Tracing count-concerts

(count-concerts (cons "a" (cons "b" empty)))
⇒ (cond [(empty? (cons "a" (cons "b" empty))) 0]
   [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))])
⇒ (cond [false 0]
   [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))])
⇒ (cond [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))])
⇒ (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))
⇒ (\(+\ 1\ (\text{count-concerts (cons } "b" \text{ empty})))\)
⇒ (\(+\ 1\ (\text{cond } [(\text{empty? (cons } "b" \text{ empty)) } 0][\text{else (} (+\ 1 \ldots\ )))])\)
⇒ (\(+\ 1\ (\text{cond } [\text{false } 0][\text{else (} (+\ 1 \ldots\ )))])\)
⇒ (\(+\ 1\ (\text{cond } [\text{else (} (+\ 1 \ldots\ )))])\)
⇒ (\(+\ 1\ (+\ 1\ (\text{count-concerts (rest (cons } "b" \text{ empty}))))))\)
⇒ (\(+\ 1\ (+\ 1\ (\text{count-concerts empty})))\)
⇒ (\(+\ 1\ (+\ 1\ (\text{cond } [(\text{empty? empty) } 0][\text{else (} (+\ 1 \ldots\ )))])\)
⇒ (\(+\ 1\ (+\ 1\ (\text{cond } [\text{true } 0][\text{else (} (+\ 1 \ldots\ )))])\)
⇒ (\(+\ 1\ (+\ 1\ 0))⇒ (\(+\ 1\ 1)⇒ 2\)

Here we have used an omission ellipsis to avoid overflowing the slide.
Condensed traces

The full trace contains too much detail, so we instead use a condensed trace of the recursive function. This shows the important steps and skips over the trivial details.

This is a space saving tool we use in these slides, not a rule that you have to understand.
The condensed trace of our example

(count-concerts (cons "a" (cons "b" empty)))
⇒ (+ 1 (count-concerts (cons "b" empty)))
⇒ (+ 1 (+ 1 (count-concerts empty)))
⇒ (+ 1 (+ 1 0))
⇒ 2

This condensed trace shows more clearly how the application of a recursive function leads to an application of the same function to a smaller list, until the base case is reached.
Use of a condensed trace

From now on, for the sake of readability, *we will tend to use condensed traces*. At times we will condense even more (for example, not fully expanding constants).

If you wish to see a full trace, you can use the Stepper.

But as we start working on larger and more complex forms of data, it becomes harder to use the Stepper, because intermediate expressions are so large.
Example: count-waterboys

;; (count-waterboys los) produces the number of occurrences
;; of "Waterboys" in los
;; count-waterboys: (listof Str) → Nat
;; Examples:
(check-expect (count-waterboys empty) 0)
(check-expect (count-waterboys (cons "Waterboys" empty)) 1)
(check-expect (count-waterboys (cons "DaCapo"
   (cons "U2" empty))) 0)

(define (count-waterboys los) . . . )
Generalizing: count-waterboys

The template is a good place to start writing code. Write the template. Then, alter it according to the specific function you want to write.

For instance, we can generalize count-waterboys to a function which also consumes the string to be counted.

;;; count-string: Str (listof Str) → Nat
(define (count-string s los) . . . )

The recursive function application will be (count-string s (rest los)).
More about list templates

Here are three crucial questions to help think about functions consuming a list and filling in their templates:

• What does the function produce in the base case? Fill in that part of the template.

• What does the function do to the first element in a non-empty list? Fill in that part of the template.

• How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list?
Refining the \textbf{(listof X) template}

Sometimes, each \textit{X} in a \textbf{(listof X)} may require further processing. Indicate this with a template for \textit{X} as a helper function.

\[
\text{;; listof-X-template: (listof X) } \rightarrow \text{ Any}
\]

\[
\text{(define (listof-X-template lox)}
\]

\[
\text{(cond [(empty? lox) \ldots ]}
\]

\[
\text{[else (\ldots (X-template (first lox)) \ldots}
\]

\[
\text{\ldots (listof-X-template (rest lox)) \ldots ]]))}
\]

\textit{We assume this generic data definition and template from now on.}
Templates as generalizations

*Key Point:* A template provides the basic shape of the code as suggested by the data definition.

Later in the course, we will learn about an abstraction mechanism (higher-order functions) that can reduce the need for templates.

We will also discuss alternatives for tasks where the basic shape provided by the template is not right for a particular computation.
Patterns of recursion

The list template has the property that *the form of the code matches the form of the data definition.*

We will call this **simple recursion**.

There are other patterns of recursion which we will see later on in the course.

Until we do, the functions we write (and ask you to write) will use simple recursion (and hence will fit the form described by such templates).

Use the templates.
Simple recursion

A clearer definition: In simple recursion, every argument in a recursive function application is either:

• unchanged, or
• one step closer to a base case according to a data definition

(define (func lst) ... (func (rest lst)) ...) ;; Simple
(define (func lst x) ... (func (rest lst) x) ...) ;; Simple
(define (func lst x) ... (func (process lst) x) ...) ;; NOT Simple
(define (func lst x)
  ... (func (rest lst) (math-function x)) ...) ;; NOT Simple
Useful list functions

A closer look at `count-concerts` reveals that it will work just fine on any list.

In fact, it is a built-in function in Racket, under the name `length`.

Another useful built-in function is `member?`, which consumes an element of any type and a list, and returns `true` if the element is in the list, or `false` if it is not present.
Producing lists from lists

Consider **negate-list**, which consumes a list of numbers and produces the same list with each number negated (3 becomes $-3$).

;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)

(check-expect (negate-list empty) empty)

(check-expect (negate-list (cons 2 (cons $-12$ empty)))
  (cons $-2$ (cons 12 empty)))

Since **negate-list** consumes a (listof Num), we use the general list template to write it.
Example: negate-list with template

;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons −12 empty)))
  (cons −2 (cons 12 empty)))

(define (negate-list lon)
  (cond [(empty? lon) . . . ]
      [else (. . . (first lon) . . . (negate-list (rest lon)) . . . )]]))
Example: negate-list completed

;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)

;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons −12 empty)))
  (cons −2 (cons 12 empty)))

(define (negate-list lon)
  (cond [(empty? lon) empty]
    [else (cons (− (first lon)) (negate-list (rest lon))))])
A condensed trace

\[(\text{negate-list } (\text{cons } 2 (\text{cons } -12 \text{ empty}))))\]
\[\Rightarrow (\text{cons } (\text{cons } -2) (\text{negate-list } (\text{cons } -12 \text{ empty}))))\]
\[\Rightarrow (\text{cons } -2 (\text{negate-list } (\text{cons } -12 \text{ empty}))))\]
\[\Rightarrow (\text{cons } -2 (\text{cons } (\text{cons } -\text{cons } -12) (\text{negate-list } \text{empty}))))\]
\[\Rightarrow (\text{cons } -2 (\text{cons } 12 (\text{negate-list } \text{empty}))))\]
\[\Rightarrow (\text{cons } -2 (\text{cons } 12 \text{ empty})))\]
Nonempty lists

*Key Point:* Sometimes a given computation makes sense only on a nonempty list.

For example, finding the maximum of a list of numbers.

**Exercises:**

Create a self-referential data definition for `(ne-listof X)`, a nonempty list of X.

Develop a template for a function that consumes an `(ne-listof X)`.

Finally, write a function to find the maximum of a nonempty list of numbers.
Design recipe refinements

When we introduce new types, like \texttt{(ne-listof X)}, we need to include it in the design recipe.

\textit{Key Point:} For each new type, place the following someplace between the top of the program and the first place the new type is used:

- The data definition
- The template derived from that data definition

This information is only needed \textbf{once}.
Data definitions

Example:

;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))

In a self-referential data definition, at least one clause (possibly more) must not refer back to the definition itself; these are base cases.

Assignments do not need to include data definitions or templates for (listof X). You do for (ne-listof X) and other types you may define.
Templates

The template follows directly from the data definition.

Key Point: The overall shape of a self-referential template will be a cond expression with one clause for each clause in the data definition.

Self-referential data definition clauses lead to recursive expressions in the template.

Base case clauses will not lead to recursion.

The per-function part of the design recipe stays as before.
Strings and lists of characters

Processing text is an extremely common task for computer programs. Text is usually represented in a computer by strings.

In Racket (and in many other languages), a string is really a sequence of characters.

Racket provides the function string→list to convert a string to an explicit list of characters.

The function list→string does the reverse: it converts a list of characters into a string.
Representing strings

*Key Point:* Racket’s notation for the character ‘a’ is `#\a`.

The result of evaluating `(string→list "test")` is the list `(cons #\t (cons #\e (cons #\s (cons #\t empty))))`.

This is unfortunately ugly, but the `#` notation is part of a more general way of specifying values in Racket.
Example: Counting characters in a string

Write a function to count the number of occurrences of a specified character in a string. Start by counting the occurrences in a list of characters.

;; (count-char/list ch loc) counts the number of occurrences of ch in loc.
;; count-char/list: Char (listof Char) → Nat
(check-expect (count-char/list #\e (string→list " ")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)
(check-expect (count-char/list #\o (cons #\f (cons #\o (cons #\o (cons #\d empty)))))) 2)
;; (count-char/list ch loc) counts the number of occurrences of ch in loc.
;; count-char/list: Char (listof Char) → Nat
(check-expect (count-char/list #\e (string→list " ")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)

(define (count-char/list ch loc)
  (cond [(empty? loc) 0]
        [else (+ (cond [(char=? ? ch (first loc)) 1]
                   [else 0])
               (count-char/list ch (rest loc)))]))
Wrapper functions

Our functions should be easy to use. The problem statement was to count characters in a string, not in a list of characters.

We shouldn’t expect the user of our function to know that to use \texttt{count-char/list} they need to convert their string to a list of characters.

In such cases it’s good practise to include a \texttt{wrapper function} – a simple function that “wraps” the main function and takes care of housekeeping details like converting the string to a list.
;; (count-char ch s) counts the number of occurrences
;; of ch in s.
;; count-char: Char Str → Nat
(check-expect (count-char #\e " " ) 0)
(check-expect (count-char #\e "beekeeper") 5)

(define (count-char ch s)
  (count-char/list ch (string→list s)))
Characteristics of wrapper functions

Wrapper functions:

• are short and simple

• always call another function that does much more

• *sets up the appropriate conditions for calling the other function*, usually by transforming one or more of its parameters or providing a starting value for one of its arguments
Goals of this module

You should understand the data definitions for lists, how the template mirrors the definition, and be able to use the template to write recursive functions consuming this type of data.

You should understand box-and-pointer visualization of lists.

You should understand the additions made to the semantic model of Beginning Student to handle lists, and be able to do step-by-step traces on list functions.
You should understand and use \((\text{listof } X)\) notation in contracts.

You should understand strings, their relationship to characters and how to convert a string into a list of characters (and vice-versa).
Module 06 Summary

Lists

1. Two functions to create lists: empty and cons (i.e. construct). [4]
2. Two functions to check lists: empty? and cons? (e.g. is it non-empty). [7]
3. Two functions to break-up a list into two parts: first and rest. [7]
4. Use (listof X) in contracts, where X is one of Num, Bool, Sym, Any etc. [15]
5. Use anyof to list the mixed types explicitly in a contract. [16]
6. (cons a b) is a value, i.e. it won’t be simplified. [17]
7. For (cons a b), a must be a value and b must be a list. [17]
Module 06 Summary

Recursion, Templates and Traces

8. A recursive definition consists of a base case and a recursive (i.e. self-referential) case. [20]

9. A template is a general framework within which we fill in specifics. [22]

10. A function is recursive when the body of the function involves an application of the same function. [30]

11. We will typically use a condensed trace where we just show the results of the application of each recursive function call (rather than substitute the body of the function). [33-35]
Module 06 Summary

Design Recipe and Templates

12. For recursive list functions,
   (a) start with cond answers for the base cases,
   (b) then consider the first element
   (c) and finally how to combine the answers from first and rest of
       the list. [38]


14. Create a self-referential data definition and template for each
    new self-referential type. [49]

15. Then follow the Design Recipe for each function. [51]
Module 06 Summary

Simple Recursion

16. For simple recursion the form of the code matches the form of the data definition. [41]

17. For simple recursion, every argument is either (a) unchanged, [42] (b) one step closer to a base case. [42]

18. Two list functions are member? and length. [43]

19. The base case for the data definition of a non-empty list of Nums would be (cons Num empty) rather than empty. [48]
Module 06 Summary

Strings and Wrapper Functions

20. A string can be thought of as a list of characters. [45]
21. Racket provides two functions `string->list` and `list->string` to convert back and forth. [52]
22. The Racket notation for the character ‘a’ is `\a`. [50]
23. A **wrapper function** is a small function that modifies the arguments passed to a **primary function**. [58]