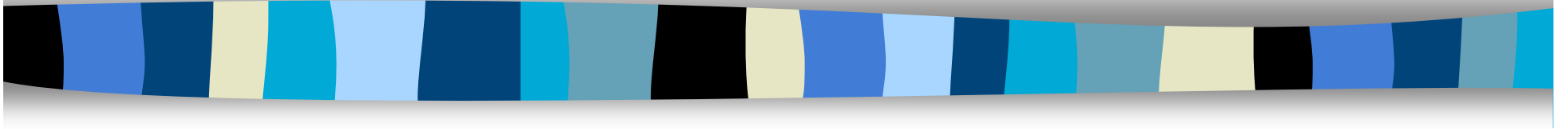


Processing two complex values





Announcements:

Lab this week

Quiz for next class (two)

Exams are being 'processed'
(hopefully ready by Fri.)



Question:

Consider mutually recursive definitions.

- * Each refers to the other...
- * But how can they be defined if they “essentially define themselves”?

What is an example?



Question: tree I

parent is :

(make-parent loc)

where loc is **list-of-children**

list-of children is:

empty

(cons p loc),

where p is a **parent**

loc is a **list-of-children**

Can we make this recursive?



Question: tree II

tree-node is :

empty

(make-node p loc)

where p is a (parent) **tree-node**

loc is a list of child **tree-nodes**

Here, mutual recursion is 'as hard as' recursion.
Why does recursion work?



Processing two complex values

What is a 'complex' value?



Familiar e.g.: add on Naturals

```
:: Nat-add: list list -> list
```

```
:: (Nat-add '(1 1) '(1 1)) yields '(1 1 1 1)
```

(Yes, I am cheating on representation a little. Let '1' be the Natural '1'.)

- How many recursive arguments?
 - Which argument do we recur on?
 - What about the 'other' argument?
 - Have we seen Nat-add before?
-



concat (argument 1)

:: **concat**: list list -> list

:: (concat '(1 2) '(3 4)) yields '(1 2 3 4)

```
(define (concat list-1 ...)
```

```
  (cond [(empty? list-1) ...]
```

```
        [else (... (first list-1)
```

```
                  (concat (rest list-1)
```

```
                  ...))]))
```



concat (argument 2)

:: concat: list list -> list

:: (concat '(1 2) '(3 4)) yields '(1 2 3 4)

```
(define (concat list-1 list-2)
```

```
  (cond [(empty? list-1) list-2]
```

```
        [else (... (first list-1)
```

```
                   (concat (rest list-1)
```

```
                           list-2))]))
```



Summary of concat

- Two recursive args
 - Recur on one arg.
 - Treat other arg 'atomically'

 - How does this template compare?
-



Processing two complex values

Let's try `list-pick`

- `list pick: list-of-sym N[>= 1] -> sym`
 - return *n*th sym from los, count from 1
 - signal error if no *n*th item
- `Examples`
 - how should we construct examples?



list-pick base cases

- (empty? los) (= n 1) ...
 - (empty? los) (> n 1) ...
 - (cons? los) (= n 1) ...
 - (cons? los) (> n 1) ...
-



Summary of `list-pick`

- Two recursive args
 - Each arg has different data def
 - Three 'special' base cases
 - Recur on **both** args (at the right time)
 - The standard template needs changes!
-



Let's look at `cross`:

- `cross`: list-of-sym list-of-num \rightarrow pairs of sym and num
 - think 'cross product'
- Examples
 - `(cross '(a b c) '(1 2))`
 - `(list '(a 1) '(a 2) '(b 1) '(b 2) '(c 1) '(c 2))`
 - `(a b) (1 2 3) = ((a 1) (a 2) (a 3) (b 1) ...)`
 - `empty (1 2) = empty`



CROSS:

- How is this like `list-pick`?
 - How is it different?
 - What are the base cases?
 - Which argument do we recur on?
-



cross: template

- (empty? los) ...
 - (else ...
-



Processing two complex values

Three (main) possibilities

- First is primary, second is atomic
 - append
 - First is primary, second is in lockstep
 - diff, zip, list=?, equal?, ...
 - Neither is primary, both must be processed together
 - (merge '(1 4 5) '(2 3)) yields '(1 2 3 4 5)
-



Processing two complex values

Impact on template

- First is primary, second is atomic
 - Nothing new
 - First is primary, second is in lockstep
 - Test on one (usually first), follow on other
 - Neither is primary, both must be processed together
 - Requires thinking! No general template!
-



For Next Class

- Homework already online
 - Reading:
 - Ch 18: Local definitions and scope
 - Quiz on reading
-