CPSC 110 Concepts Review

CPSC 110

Peyton Seigo

2018-11-5

CPSC 110 Concepts Review

Concepts I need to practice

- Backtracking: if not false
- Binary search trees
- Generative recursion

Useful built-in functions

- `(string-length s) -> Natural`
- `(substring s i j) -> String`
- `(string-ith s i) -> String (one long)`
- `(string-downcase s) -> String`
- `(string-upcase s) -> String`

Concepts

Core recipes:

- HtDF
- HtDD
  - Self-ref
  - Mutual-ref
  - Cyclic data !!!
- HtDW
  - Main function: `(@template htdw-main)`

Data driven templating:

- **Compound data**
- **Ref**
- **Self-ref** (lists)
  - **Naturals**: treating naturals as lists
- **Mutual-ref** (lists of non-primitive types)
- **Helpers**
  - `fn-composition`?
- **Binary Search Trees**
- Cross-product tables: **Two `one-of` types**
- **Local**
  - 4 uses of local:

    * Encapsulation
    * Reduce recomputation
    * Readability, D.R.Y.
    * To pass to an abstract function
- **Abstraction**
- **Generative Recursion**
  - **Search** w/ genrec

Questions

- NOTE: see if there is a syllabus with all the outcomes and skills.
- Q: If designing a tail recursive function with self-ref template AND accumulators, do you use `encapsulated` AND `accumulator` for both, or just `accumulator`? Because we combine the two templates into one `local`.
  - A: You just use `accumulator`.

`(require spd/tags)`: @ tags

HtDF

- `(@HtDF FunctionName)`
- `(@signature Type1 Type2 ... -> ResultType)`
- `(@template s1 s2 ...)`
  - `s` is a source for a template

Sources for `@template`

- `TypeName`
  - Name of type the template is based on.
  - For encapsulation: Separate `TypeName` for each encapsulated function.
- `(listof <TypeName>)`
  - List of some type. Can be used in `signature` as well.
- `add-param`
  - Additional parameters are treated as atomic data.
  - Add parameter to each `...`, like `(... ad-t1 ad-t2)` etc.
- `htdw-main`
  - `main` fn in HtDW, with a call to big-bang.
- `fn-composition`
  - Composition of calls to 2+ helper functions.

- `backtracking` search
- `2-one-of`
  - Cross-product table.
  - Possible case reduction.
- `encapsulated`
  - Encapsulation of 2+ fns.
  - Usually mutually recursive.
- `use-abstract-fn`
  - Call to 1+ abstract fns, either built in or user defined.
  - If more than 1, use `fn-composition`
- `genrec`
  - Generative recursion.
- `bin-tree, arb-tree`
  - Requires use of genrec, and indicates that template is a traversal of a generated binary or arbitrary-arity tree.
- `accumulator`
  - 1 or more accumulators
- `for-each`
  - Call to `for-each`.

HtDW Handlers

Template for key & mouse handlers use the *large enumeration rule*. So the tag and template look like this:

```
(@template KeyEvent add-param)
(define (handle-key ws ke)
  (cond [(key=? ke " ") (... ws)]
        [else (... ws)]))
```

Templates

Binary Search Tree

```
(@HtDD BST)
(define-struct node (key val l r))
;; BST (Binary Search Tree) is one of:
;;  - false
```

```
;;  - (make-node Natural String BST BST)
;; interp. false means no BST, or empty BST
;;         key is the node key
;;         val is the node val
;;         l and r are left and right subtrees
;; INVARIANT: for a given node:
;;     key is > all keys in its l(eft)  child
;;     key is < all keys in its r(ight) child
;;     the same key never appears twice in the tree

;; !!! examples

(@dd-template-rules one-of
                    atomic-distinct ; false
                    compound        ; (make-node Natural String BST BST)
                    self-ref        ; (node-l) is BST
                    self-ref)       ; (node-r) is BST
(define (fn-for-bst bst)
  (cond [(false? bst) (...)]
        [else (...
               (node-key bst)
               (node-val bst)
               (fn-for-bst (node-l bst))
               (fn-for-bst (node-r bst)))]))
```

No special `@template` tags, just one for the type BST.

Arbitrary Arity Tree

An arbitrary arity tree for filesystem elements.

```
(@HtDD Element ListOfElement)
(define-struct elt (name data subs))
;; Element is (make-elt String Integer ListOfElement)
;; interp. An element in the file system, with name, and EITHER data or subs.
;;         If data is 0, then subs is considered to be list of sub elements.
;;         If data is not 0, then subs is ignored.
```

```
;; ListOfElement is one of:
;;  - empty
;;  - (cons Element ListOfElement)
;; interp. A list of file system Elements

;; !!! examples

(define F1 (make-elt "F1" 1 empty))
(define F2 (make-elt "F2" 2 empty))
(define F3 (make-elt "F3" 3 empty))
(define D4 (make-elt "D4" 0 (list F1 F2)))
(define D5 (make-elt "D5" 0 (list F3)))
(define D6 (make-elt "D6" 0 (list D4 D5)))
(define MT-DIR (make-elt "MT-DIR" 0 empty))

(@dd-template-rules compound
                    ref)
(define (fn-for-element e)
  (... (elt-name e)
       (elt-data e)
       (fn-for-loe (elt-subs e))))

(@dd-template-rules one-of
                    atomic-distinct
                    ref
                    self-ref)
(define (fn-for-loe loe)
  (cond [(empty? loe) (...)]
        [else
         (... (fn-for-element (first loe))
              (fn-for-loe (rest loe)))]))
```

Built-in Abstract Functions

```
CONSUMES         PRODUCES     |  ABSTRACT FUNCTION
------------------------------+-----------------
Natural       -> (listof X)   |  build-list
```

```
(listof X)    -> (listof X)  |  filter
(listof X)    -> (listof Y)  |  map
(listof X)    -> Boolean     |  andmap
(listof X)    -> Boolean     |  ormap
Y (listof X) -> Y            |  foldr
Y (listof x) -> Y            |  foldl
```

Signature + purpose for each built-in abstract function according to *Language* page.

```
(@signature Natural (Natural -> X) -> (listof X))
;; produces (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

(@signature (X -> Boolean) (listof X) -> (listof X))
;; produce a list from all those items on lox for which p holds
(define (filter p lox) ...)

(@signature (X -> Y) (listof X) -> (listof Y))
;; produce a list by applying f to each item on lox
;; that is, (map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
(define (map f lox) ...)

(@signature (X -> Boolean) (listof X) -> Boolean)
;; produce true if p produces true for every element of lox
(define (andmap p lox) ...)

(@signature (X -> Boolean) (listof X) -> Boolean)
;; produce true if p produces true for some element of lox
(define (ormap p lox) ...)

(@signature  (X Y -> Y) Y (listof X) -> Y)
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base lox) ...)

(@signature  (X Y -> Y) Y (listof X) -> Y)
;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base lox) ...)
```

Other concepts

- HtDW
- Binary Search Trees
- Arbitrary Arity Trees
- Built-in Abstract Functions
- Writing Abstract Fold Functions
- Generative Recursion + Arbitrary Arity Tree + Backtracking Search
- Tail Recursion
- Accumulators
- Graphs
- Mutation