Module 5b: Helpers

CPSC 110

Peyton Seigo

2018-10-05

Module 5b: Helpers

Learning goals

Be able to design functions that use helper functions for each of the following reasons:

- at references to other non-primitive data definitions (this will be in the template)
- to form a function composition
- to handle a knowledge domain shift
- to operate on arbitrary sized data

Notes

Overview

New rules:

1. Rule of function composition
2. Rule of operating on arbtirary sized data
3. Rule of change in knowledge domain

When we design more complicated functions, we design many helper functions using the different rules to divide the work of the main function.

Function Composition

- For the first, top-level function, if function composition is needed, **discard the entire template**
  - The new template tag is `(@template fn-composition)`
  - New template is `(third-fn (second-fn (first-fn <parameter>)))`
- Tests
  - Tests do not need to fully exercise the child functions; but they do need to exercise the **combination** of the functions
  - Base cases do not need to be tested

Operating On Arbitrary Sized Data

- If a function needs to operate on an entire list–and not just the first element–you must create a (recursive) helper function.
- Sometimes the signature can't say everything that matters about the consumed data. In those cases we can write an **assumption** as part of the function design.
- Trust the natural recursion

  - Assume that the RESULT self-reference function call WILL produce what it's supposed to.
  - When a function called `sort-images` calls itself in the function, assume `(sort-images (rest list))` is already sorted.
  - This assumption is inherited by the helper function!

Knowledge Domain Shift

- When the body of a function must shift to a new knowledge domain it should call a helper function to do the work in the new domain.

In the context of the video example, `insert` has a knowledge domain shift:

- `insert` is a function about inserting into a sorted list, while
- the `if` statement's question is about comparing the "sizes" of two images.
- This is a **knowledge domain shift**! We must write a helper function to do the work in the new domain (comparing sizes of images)

Videos: overview tree diagram

```
      arrange-images
       /         \
 sort-images   layout-images
   /
insert
  |
larger
```

- `arrange-images`: function composition, causes
  - `layout-images`
  - `sort-images`: operate on arb-sized data, causes
    - `insert`: knowledge domain shift, causes
      - `<area>?`

Notes about these functions:

- sort-images inserts the first element of the list into the rest of the **sorted** list (non-decreasing by size/area)
- insert assumes that `lst` is sorted and inserts `image` in it's proper place,

Other notes

- In larger programs, tests and constants would be in a separate file

- Functions with additional atomic parameters (`add-param`):
  - *"Note that when designing functions that consume additional atomic parameters, the name of that parameter gets added after every `...` in the template. Templates for functions with additional complex parameters are covered in Functions on 2 One-Of Data."*

What an `add-param` template looks like

Before adding an atomic parameter:

```
(@template ListOfImage)
(define (insert loi)
  (cond [(empty? loi) (...)]
        [else
         (... (first loi)
              (insert (rest loi)))]))
```

After adding an atomic parameter:

```
(@template ListOfImage add-param)
(define (insert img loi)
  (cond [(empty? loi) (... img)]       ; + to base case
        [else
         (... img                      ; + to combination
              (first loi)
              (insert (... img)        ; + to nat. recursion
                      (rest loi)))]))
```

Simpler example

A simpler example, before adding an atomic parameter:

```
(@template String)
(define (my-fun str1)
  (... str1))
```

After adding an atomic parameter:

```
(@template String add-param)
(define (my-fun str1 str2)
  (... str1 str2))
```