1 Reading

- Little Schemer ch. 2-4
- EOPL 1.1 - 1.2

2 Why Do We Study Flat Recursion?

The objective of this lecture is to prepare you to study representations and operations on data that is inherently recursive in nature. Examples of such data include languages (grammars), and other recursive data (like windows, drawings, etc) We will study the basic ideas using lists at first, but these ideas also apply to other recursive data.

Why? recursion used extensively in grammars for programming languages matching recursion produces a "well-structured" program, that is easy to write and understand.

3 Inductive Definition of Lists

Let us start with inductively defining lists.

**Definition 3.1** A *(list-of T)* is

- *either* ()
- *or a pair, (cons v l), where v:T and l:(list-of T)*

An alternative way of writing is by specifying the grammar for forming lists as shown below.

<list-of-T>::= () | {<T> . <list-of-T>}

3.1 Tips (summary)

Warning: be systematic! I’m trying to give you another way to think; the usual imperative way (try something and debug) wastes lots of time (hours for 4 line program). There is no way you’ll get the right answer quickly by tinkering and debugging.

3.2 Tips for Reducing HW Time for Recursive Programs

1. Write out your own examples if needed
   - base case(s)
   - related simpler example
2. Specify the input and output data types.
3. Use an outline that matches the grammar for the input data type.
4. Fill in the outline by generalizing the examples
   - What’s returned in the base case(s)?
   - How do we get the answer from
Example 3.2 Write a procedure to sum the squares of numbers in a list.

We will follow the tips above for writing this recursive procedure. First, we need to write the base case.

$sqrt = 0$

then we need to write related simpler examples.

$sqrt (3 4 2) = \Rightarrow 29$
$sqrt (4 2) = \Rightarrow 20$
$sqrt (2) = \Rightarrow 4$

We know that the input data type is a list with the following grammar.

\[
<list-of-T>::= () | (<T> . <list-of-T>)
\]

Now let us fill the outline of the recursive procedure by following the grammar.

\[
(define \text{sum-squares} \\
  (lambda \text{(lon)} \\
    (cond \\
        ((null? lon) ______) \\
        (else (_________ \\
                       (sum-squares (cdr lon))))))))
\]

Now we have to fill in the outline by generalizing the examples. The first question is what is returned in the base case (s).

$sqrt (()) = \Rightarrow 0$

From this we can fill in part of the recursive procedure that has to do with the base case.

\[
(define \text{sum-squares} \\
  (lambda \text{(lon)} \\
    (cond \\
        ((null? lon) 0) \\
        (else (_________ \\
                       (sum-squares (cdr lon))))))))
\]

How do we get the answer from the recursive call’s answer and other information. The key here is to use the related simpler example to ask how we can generalize the function.

$sqrt (3 4 2) = \Rightarrow 29$

\[
\text{lst} (\text{sum-squares lst})
\]

$sqrt (4 2) = \Rightarrow 20$

\[
(cdr \text{lst}) (\text{sum-squares (cdr lst)})
\]

Based on all of this information, it is clear that the final form ought to be the following.

\[
(define \text{sum-squares} \\
  (lambda \text{(lon)} \\
    (cond \\
        ((null? lon) 0) \\
        (else (+ (square (car lon)) \\
                       (sum-squares (cdr lon))))))))
\]

3.3 Debugging Tips

The rule of thumb for debugging is to work bottom up from subexpressions.
If you need to use a recursive call when debugging, use the value instead.
(+ (car lst) (car lst) 20)

Don’t try things randomly when debugging, use a strategy!

3.4 Derivation from Related Examples

Example 3.3 (zip) Write a procedure, zip, that takes two lists and produces a list of two element lists. For example,

\[
\begin{align*}
\text{(zip '}(3 \ 4 \ 2) \ '}(5 \ 9 \ 7) & \\
\Rightarrow & \ ((3 \ 5) \ (4 \ 9) \ (2 \ 7)) \\
\text{(zip '}(4 \ 2) \ '}(9 \ 7) & \\
\Rightarrow & \ ((4 \ 9) \ (2 \ 7)) \\
\text{(zip '}(\) \ '}(1 \ 4 \ 1 \ 5 \ 9) & \\
\Rightarrow & \ () \\
\text{(zip '}(2 \ 7 \ 3 \ 4) \ '}(\) & \\
\Rightarrow & \ ()
\end{align*}
\]

Let us follow the tips again.

- What’s a related simpler example for the second example?
- What are the cases? What’s the outline?
- What is the answer for the base cases?
- How to decompose? Where is the recursion?
- How do we get from the answer for the recursion to what we want? Look at the related example, generalize.

Following the tips, here is the final form of the zip function.

\[
\begin{align*}
\text{(define zip} \\
\text{(lambda (fsts snds))} \\
\text{(cond} \\
\text{((null? fsts) '()} \\
\text{((null? snds) '()} \\
\text{(else (cons (list (car fsts) (car snds))} \\
\text{(zip (cdr fsts) (cdr snds)))))))))}
\end{align*}
\]

So how does the zip function works?

\[
\begin{align*}
\text{(zip '}(3 \ 4 \ 2) \ '}(5 \ 9 \ 7) & \\
= & \ ((\text{lambda} (\text{fsts snds} \text{ (cond ...)))} \\
\text{'}(3 \ 4 \ 2) \ '}(5 \ 9 \ 7) \\
= & \ (\text{cond} ((\text{null? '}(3 \ 4 \ 2)) '()) \\
\text{((null? '}(5 \ 9 \ 7)) '()) \\
\text{(else (cons (list (car '}(3 \ 4 \ 2)) (car '}(5 \ 9 \ 7)))} \\
\text{(zip (cdr '}(3 \ 4 \ 2)) (cdr '}(5 \ 9 \ 7)))))) \\
= & \ (\text{cons (list (car '}(3 \ 4 \ 2)) (car '}(5 \ 9 \ 7))) \\
\text{zip (cdr '}(3 \ 4 \ 2)) (cdr '}(5 \ 9 \ 7)))) \\
= & \ (\text{cons (list 3 5} \\
\text{(zip '}(4 \ 2) \ '}(9 \ 7))) \\
= & \ (\text{cons (list 3 5} \\
\text{cons (list 4 9} \\
\text{zip '}(2 \ '}(7))))
\end{align*}
\]
Exercise 3.1  [xerox] Write a procedure, xerox, that takes a list of numbers and returns a list with those numbers duplicated. For example,

(xerox '(3 9 2))  =>  (3 3 9 9 2 2)
(xerox  '  (9 2))  =>  (9 9 2 2)
(xerox  '  ())  =>  ()

As discussed previously, following are the questions that you need to consider.

- What's a related example? (xerox  '(9 2))  =>  (9 9 2 2)
- What are the input and output types?
- How do you make (3 9 9 2 2) from (3 9 2) and (9 9 2 2)?
- How do you make (3 3 9 9 2 2) from (3 9 2) and (3 9 9 2 2)?

Example 3.4 [insert-before] Write a procedure, insert-before, that takes two elements, sought and to-insert, and a list of elements and which returns a list that is like the argument except that to-insert is placed in the output list just before the first occurrence of sought. For example,

(insert-before 'x 'q '())  =>  ()
(insert-before 'x 'q '(a x y z))
   =>  (a x y z)
(insert-before 'x 'q '(a x y z))
   =>  (a q x y z)

Grammar for input data:
<list-of-symbol> ::= ()
                    | (<symbol> . <list-of-symbol>)

Outline that matches grammar:
(define insert-before
  (lambda (where what los)
    • What examples are we missing?
    • Which argument has an inductively specified type?
    • so the outline, which gives partial credit, is 2 cases, with a test for the first
• also in the 2nd case, will recurse on cdr if need be because that’s where the grammar is recursive

• in the second case there are also 2 cases but those are not the context-free cases.

• finish writing this, by working with the examples...

• **Summary:** it’s the input data that determines the overall structure.

**Exercise 3.2**  [Insert-before-all] In pairs follow the process just described to write a recursive procedure insert-before-all. [insert-before-all] Write a procedure, insert-before, that takes two elements, sought and to-insert, and a list of elements and which returns a list that is like the argument except that to-insert is placed in the output list just before the first occurrence of sought. For example,

\[
\begin{align*}
(\text{insert-before-all} \ 'x' \ 'q' \ ()) & \Rightarrow () \\
(\text{insert-before-all} \ 'x' \ 'q' \ '(x \ y \ z \ x)') & \Rightarrow (q \ x \ y \ z \ q \ x) \\
(\text{insert-before-all} \ 'x' \ 'q' \ '(a \ x \ y \ z \ x)) & \Rightarrow '(a \ q \ x \ y \ z \ q \ x)
\end{align*}
\]

**Exercise 3.3** How does this differ from insert-before?