More lists

Readings: HtDP, sections 11, 12, 13 (Intermezzo 2).

Topics:

• Sorting a list [2–9]
• List abbreviations [10–14]
• Lists containing lists [15–29]
• Dictionaries and association lists [30–38]
• Lists of lists as 2D data [39–41]
• Processing two lists simultaneously [42–60]
• Consuming a list and a number [61–67]
• List equality [68-71]
Sorting a list

When writing a function to consume a list, we may find that we need to create an helper function to do some of the work.

*Key Point:* The helper function may or may not be recursive itself.

Sorting a list of numbers provides a good example; in this case the solution follows easily from the templates and design process.

In this course and CS 136, we will see several different sorting algorithms.
The list template

;; (sort lon) sorts the elements of lon in nondecreasing order
;; sort: (listof Num) → (listof Num)
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) . . . )

(define (sort lon)
  (cond [(empty? lon) . . . ]
    [else . . . (first lon) . . . (sort (rest lon)) . . . ])))

If the list lon is empty, so is the result.

Otherwise, the template suggests doing something with the first element of the list, and the sorted version of the rest.
Filling in the list template

;;; (sort lon) sorts the elements of lon in nondecreasing order
;;; sort: (listof Num) → (listof Num)
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) . . . )

(define (sort lon)
    (cond [[(empty? lon) empty]
            [else (insert (first lon) (sort (rest lon)))]])

Approach: insert (which we will implement) will be a recursive helper function that consumes a number and a sorted list. It inserts the number into the sorted list.
A condensed trace of **sort** and **insert**

\[
(sort \ (cons \ 2 \ (cons \ 4 \ (cons \ 3 \ empty))))
\]

\[\Rightarrow \ (insert \ 2 \ (sort \ (cons \ 4 \ (cons \ 3 \ empty))))\]

\[\Rightarrow \ (insert \ 2 \ (insert \ 4 \ (sort \ (cons \ 3 \ empty))))\]

\[\Rightarrow \ (insert \ 2 \ (insert \ 4 \ (insert \ 3 \ (sort \ empty))))\]

\[\Rightarrow \ (insert \ 2 \ (insert \ 4 \ (insert \ 3 \ empty))) \ ; \ insert \ 3 \ into \ empty \ list\]

\[\Rightarrow \ (insert \ 2 \ (insert \ 4 \ (cons \ 3 \ empty))) \ ; \ insert \ 4\]

\[\Rightarrow \ (insert \ 2 \ (cons \ 3 \ (cons \ 4 \ empty))) \ ; \ insert \ 2\]

\[\Rightarrow \ (cons \ 2 \ (cons \ 3 \ (cons \ 4 \ empty))) \ ; \ sorted \ list\]
Template for helper function **insert**

We again use the list template for **insert**.

;; (insert n slon) inserts the number n into the sorted list slon
;; so that the resulting list is also sorted.
;; insert: Num (listof Num) → (listof Num)
;; requires: slon is sorted in nondecreasing order
(define (insert n slon)
  (cond [(empty? slon) . . . ]
        [else (. . . (first slon) . . .
               (insert n (rest slon)) . . . )])))

CS 135 Fall 2019 08: More lists 6
Filling out the template

Recall that filling out the template is a matter of providing answers to certain questions.

If slon is empty, the result is the list containing just n.

If slon is not empty, another conditional expression is needed. n is the first number in the result if it is less than or equal to the first number in slon.

Otherwise, the first number in the result is the first number in slon, and the rest of the result is what we get when we insert n into (rest slon).
The implementation of **insert**

```
(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]))
```
A condensed trace of insert

\[(\text{insert } 4 \ (\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 5 \ \text{empty}))))\]

\[\Rightarrow (\text{cons } 1 \ (\text{insert } 4 \ (\text{cons } 2 \ (\text{cons } 5 \ \text{empty}))))\]

\[\Rightarrow (\text{cons } 1 \ (\text{cons } 2 \ (\text{insert } 4 \ (\text{cons } 5 \ \text{empty}))))\]

\[\Rightarrow (\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 4 \ (\text{cons } 5 \ \text{empty})))))\]

Our sort with helper function insert are together known as insertion sort.
List abbreviations

*Key Idea:* Now that we understand lists, we can abbreviate them.

In DrRacket, “Beginning Student With List Abbreviations” provides new syntax for list abbreviations, and a number of additional convenience functions.

Remember to follow the instructions in Module 01 when changing language levels.
List abbreviation 1: list

The expression

\((\text{cons } \text{exp}_1 \ (\text{cons } \text{exp}_2 \ (\ldots \ (\text{cons } \text{exp}_n \ \text{empty})\ldots ))))\)

can be abbreviated as

\((\text{list } \text{exp}_1 \ \text{exp}_2 \ldots \ \text{exp}_n)\)

The result of the trace we did on the last slide can be expressed as

\((\text{list } 1 \ 2 \ 4 \ 5)\).
Accessing elements of a list

(second my-list) is an abbreviation for (first (rest my-list)).

third, fourth, and so on up to eighth are also defined.

*Key Point:* Use these sparingly to improve readability.

The templates we have developed remain very useful.
**cons vs. list**

Note that `cons` and `list` have different results and different purposes.

We use `list` to *construct a list of fixed size* (whose length is known when we write the program).

We use `cons` to *construct a longer list* from one new element (the first) and a list of arbitrary size (whose length is known only when the second argument to `cons` is evaluated during the running of the program).
List abbreviation 2: Quoting lists

If lists built using `list` consist of *just symbols, strings, and numbers*, the list can be further abbreviated using the quote notation we used for symbols.

`(cons 'red (cons 'blue (cons 'green empty)))` can be written `'(red blue green)`.

`(list 5 4 3 2)` can be written `'(5 4 3 2)`, because quoted numbers evaluate to numbers; that is, `'1` is the same as 1.

What is `'()`?
Lists containing lists

Key Observation: Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

Here are two different two-element lists.

\[
\begin{array}{c}
1 \\
3
\end{array} \rightarrow
\begin{array}{c}
2 \\
4
\end{array}
\]

(cons 1 (cons 2 empty))

(cons 3 (cons 4 empty))
Example: a list containing another list

Here is a one-element list whose single element is one of the two-element lists we saw previously.

\[(\text{cons } (\text{cons } 3 (\text{cons } 4 \text{ empty})) \text{ empty})\]

We can create a two-element list by \text{cons}ing the other list onto this one-element list.
Example: a list containing two other lists

We can create a two-element list, each of whose elements is itself a two-element list.

(cons (cons 1 (cons 2 empty))
 (cons (cons 3 (cons 4 empty)) empty))
Expressing Nested Lists

We have several ways of expressing this list in Racket:

\[
\begin{align*}
& (\text{cons} (\text{cons} 1 (\text{cons} 2 \text{ empty})) \\
& \quad (\text{cons} (\text{cons} 3 (\text{cons} 4 \text{ empty})) \\
& \quad \text{empty})) \\
& \text{(list} \text{(list} 1 2) \text{(list} 3 4)\text{)} \\
& '((1 2) (3 4))
\end{align*}
\]

Clearly, the abbreviations are more expressive.
Example: taxes

A company needs to process their payroll – a list of employee names and their salaries. It produces a list of each employee name and the tax owed. The tax owed is computed with tax-payable from Module 04.

Payroll:

(list (list "Asha" 50000)
     (list "Joseph" 100000)
     (list "Sami" 10000))

TaxOwed:

(list (list "Asha" 7724.62)
     (list "Joseph" 18423.915)
     (list "Sami" 1500))
Data definitions for taxes

;;; A Payroll is one of:
;;; * empty
;;; * (cons (list Str Num) Payroll)

;;; A TaxOwed is one of:
;;; * empty
;;; * (cons (list Str Num) TaxOwed)

Note: Both Payroll and TaxOwed are (listof X) where X is a two-element list.
Template: preliminary version

;; (payroll-template pr)

;; payroll-template: Payroll → Any

(define (payroll-template pr)
  (cond [(empty? pr) . . .]
        [(cons? pr) . . . (first pr) . . .
         . . . (payroll-template (rest pr)) . . .]
        (else . . .)))

A payroll is just a list, so it looks exactly like the (listof X) template – so far...
Template: refinements

Key Point: Some information from our data definition is not yet captured in the template: The list’s first item is known to be of the form (list Str Num).

It’s useful to reflect that fact in the template:

- It reminds us of all the data available to us when solving the problem.

- Our solutions (derived from the template) will often access the parts of the sublist.
Template: final version

;; (payroll-template pr)

;; payroll-template: Payroll → Any

(define (payroll-template pr)
  (cond [(empty? pr) . . . ]

    [(cons? pr) (. . . (first (first pr)) . . .
                     . . . (first (rest (first pr))) . . .
                     . . . (payroll-template (rest pr)) . . . )]

    [(cons? pr) . . . (first (first pr)) . . .
                 . . . (first (rest (first pr))) . . .
                 . . . (payroll-template (rest pr)) . . . ]))

Some short helper functions (namely name and amount) will make our code more readable.
;; (name lst) produces the first item from lst – the name.
(define (name lst) (first lst))

;; (amount lst) produces the second item from lst – the amount.
(define (amount lst) (first (rest lst)))

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) . . .]
        [(cons? pr) (. . . (name (first pr)) . . .
                      . . . (amount (first pr)) . . .
                      . . . (payroll-template (rest pr)) . . .))])

Note: Non-recursive helper functions only need a purpose.
Using the design recipe: fill in template

;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; compute-taxes: Payroll → TaxOwed
(check-expect (compute-taxes test-payroll) test-taxes)
(define (compute-taxes payroll)
  (cond 
    [(empty? payroll) ... ]
    [(cons? payroll) (... (name (first payroll)) ... 
      ... (amount (first payroll)) ... 
      ... (compute-taxes (rest payroll)) ... )]))

Now fill in the details...
One Version of **compute-taxes**

;; (compute-taxes payroll) calculates the tax owed for each employee/salary pair in the payroll.

;; compute-taxes: Payroll → TaxOwed

(check-expect (compute-taxes test-payroll) test-taxes)

(define (compute-taxes payroll)
    (cond 
        [(empty? payroll) empty]
        [(cons? payroll) (cons (list (name (first payroll))
                                (tax-payable (amount (first payroll))))
                              (compute-taxes (rest payroll)))]))

We could create a helper function to translate Payroll into TaxOwed.
Another version of **compute-taxes**

;; (sr→tr salary-rec) consumes a salary record and produces the corresponding tax record

;; sr→tr: (list Str Num) → (list Str Num)

(define (sr→tr salary-rec)
  (list (name salary-rec) (tax-payable (amount salary-rec))))

(define (compute-taxes-alt payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll) (cons (sr→tr (first payroll))
                                  (compute-taxes-alt (rest payroll)))]))