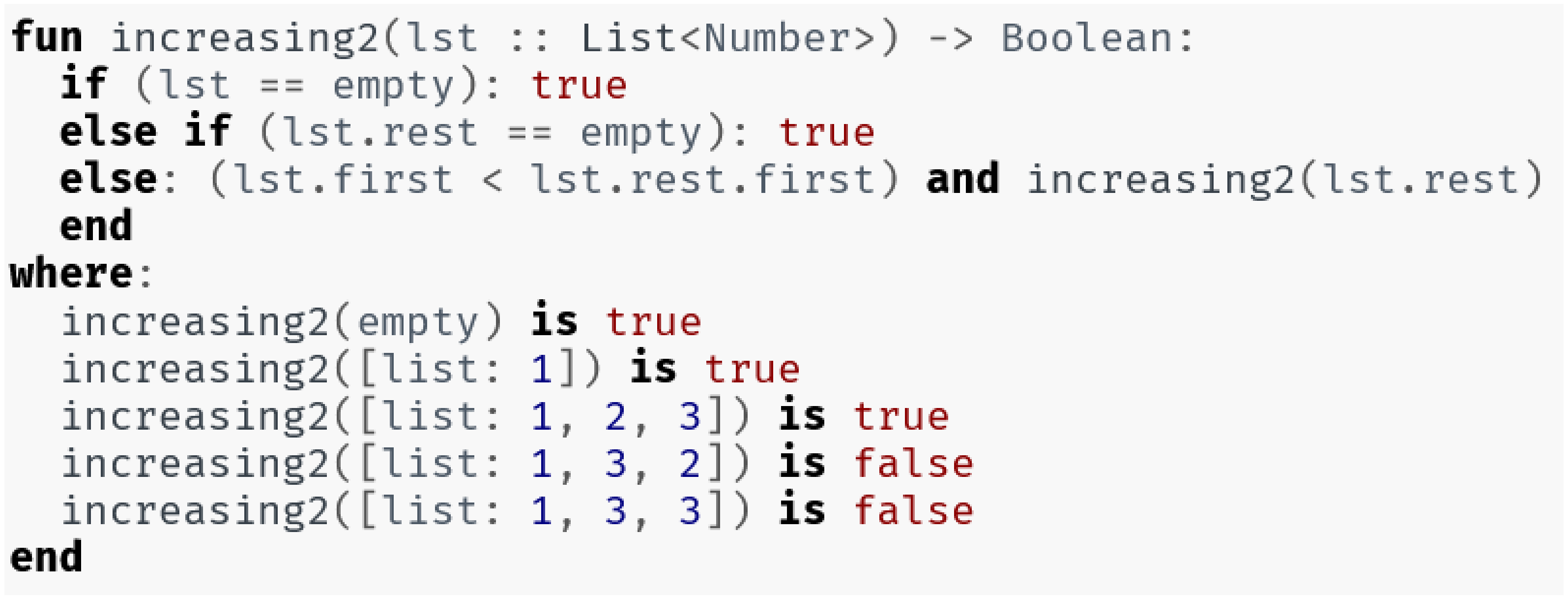
```
fun increasing1(lst :: List<Number>) -> Boolean:

  ...?...

where:
  increasing1(empty) is true
  increasing1([list: 1]) is true
  increasing1([list: 1, 2, 3]) is true
  increasing1([list: 1, 3, 2]) is false
  increasing1([list: 1, 3, 3]) is false
end
```

Why must the first two tests come out true?

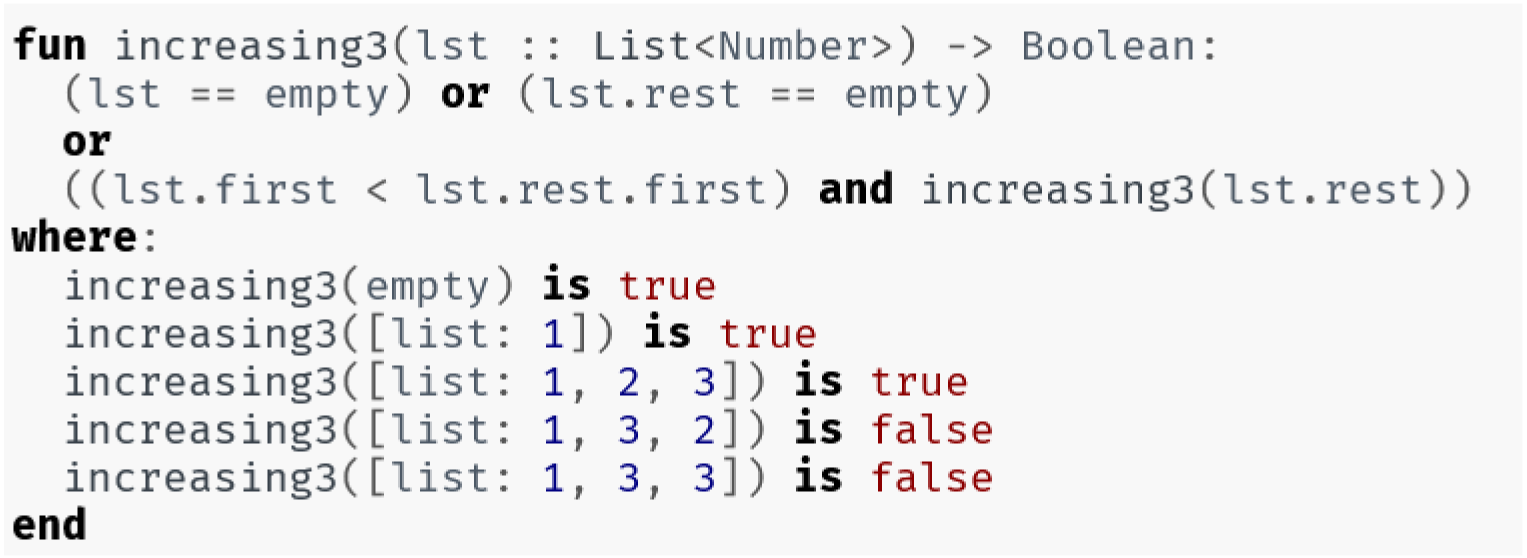
```
fun increasing1(lst :: List<Number>) -> Boolean:
  cases (List) lst:
  | empty => true
  | link(first, rest)
    =>
    if (rest == empty): true
      else: (first < rest.first) and increasing1(rest)
    end
  end
where:
  increasing1(empty) is true
  increasing1([list: 1]) is true
  increasing1([list: 1, 2, 3]) is true
  increasing1([list: 1, 3, 2]) is false
  increasing1([list: 1, 3, 3]) is false
end
```



```
fun increasing2(lst :: List<Number>) -> Boolean:
  if (lst == empty): true
  else if (lst.rest == empty): true
  else: (lst.first < lst.rest.first) and increasing2(lst.rest)
  end
where:
  increasing2(empty) is true
  increasing2([list: 1]) is true
  increasing2([list: 1, 2, 3]) is true
  increasing2([list: 1, 3, 2]) is false
  increasing2([list: 1, 3, 3]) is false
end
```

In this definition, we refrain from using a cases statement and rely on if-else instead.

This works, but it’s cumbersome and hard to read.



```
fun increasing3(lst :: List<Number>) -> Boolean:
  (lst == empty) or (lst.rest == empty)
  or
  ((lst.first < lst.rest.first) and increasing3(lst.rest))
where:
  increasing3(empty) is true
  increasing3([list: 1]) is true
  increasing3([list: 1, 2, 3]) is true
  increasing3([list: 1, 3, 2]) is false
  increasing3([list: 1, 3, 3]) is false
end
```

In this definition, we refrain from using a cases statement and rely on logical operators or & and.

This works, but it’s also cumbersome and hard to read.

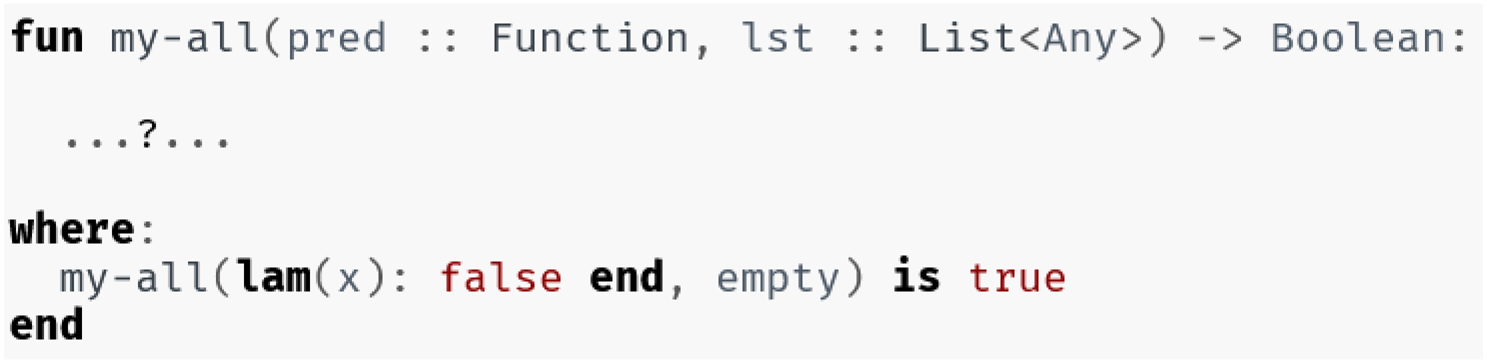
my-all

- Given a predicate (pred) and a list (lst).

- Return:
  - true if pred returns true for each element of lst.

  - false if pred returns false for at least one element of lst.

Examples



```
fun my-all(pred :: Function, lst :: List<Any>) -> Boolean:

  ...?...

where:
  my-all(lam(x): false end, empty) is true
end
```

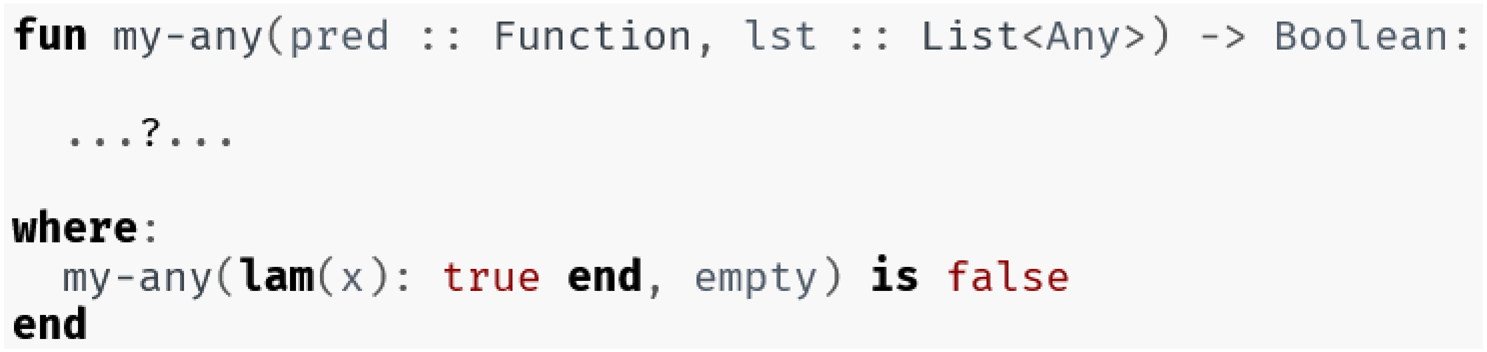
How can my-all be true on the empty list, if the lambda expression always returns false?

```
fun my-all(pred :: Function, lst :: List<Any>) -> Boolean:
  cases (List) lst:
    | empty => true
    | link(first,rest) => pred(first) and my-all(pred,rest)
  end
where:
  my-all(lam(x): false end, empty) is true
end
```

my-any

- Given a predicate (pred) and a list (lst).
- Return:
  - true if pred returns true for at least one element of lst.
  - false if pred returns false for each element of lst.

Examples



```
fun my-any(pred :: Function, lst :: List<Any>) -> Boolean:

  ...?...

where:
  my-any(lam(x): true end, empty) is false
end
```

How can my-all be false on the empty list, if the lambda expression always returns true?

```
fun my-any(pred :: Function, lst :: List<Any>) -> Boolean:
  cases (List) lst:
    | empty => false
    | link(first,rest) => pred(first) or my-all(pred,rest)
  end
where:
  my-any(lam(x): true end, empty) is false
end
```