Problem Solving and Abstraction (CMPU 101)

Tom Ellman
Lecture 11
How would we write a function that takes a list of numbers and returns its sum?
fun my-sum(lst :: List<Number>) -> Number:
   ...
end
where:
   my-sum([list: 3, 1, 4]) is 3 + 1 + 4
end
fun my-sum(lst :: List<Number>) -> Number:
  ...
end

where:
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
  my-sum([list: 1, 4]) is 1 + 4
end
fun my-sum(lst :: List<Number>) -> Number:
   ...
end

where:
   my-sum([list: 3, 1, 4]) is 3 + 1 + 4
   my-sum([list: 1, 4]) is 1 + 4
   my-sum([list: 4]) is 4
end
fun my-sum(lst :: List<Number>) -> Number:
    ...
end

where:
    my-sum([list: 3, 1, 4]) is 3 + 1 + 4
    my-sum([list: 1, 4]) is 1 + 4
    my-sum([list: 4]) is 4
    my-sum([list: ]) is ...?...
end
The empty list can be written both as:

\[ \text{list: } \]

and, a bit less awkwardly, as:

\text{empty}
fun my-sum(lst :: List<Number>) -> Number:
    ...
end

where:
    my-sum([list: 3, 1, 4]) is 3 + 1 + 4
    my-sum([list: 1, 4]) is 1 + 4
    my-sum([list: 4]) is 4
    my-sum([list: ])) is ...?...
end

Consider:
    my-sum(L.append(empty, [list: 1 2 3]))

Shouldn’t it be the same as:
    my-sum(empty) + my-sum([List: 1 2 3])
fun my-sum(lst :: List<Number>) -> Number:

... 
end

where:

my-sum([list: 3, 1, 4]) is 3 + 1 + 4
my-sum([list: 1, 4]) is 1 + 4
my-sum([list: 4]) is 4
my-sum([list: ]) is 0
end

Consider:

my-sum(L.append(empty, [list: 1 2 3]))

Shouldn’t it be the same as:

my-sum(empty) + my-sum([List: 1 2 3])
The Secret Nature of Lists

Writing our input as [list: 3, 1, 4] hide 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))))
Consider a non-empty list like:

\[ \text{[list: } \langle \text{first} \rangle \ldots ?\ldots \text{ ]} \]

Pyret translates it into something like this:

\[ \text{link(} \langle \text{first} \rangle , \langle \text{rest} \rangle ) \]

Where:

- \langle \text{first} \rangle is any number.
- \langle \text{rest} \rangle could be any list of numbers.

What can we say about: \text{my-sum(link(} \langle \text{first} \rangle , \langle \text{rest} \rangle \text{))} ?

The sum is: \langle \text{first} \rangle + \text{my-sum(} \langle \text{rest} \rangle \text{).}
fun my-sum(lst :: List<Number>) -> Number:
  ...
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 (List) lst:
  | empty => ... ? ... # Result if lst is empty.
  | link(first,rest) => ... ? ... # Result if lst is not empty.
end

A *cases* expression is like an *if expression*. If the list is empty, do one thing. If it's a link, do another thing.
fun my-sum(lst :: List<Number>) -> Number:
cases (List) lst:
  | empty => ... ? ...
  | link(first, 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

A cases expression is like an if expression. If the list is empty, do one thing. If it's a link, do another thing.
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:

\[
\text{my-sum(link(3, link(1, link(4, empty)))))}
\]

\[
3 + \text{my-sum(link(1, link(4, empty)))}
\]

\[
3 + 1 + \text{my-sum(link(4, empty))}
\]

\[
3 + 1 + 4 + \text{my-sum(empty)}
\]

\[
3 + 1 + 4 + 0
\]
When we call my-sum, the result it returns depends on other calls to my-sum!
Can this really work?
Yes, but!

The sequence of recursive invocations of:
my-sum(<non-empty-list>)

must eventually lead to an invocation of
my-sum(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

Notice that my-sum(lst) leads to my-sum(rest), where rest is one shorter than lst. So eventually we reach my-sum(empty) which is just zero.
Now that we have a definition of `my-sum`:

```plaintext
fun my-sum(lst :: List<Number>) -> Number:
cases (List) lst:
  | empty  =>  0
  | link(first, rest) => first + my-sum(rest)
end
where:
  my-sum([list: ]) is 0
  my-sum([list: 3, 1, 4]) is 8
end
```

Maybe we can define a similar, and perhaps easier function: `my-length`. 
fun my-length(lst :: List<Any>) -> Number:
cases (List) lst:
    | empty => ...?...
    | link(first, rest) => ...?...
end
where:
  my-length([list: 3, 1, 4]) is 1 + my-length ([list: 1, 4])
  my-length([list: 1, 4]) is 1 + my-length ([list: 4])
  my-length([list: 4]) is 1 + my-length ([list: ])
  my-length([list: ]) is 0
end
fun my-length(lst :: List<Any>) -> Number:
cases (List) lst:
    | empty => 0
    | link(first, rest) => 1 + my-length(rest)
end
where:
my-length([list: 3, 1, 4]) is 1 + my-length ([list: 1, 4])
my-length([list: 1, 4])     is 1 + my-length ([list: 4])
my-length([list: 4])       is 1 + my-length ([list: ])
my-length([list: ])        is 0
end
Now that we have a definition of **my-sum**:

```haskell
fun my-sum(lst :: List<Number>) -> Number:
cases (List) lst:
  | empty => 0
  | link(first, rest) => first + my-sum(rest)
end
where:
  my-sum([list: ]) is 0
  my-sum([list: 3, 1, 4]) is 8
end
```

Maybe we can define a similar, and perhaps easier function: **my-product**.
fun my-product(lst :: List<Number>) -> Number:
cases (List) lst:
    | empty => ... ? ...
    | link(first, rest) => ... ? ...
end

where:
my-product([list: 3, 1, 4]) is 3 * 1 * 4
my-product([list: 1, 4]) is 1 * 4
my-product([list: 4]) is 4
my-product([list: ]) is ...?...
end

Consider:
my-product(L.append(empty, [list: 1 2 3]))
Shouldn’t it be the same as:
my-product(empty) * my-product([List: 1 2 3])
fun my-product(lst :: List<Number>) -> Number:
cases (List) lst:
  | empty => ... ? ...
  | link(first, rest) => ... ? ...
end
where:
my-product([list: 3, 1, 4]) is 3 * my-product([list: 1, 4])
my-product([list: 1, 4]) is 1 * my-product([list: 4])
my-product([list: 4]) is 4 * my-product([list: ])
my-product([list: ]) is 1
end

Consider:
  my-product(L.append(empty, [list: 1 2 3]))
Shouldn’t it be the same as:
  my-product(empty) * my-product([List: 1 2 3])
fun my-product(lst :: List<Number>) -> Number:
cases (List) lst:
  | empty => 1
  | link(first,rest) => my-product(rest)
end

where:
my-product([list: 3, 1, 4]) is 3 * my-product([list: 1, 4])
my-product([list: 1, 4]) is 1 * my-product([list: 4])
my-product([list: 4]) is 4 * my-product([list: ])
my-product([list: ]) is 1
end