List abbreviations
List abbreviations

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 abbreviations

The expression

\[(\text{cons} \ \text{exp1} \ (\text{cons} \ \text{exp2} \ (\ldots \ (\text{cons} \ \text{expn} \ \text{empty}) \ \ldots \ )))\]

can be abbreviated as

\[(\text{list} \ \text{exp1} \ \text{exp2} \ \ldots \ \text{expn})\]

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

\[(\text{list} \ 1 \ 2 \ 4 \ 5).\]
List abbreviations

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

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

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

\[(\text{cons } '\text{red } (\text{cons } '\text{blue } (\text{cons } '\text{green empty}))))\text{ can be written }'(\text{red blue green}).\]

\[(\text{list } 5 4 3 2)\text{ can be written }'(5 4 3 2), \text{ because quoted numbers evaluate to numbers; that is, }'1\text{ is the same as }1.\]

What is '()?
List containing lists
Lists containing lists

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

Here are two different two-element lists.

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

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

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

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

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

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

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

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

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

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

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

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

\[
(\text{cons} \quad 
(\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{empty})) \\
(\text{cons} \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ \text{empty})) \ \text{empty})
\]
Lists containing lists

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

\[
\text{cons}
\begin{align*}
\text{(cons} & \ 1 \ \text{(cons} \ 2 \ \text{empty})) \\
\text{(cons} & \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ \text{empty})) \ \text{empty})
\end{align*}
\]
Lists containing lists

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

\[
(\text{cons} \\
(\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{empty})) \\
(\text{cons} \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ \text{empty})) \ \text{empty}))
\]
List abbreviations

We have several ways of expressing this list in Racket:

(cons
  (cons 1 (cons 2 empty))
  (cons
    (cons 3 (cons 4 empty))
    empty))

(list (list 1 2) (list 3 4))

'( (1 2) (3 4) )

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" 7610)
  (list "Joseph" 18135)
  (list "Sami" 1500))
Data definitions

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

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

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

A payroll is just a list, so this looks exactly like the (listof X) template.
Data definition template

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 is 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.
Data definition template

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

Some short helper functions will make our code more readable.
Data definition template

;;; (name lst) produces the first item from lst (name).
(define (name lst) (first lst))

;;; (amount lst) produces the second item from lst (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)))]))

Non-recursive helper functions only need a purpose.
Start design recipe; fill in template

;; (compute-taxes payroll) calculates the tax owed by each employee / salary pair in 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))))])
Finish compute-taxes

;; (compute-taxes payroll) calculates the tax owed by each employee / salary pair in 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))))])
Alternate templates leading to a different solution

;; (payroll-template pr)
;; payroll-template: Payroll -> Any
(define (payroll-template pr)
  (cond
   [(empty? pr) ...]
   [(cons? pr) (... (salary-rec-template (first pr))
                    (payroll-template (rest pr)))]]))

(define (salary-rec-template sr)
  (... ...(name sr)
        ...(amount sr)))
Different solution

\[
\text{(define (compute-taxes-alt payroll)} \rightarrow (compute-taxes-alt (rest payroll)))
\]

;; (sr->tr salary-rec) consumes a salary record and
;; produces the corresponding tax record
;; sr->tr: (list Str Num) -> (list Str Num)

\[
\text{(define (sr->tr salary-rec)} \rightarrow (list (tax-payable (amount salary-rec)))
\]
Different kinds of lists

When we introduced lists in module 05, the items they contained were not lists. These were flat lists.

We have just seen lists of lists in our example of a list containing a two-element flat list.

In later lecture modules, we will use lists containing unbounded flat lists.

We will also see nested lists, in which lists may contain lists that contain lists, and so on to an arbitrary depth.
Dictionaries and association lists
Dictionaries

Once upon a time, a dictionary was a book in which you look up a word to find a definition.
Dictionaries

More generally, a dictionary contains a number of keys, each with an associated value.

Examples:

• Your contacts list. Keys are names, and values are telephone numbers.

• Your seat assignment for midterms. Keys are userids, and values are seat locations.

• Stock symbols (keys) and prices (values).

• Many two-column tables can be viewed as dictionaries. The previous examples can all be viewed as two-column tables.
Dictionary operations

What operations might we wish to perform on dictionaries?

• lookup: given key, produce corresponding value
• add: add a (key,value)-pair to the dictionary
• remove: given key, remove it and associated value
Association lists

One simple solution uses an association list, which is just a list of (key, value) pairs.

We store the pair as a two-element list. For simplicity, we will use numbers as keys and strings as values.

;; An association list (AL) is one of:
;;  * empty
;;  * (cons (list Num Str) AL)
Association lists

We can create association lists based on other types for keys and values. We use Num and Str here just to provide a concrete example.

We impose the additional restriction that an association list contains at most one occurrence of any key.

Since we have a data definition, we could use AL for the type of an association list, as given in a contract.

Another alternative is to use (listof (list Num Str)).
Association lists template

We can use the data definition to produce a template:

```scheme
;; al-template: AL -> Any
(define (al-template alst)
  (cond
   [(empty? alst) ...]
   [else (...)
    (first (first alst)) ... ; first key
    (second (first alst)) ... ; first value
    (al-template (rest alst))]]))
```
lookup function

Recall that lookup consumes a key and a dictionary (association list) and produces the corresponding value.

(check-expect
  (lookup 3 (list (list 1 "John") (list 3 "Winnie")))
  "Winnie")
lookup function

In coding lookup, we have to make a decision. What should it produce if the lookup fails?

Since all valid values are strings, it can produce false to indicate that the key was not present in the association list.

(check-expect
  (lookup 2 (list (list 1 "John") (list 3 "Winnie")))
  false)
lookup function

;; (lookup-al k alst) produces the value corresponding to key k, or false if k not present
;; lookup-al:

(define (lookup-al k alst)
  (cond
   [(empty? alst) false]
   [(= k (first (first alst))) (second (first alst))]
   [else (lookup-al k (rest alst))])))
add and remove function

We will leave the add and remove functions as exercises.

This solution is simple enough that it is often used for small dictionaries.

For a large dictionary, association lists are inefficient in the case where the key is not present and the whole list must be searched.

In a future module, we will impose structure to improve this situation.