Problem Solving and Abstraction
(CMPU 101)

Tom Ellman
Lecture 13
fun my-fun(lst :: List<ElementType>) -> ResultType:
cases (List) lst:
  | empty => <Value of my-fun(empty)>
  | link(first, rest)
    =>
    first <Operation> my-fun(rest)
    or
    <Function>(first, my-fun(rest))
    or
    <Expression using first and my-fun(rest)>
end
Writing my-sum using (First)/Rest Recursion Template

<Value of my-fun(empty)> replaced by 0

<Operation> replaced by +

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

• Given:
  A list (lst) of numbers.

• Return:
  A new list obtained by doubling each element of lst.
Tests for the base case (lst is empty) and the recursive case (lst is not empty).
fun double-all1(lst :: List<Number>) -> List<Number>:
    cases (List) lst:
        | empty => empty
        | link(first,rest) => link(2 * first, double-all1(rest))
    end

where:
    double-all1(empty) is empty
    double-all1([list: 1,2,3]) is [list: 2,4,6]
end
Writing **double-all1** using (First)/Rest Recursion Template

<Value of my-fun(empty)> replaced by empty

<Expression> replaced by link(2 * first, double-all(rest))

fun double-all1(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(first,rest) => link(2 * first, double-all1(rest))
  end
end
We could have implemented double-all using L.map

```scala
fun double-all2(lst :: List<Number>) -> List<Number>:
  L.map(lam(n): 2 * n end, lst)
where:
  double-all2(empty) is empty
  double-all2([list: 1,2,3]) is [list: 2,4,6]
end
```

Here we use L.map with a lambda expression that takes a number n as parameter and returns 2 * n.
Could we have written our own version of \texttt{map}?

- **Given:**
  - A list (\texttt{lst}) of some \texttt{type1}.
  - A function (\texttt{fn}) from \texttt{type1} to \texttt{type2}.

- Return a new list that results from applying \texttt{fn} to each element of \texttt{lst}.
fun my-map(fn :: Function, lst :: List<Any>) -> List<Any>:

...?....

where:
    my-map(lam(n): 2 * n end, empty) is empty
    my-map(lam(n): n * n end, [list: 1,2,3]) is [list: 1,4,9]
end
fun my-map(fn :: Function, lst :: List<Any>) -> List<Any>:
  cases (List) lst:
    | empty => empty
    | link(first,rest) => link(fn(first),my-map(fn,rest))
  end
where:
  my-map(lam(n): 2 * n end, empty) is empty
  my-map(lam(n): n * n end, [list: 1,2,3]) is [list: 1,4,9]
end
collect-above

• Given:
  A list (lst) of numbers.
  A number (bnd).

• Return:
  A list of all members of lst that are greater than bnd.
fun collect-above(lst :: List<Number>, bnd :: Number) -> List<Number>:

...?...

where:
  collect-above([list:], 2) is [list:]
  collect-above([list: 1, 2, 3, 4, 5], 2) is [list: 3, 4, 5]
end
fun collect-above(lst :: List<Number>, bnd :: Number) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(first, rest)
      =>
      if (first > bnd):
        link(first, collect-above(rest, bnd))
      else:
        collect-above(rest, bnd)
  end
end

where:
  collect-above([list:], 2) is [list:]
  collect-above([list: 1,2,3,4,5], 2) is [list: 3, 4, 5]
end
filter-above

• Given:
  A list (\textit{lst}) of numbers.
  A number (\textit{bnd}).

• Return:
  A list of all members of \textit{lst} that are \textbf{not} greater than \textit{bnd}. 
The definition of `filter-above` differs from `collect above` by interchanging the if and else branches.

```plaintext
fun filter-above(lst :: List<Number>, bnd :: Number) -> List<Number>:
cases (List) lst:
    | empty => [list: ]
    | link(first, rest) =>
        if (first > bnd):
            filter-above(rest, bnd)
        else:
            link(first, filter-above(rest, bnd))
end
where:
    filter-above([list:], 3) is [list: ]
    filter-above([list: 1,2,3,4,5], 3) is [list: 1,2,3]
end
```
my-collect

• Generalize collect-above.

• Don’t check each element with a numeric bound.

• Instead check each element with a predicate.
Aside from changing the function name and parameters, we need only to replace a comparison of numbers with the application of a predicate.
my-filter

• Generalize filter-above.

• Don’t check each element with a numeric bound.

• Instead check each element with a predicate.

Try it yourself!
sum-of-squares

• Given a list (lst) of numbers.

• Return the sum of the squares of each element of lst.
fun sum-of-squares(lst :: List<Number>) -> Number:

  ...

where:
  sum-of-squares(empty) is 0
  sum-of-squares([list: 1,2,3]) is 14
end
fun sum-of-squares(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(fst, rst) => (fst * fst) + sum-of-squares(rst)
  end
where:
  sum-of-squares(empty) is 0
  sum-of-squares([list: 1,2,3]) is 14
end
fun sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(fst, rst) => fst + sum(rst)
  end
where:
  sum(empty) is 0
  sum([list: 1,2,3]) is 6
end

fun ssq1(lst :: List<Number>) -> Number:
  sum(L.map(lam(n): n * n end, lst))
where:
  ssq1(empty) is 0
  ssq1([list: 1,2,3]) is 14
end
my-fold(fn,b,lst)

• A generalization of every recursive function that we have written so far.

• Given:
  – A list (lst)
  – A value (b) for the base case.
  – A function (fn) that combines:
    
    first and my-fold(fn,b,rest)
    
    as
    
    fn(first,my-fold(fn,b,rest)
Visualizing my-fold\((fn,b,lst)\)

\[
\text{link}(1, \\
\quad \text{link}(2, \\
\quad \quad \text{link}(3, \text{empty}))) \quad \rightarrow \quad \text{fn}(1, \\
\quad \quad \text{fn}(2, \\
\quad \quad \quad \text{fn}(3, b)))
\]

Notice that \text{fn} replaces \text{link} and \text{b} replaces \text{empty}.

Unfortunately the \text{fold} function of Pyret reverses the order of the parameters to \text{fn}, so this picture does not apply to Pyret’s \text{fold}. 
Implementing my-sum using my-fold

\[
\text{link}(1, \\
\text{link}(2, \\
\text{link}(3, \text{empty}))) \quad \rightarrow \quad +(1, \\
+(2, \\
+(3, 0)))
\]
Implementing my-sum using my-fold

```
fun my-sum-fold(lst :: List<Number>) -> Number:
    my-fold(lam(x,y): x + y end, 0, lst)
where:
    my-sum-fold(empty) is 0
    my-sum-fold([list: 1, 2, 3]) is 6
end
```

Unfortunately, we cannot use + as a parameter to my-fold, so we must write a lambda expression equivalent to the + operator.
Implementing sum of squares using my -fold

```plaintext
fun ssq3(lst :: List<Number>) -> Number:
  my-fold(lam(e,r): (e * e) + r end, 0, lst)
where:
  ssq3(empty) is 0
  ssq3([list: 1,2,3]) is 14
end
```
Implementation of my-fold

fun my-fold(fn :: Function, b :: Number, lst :: List<Number>) -> Number:
  cases (List) lst:
  | empty => b
  | link(fst, rst) => fn(fst, my-fold(fn, b, rst))
end

where:
  my-fold(lam(e,r): (e * e) + r end, 0, empty) is 0
  my-fold(lam(e,r): (e * e) + r end, 0, [list: 1,2,3]) is 14
end