L7

| | | |
|---|---|---|
| 5 | clickers | |
| 15 | ListOfString | present |
| 15 | contains-canucks | do/interactive |
| 10 | revise recipes | present |
| 5 | ListOfNumber | cam |
| 15 | sum | class |
| 10 | count/product | class |

note 288 views

# Section 101 To Do for September 26th

Hard work today on self-reference and recursion!

To wrap up today's work you should:

- Review the first 5 videos of the Self-Ref module. These cover the same ground as we covered in lecture today, but go into some important points in more detail.
- Work through the "Self-Ref - Designing with Lists" video carefully. This is a chance to really solidify your understanding of functions that operate on lists.
- Work through the "Self-Ref - Positions in List Templates" video.
- Do the practice problems in the Self-Ref module.

Before the next class you should watch the "Ref - The Reference Rule Part 1" video. Note that in this case we are saying watch, not work through. It will be sufficient in this case to just watch the video. But do pay attention - there may still be clickers on it, and more importantly you will need to have watched it in order to be able to work through the example we will do next time.

**Super important before next time:** In the next class we will design a world program in which the world state is a list of structures. Its a fun program where you can click the mouse on the screen to make it rain where you click. This is the most complex program we have done yet. Its super important that you be comfortable with world problems, lists and compound data all three before you come to class Tuesday!

The good news is that the weather forecast is GREAT for studying 110!

| Fri Sep 27 | Sat Sep 28 | Sun Sep 29 | Mon Sep 30 | Tue Oct 1 |
|---|---|---|---|---|
| Rain | Rain | Rain | Rain | Rain |
| 14°C | 13°C | 14°C | 13°C | 14°C |
| Feels like 14 | Feels like 13 | Feels like 13 | Feels like 12 | Feels like 14 |
| Low 10°C | Low 11°C | Low 11°C | Low 11°C | Low 11°C |

101 todo lecture

5 days ago by Gregor Kiczales

**followup discussions** *for lingering questions and comments*

```
;;234567890123456789012345678901234567890123456789012345678901234
;;|<--             64 columns wide for projection            -->|

;; Self-Ref - List Data Definition and Function on a List

;;
;; Let's try to design with lists a little. We want to be able to
;; ask whether
;; "Canucks" appears in lists like:
;;
;;   (cons "Flames" (cons "Canucks" (cons "Leafs" empty)))
;;


;; Here's a try at a type comment:

;; 3String is (cons String (cons String (cons String empty)))
;; interp. a list of 3 strings
(define TS1 (cons "Flames" (cons "Canucks" (cons "Leafs" empty))))
```

```
;
But what about lists of ARBITRARY length. That's what we set
out to do. The above won't do that. It won't do to have one DD
and fn for lists 3 long, another for lists 4 long and so on.
Something about the above can't be right.

Here's very different kind of data definition to consider:
```

```
;; ListOfString is one of:
;;  - empty
;;  - (cons String ListOfString)
;; interp. a list of strings
```

self reference

have printed version to draw on

```
;
Something there may strike you as funny.  The ListOfString data
definition refers to a non-primitive type... but that type is
itself! That is called being self-referential or recursive, and
its incredibly important and interesting. Only some cases of
self-reference make any real sense, and we'll talk about what
characterizes those cases shortly.

Its worth drawing an arrow on the type comment from the REFERENCE
to ListOfString back to the DEFINITION of ListOfString. We'll
call that a self-reference arrow and label it with SR. to
mark it as being different from the reference arrow we saw
last week.

What about the lists we have been working with? Does this types
comment match them?

empty is ListOfString

(cons "Canucks" empty) is ListOfString

(cons "Flames" (cons "Canucks" empty)) is ListOfString

And so on, for String of arbitrary length. Each time there is
one more item on the list we just follow the self-reference
back to ListOfString one more time.

The self-reference gives us the arbitrary size we want.



Now let's do the data definition examples, that's not hard.
```

```
(define LOS1 empty)
(define LOS2 (cons "Canucks" empty))
(define LOS3 (cons "Flames" LOS2))
```

check against Type comment

draw SR arrow now

```
;
For the template let's follow the existing template rules FOR NOW
```

```
#;
(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else
         (... (first los)
              (fn-for-los (rest los)))]))
;; Template rules used:
;; - one of: 2 cases
;; - atomic distinct: empty
;; - compound: cons
;; - atomic non-distinct: (first los) is of type String
;; - ?????????: (rest los) is of type ListOfString
;;   ^^^^^^^^^
;;       ||
```

```
;; we don't really know what to do here, but we know (rest los) is
;; of type ListOfString. So let's just put a call to ~~some~~ fn-for-los
;; at this point in the template; ~~that's what the reference rule~~
;; ~~tells us to do for ordinary references.~~
```

we know we have to call some fn based on this template

; OK, so now let's try to design a function. We want to design a function that produces true if a ListOfString contains "Canucks".

```
;; ListOfString -> Boolean
;; produces true if los contains "Canucks"
(check-expect (contains-canucks? empty) false)
(check-expect (contains-canucks? (cons "Canucks" empty)) true)
(check-expect (contains-canucks? (cons "Flames" (cons "Canucks" empty))) true)
#;
(define (contains-canucks? los)
  (cond [(empty? los) ...]
        [else
         (... (first los)
              (fn-for-los (rest los)))]))
```

; So now, working from the examples as usual:

first example tells us the answer for first cond clause is false

second and third examples tell us that sometimes second cond clause produces true and other times it produces false. so there is an if here

second clause tells us that if (first los) is "Canucks" should produce true

that gets us to:

```
#;
(define (contains-canucks? los)
  (cond [(empty? los) false]
        [else
         (if (string=? (first los) "Canucks")
             true
             (fn-for-los (rest los)))]))
```

~~[illegible]~~

add NR only + discuss after clicker

;Now we have to see whether rest contains "Canucks" and produce true if it does and false otherwise.

The template tells us we have to use a new or existing function for that.

HMMM... Do we have an existing function that will tell is whether the rest of the list containts "Canucks?"

Yes, the purpose of contains-canucks? right here says that it will. So we can just use that.
And in that same "stroke of the pen" we've also completed the function, so we should be done!

```
#;
(define (contains-canucks? los)
  (cond [(empty? los)  false]
        [else (if (string=? (first los) "Canucks")
                  true
                  (contains-canucks? (rest los)))]))
```

;That may or may not seem incredible to you.

We'l

~~But~~

(

to

(c

which

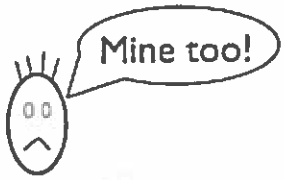
NOTE:

;

POLL 1:

Will this function work properly:

(A) YES
(B) NO
(C) My head hurts.




~~don't go there~~

picture

which the if
ll go ahead

```
(define (contains-canucks? los)
  (cond [(empty? los)  false]
        [else (or (string=? (first los) "Canucks")
                  (contains-canucks? (rest los)))]))
```

add SR arrow and talk about templates based on type theory

```
;;234567890123456789012345678901234567890123456789012345678901234
;;|<--             64 columns wide for projection            -->|

;; Self-Ref -  HtDF Recipe for Arbitrary-Sized Data
```

6

```
;
 Let's incorporate what we've seen into the design recipes.

 First, we have a new concept to use for designing data definitions.

   When the information to be represented is of ARBITRARY SIZE
   use a SELF-REFERENTIAL data definition.

   this will lead to types like ListOfString, ListOfNumber,
   ListOfFirework etc.

   A WELL-FORMED self referential data definition has:
    - at least one case withOUT self reference (called the base case)
    - at least one case with self reference

 There is a new template rule, for self-referential data definitions.
   The rule says that for a self reference we add a NATURAL RECURSION
   to the template. A natural recursion is a call back to the same
   function. (As opposed to some other function operating on that type).

   This works when the data definition is well-formed because for
   any given data, the base case will eventually be reached.

   The natural recursion in the function and the self-reference
   in the data definition correspond to each other.

 When designing a function for a well-formed self-referential data
 definition there are a few special points to be aware of.

   Always have a base case example first.
   Always have at least one example with a list >= 2 long.
   More examples may be needed depending on the function.

   Use the first example to tell you what the base case of the
   cond should do.

   Use the other examples to tell you what to do with the first
   item of the list and how to combine that with the result from
   the natural recursion.

   TRUST THE NATURAL RECURSION TO PRODUCE THE CORRECT RESULT!!!

   Avoid editing the natural recursion itself! Use it by assuming
   that the function will satisfy its signature and purpose. Just
   count on it to do that.

   The complete final version is:

 ;; ListOfString is one of:
 ;;  - empty
 ;;  - (cons String ListOfString)
 ;; interp. a list of strings
```

show on recipes

annotate DD

on recipes

show on code

```
(define LOS1 empty)
(define LOS2 (cons "Canucks" empty))
(define LOS3 (cons "Flames" LOS2))


#;
(define (fn-for-los los)
  (cond [(empty? los) ...]
        [else
         (... (first los)
              (fn-for-los (rest los)))]))
;; Template rules used:
;; - one of (2 cases)
;; - atomic distinct (empty)
;; - compound (cons)
;; (- atomic non-distinct (first los) is of type String)
;; - self-reference (rest los) is of type ListOfString



;; ListOfString -> Boolean
;; produces true if los contains "Canucks"
(check-expect (contains-canucks? empty) false)
(check-expect (contains-canucks? (cons "Canucks" empty)) true)
(check-expect (contains-canucks? (cons "Flames" (cons "Canucks" empty))) true)

(define (contains-canucks? los)
  (cond [(empty? los)  false]
        [else (or (string=? (first los) "Canucks")
                  (contains-canucks? (rest los)))]))
```

```
;; ===========================================================================
```

```
;
PROBLEM:

  Design a data definition to represent an arbitrary collection of numbers.
  You should call the type ListOfNumber.

  Design a function to count how many numbers are in a ListOfNumber.
```

```
;; ListOfNumber is one of:
;;  - empty
;;  - (cons Number ListOfNumber)
;; interp. a list of numbers
(define LON1 empty)
(define LON2 (cons 1 LON1))
(define LON3 (cons 3.4 LON2))
#;
(define (fn-for-lon lon)
  (cond [(empty? lon) (...)]
```

```
        [else
         (... (first lon)
              (fn-for-lon (rest lon)))]))
;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: cons
;;  - atomic non-distinct: first is Number
;;  - self-reference: rest is ListOfNumber


;; ListOfNumber -> Natural
;; count how many numbers in lon
(check-expect (count empty) 0)
(check-expect (count (cons 1 empty)) 1)
(check-expect (count (cons 1 (cons 1 empty))) 2)

#;
(define (count lon)
  (cond [(empty? lon) 0]
        [else
         (... (first lon)
              (count (rest lon)))])) ; we can DEPEND ON THE NATURAL
;;                                   ; RECURSION to produce the number
;;                                   ; of nums in (rest lon). We just
;;                                   ; need to add 1 to that!


(define (count lon)
  (cond [(empty? lon) 0]
        [else
         (+ 1    ;remember we don't always use the whole template
            (count (rest lon)))]))
```

# Please get out your clickers

And put bags UNDER your seat.

base

```
;; ListOfString is one of:
;;  - empty
;;  - (cons String ListOfString)
;; interp. a list of strings

(define LOS1 empty)
(define LOS2 (cons "Canucks" empty))
(define LOS3 (cons "Flames" LOS2))
```

SR

self reference

c2

```
#;
(define (fn-for-los los)
  (cond [(empty? los) ...]
        [else
         (... (first los)
              (fn-for-los (rest los)))]))
```

natural recursion

```
;; Template rules used:
;; - one of: 2 cases
;; - atomic distinct: empty
;; - compound: (cons String ListOfString)
;; - self-reference: (rest los) is ListOfString
```

base first

```
;; ListOfString -> Boolean
;; produces true if los contains "Canucks"
(check-expect (contains-canucks? empty) false)
(check-expect (contains-canucks? (cons "Canucks" empty)) true)
(check-expect (contains-canucks? (cons "Flames" (cons "Canucks" empty))) true)
(check-expect (contains-canucks? (cons "Flames" (cons "Sharks" empty))) false)

(define (contains-canucks? los)
  (cond [(empty? los)  false]
        [else (if (string=? (first los) "Canucks")
                  true
                  (contains-canucks? (rest los)))]))
```

trust the NR

C3

```
;; ListOfString is one of:
;;  - empty
;;  - (cons String ListOfString)
;; interp. a list of strings
(define LOS1 empty)
(define LOS2 (cons "Canucks" empty))
(define LOS3 (cons "Wings" (cons "Bruins" empty)))
(define LOS4 (cons "Sharks" LOS3))
#;
(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else
         (... (first los) ;String
              (fn-for-los (rest los)))]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: (cons String ListOfString)
;;  - self-reference: (rest los) is ListOfString


;; ListOfString -> Boolean
;; produce true if "Canucks" is in it
(check-expect (contains-canucks? empty) false)
(check-expect (contains-canucks? (cons "Canucks" empty)) true)
(check-expect (contains-canucks? (cons "Wings" (cons "Bruins" empty))) false)
(check-expect (contains-canucks? (cons "Wings" (cons "Canucks" empty))) true)


(define (contains-canucks? los)
  (cond [(empty? los) false]
        [else
         (if (string=? (first los) "Canucks")
             true
             (contains-canucks? (rest los)))]))
#;
;; template from ListOfString
(define (contains-canucks? los)
  (cond [(empty? los) false]
        [else
         (if (string=? (first los) "Canucks")
             true
             (contains-canucks? (rest los)))]))  ;TRUST THE NATURAL RECURSION
```

```
;; ListOfNumber is one of:
;;  - empty
;;  - (cons Number ListOfNumber)
;; interp. a list of numbers

(define LON1 empty)
(define LON2 (cons 1 empty))
(define LON3 (cons 3.2 LON2))


#;
(define (fn-for-lon lon)
  (cond [(empty? lon) ...]
        [else
         (... (first lon)
              (fn-for-lon (rest lon)))]))
```

;
PROBLEM:

Design a function that consumes a list of numbers and produces the sum of all the numbers in the list.

count ← product →

;
PROBLEM:

Design a function that consumes a list of numbers and produces the product of all the numbers in the list.

count