Module 0: The First Day

CPSC 110

Peyton Seigo

2018-09-13

Professor

- Professor: Dr. K, Dr. Kemi, or DR. Ola
- Oluwakemi Olamudzengi
- Course coordinator: Trey Schiefelbein – cpsc110-admin@cs.ubc.ca

Module 1: Beginning Student Language

CPSC 110

Peyton Seigo

2018-09-13

Module 1: Beginning Student Language

- **Expression**: a value, primitive call, control structure; evalutes to a value
- **Primitive call, or "call to primitive"**: expression with an operator and operand(s)
  - `(<id> <expr>* )` where the number of `<expr>`s determines the number of arguments supplied to the function named by `<id>`.
  - Composed of a *function identifier* (a primitive) and one or more *expressions*
  - Evaluation: first reduce operands to values, then apply primitive to values
  - "*call to string/image/number primitive*", based on parameter type
- **Function definition**: `define` with a function name so that it may be called
  - Returns value of the last expression
  - "*Racket programmers prefer to avoid side-effects, so a definition usually has just one expression in it's body*" (2.2 Simple Definitions and Expressions)
- **Function call or "procedure application"**: call to a defined function
- **Predicate**: a primitive or function that takes an input and returns a boolean value

Module 2: How to Design Data

CPSC 110

Peyton Seigo

2018-09-19

Module 2: How to Design Data

In summary, a **data definition** describes:

- how to form data of a new type
- how to represent information as data (information into data)
- how to interpret data as information (information from data)
- template for operating on data

Terminology

- cond is a **multi-armed conditional**: can have any number of cases, all at the same level
- **Problem domain**: contains information about a problem (e.g. light is red)
  - every program has a problem domain
- **Program**: uses data to represent information in the problem domain
- **Data definition**: describes how information is represented as data
  - *type comment* defines a new type name
  - body shows how to form data of that type
  - *interpretation* explains how to interpret data of this type as information, thereby establishing the information/data correspondence
  - *template* skeleton for one-argument functions that consume data of this type
    - demonstrates each possible case for the data type
- **Atomic information**: can't be taken apart into pieces AND still be meaningful in the problem domain
  - e.g. the city name "Vancouver" can be broken into V-a-n-c-o-u-v-e-r, but they are not meaningful to a city name (whereas a city itself is meaningful to a province, country, etc.)
  - *"the elapsed time since the start of the animation, the x coordinate of a car or the name of a cat"*
- **Orthogonality**: the HtDF (and HtDW) recipes work will all forms of data. the recipe is mostly orthogonal to the form of data.
- **Interval**: an interval of Numbers, Integers, or Naturals
- **Enumeration**: used when the information in problem domain consists of a fixed number of distinct items
  - *e.g. colours, letter grades, etc.*
  - Each "one-of" is a subclass
  - Do NOT collapse subclasses into a single `cond` case
  - Any data *can* be used, but strings should always be used
  - Interp. is often redundant, examples are nearly always redundant

- **Itemization**: is comprised of 2 or more subclasses, at least one of which is NOT a distinct data item
  - **Rule 1**: If a given subclass is the last subclass of its type, we can reduce the test to just the guard, i.e. `(number? n)`.
  - **Rule 2**: If all remaining subclasses are of the same type, then we can eliminate all of the guards.
  - **WARNING**: in a mixed data itemization template, the type specific predicates (i.e. <=) must be guarded against being called on the wrong type of data.
    * For example, `Integer[1, 10]` should test `(number? n)` if it's the only subclass with numbers, or `(and (number? n)(>= 1 n 10))` if there are multiple subclasses with numbers.
  - **Functions operating on itemizations**: should have at least as many tests as there as cases in the itemization. In the case of adjoining intervals, **it is critical to test the boundaries**.
  - If there are any discrete `string`s as the last subclasses, use an **`else`** instead of `(string =? n "...")`.
  - Always assume the user follows your data definition. Don't do more checks than you need to.

Syntax and structures

- **cond**: expression that has different behaviour based on any number of predicates
- `#;` comments out the entire expression or definition that follows the `#;`
- **`()` and `[]` are equivalent**: `[]` is used with `cond` cases by convention for clarity
- **`Integer[0, 33)`**: a range of `Integer`s from `0` (inclusive) to 33 (non-inclusive)

How to Design Data (HtDD) Recipe

Notes

- Anything to help understand what a data type represents belongs in the interpretation
  - e.g. for movie theatre seats, *"1 and 32 are aisle seats"*

Recipe

The first step of the recipe is to identify the inherent structure of the information.

Once that is done, a data definition consists of four or five elements:

1. A possible structure definition (not until compound data)
2. A type comment that defines a new type name and describes how to form data of that type.

3. An interpretation that describes the correspondence between information and data.
4. One or more examples of the data.
5. A template for a 1 argument function operating on data of this type.

```
;; Data definitions:

(@HtDD SomeType)
;; SomeType is Natural
;; interp. the airspeed velocity of an unladen swallow
(define ST1 24)
(define ST2 10)

(@dd-template-rules atomic-non-distinct)
(define (fn-for-some-type st)
  (... st))

;; Function definitions:

(@HtDF survive?)
(@signature SomeType -> Boolean)
;; produce true if given 24
(check-expect (survive? 24) true)
(check-expect (survive? 0) false)

; (define (survive?) false) ; stub

(@template SomeType) ; copied from data def. & modified in place
(define (survive? st)
  (= st 24))
```

Module 3a: How to Design Worlds

CPSC 110

Peyton Seigo

2018-09-19

Module 3a: How to Design Worlds

- Be able to explain the inherent structure of interactive graphical programs.
- Be able to use the How to Design Worlds (HtDW) recipe to design interactive programs with atomic world state.
- Be able to read and write big-bang expressions.

How to Design Worlds (HtDW) Recipe

World program design is divided into two phases, each of which has sub-parts:

1. Domain analysis (use a piece of paper!)
    1. Sketch program scenarios
    2. Identify constant information
    3. Identify changing information
    4. Identify big-bang options
2. Build the actual program
    1. Constants (based on 1.2 above)
    2. Data definitions using HtDD (based on 1.3 above)
    3. Functions using HtDF
        1. main first (based on 1.3, 1.4 and 2.2 above)
        2. wish list entries for big-bang handlers
    4. Work through wish list until done

Working through the recipe

- `empty-scene` is a primitive that allows you to create a background
- HtDF
    - Use the world constants in `check-expect`s
        * Clarity + correctness if constants change
    - Using named constants provides a "single point of control"
    - **Large enumeration rule** (i.e. `KeyEvent`)
        * only include `cond` cases the function cares about
        * other cases are handled by an `else`
- HtDD
    - It is important to state units in the interpretation
- HtDW
    - Work on this process until the flow from one recipe to the next is SECOND NATURE!
    - **Wish list entry**: a big-bang handler's signature, purpose, `!!!`, and stub

  - The domain analysis is a **model** of the program
    * Improve program by marking up analysis

Notes

Interactive behaviour is generally defined as:

- changing state
- changing display
- keyboard and/or mouse affects behaviour

Terminology

- `big-bang` is **polymorphic**: it can work for any type of world state
  - an interface that works for many different types of data

Module 3b: Compound Data

CPSC 110

Peyton Seigo

2018-09-21

Module 3b: Compound Data

- Be able to identify domain information that should be represented as compound data.
- Be able to read and write define-struct definitions.
- Be able to design functions that consume and/or produce compound data.
- Be able to design world programs that use compound world state.

Notes

- `define-struct`: defines four general definitions
  - Definitions:
    - the **struct** itself
    - **constructor**: `make-<struct-name>`
    - **selector(s)**: `<struct-name>-<field-name>`
      - A unique selector is created for each field name
    - **predicate**: `<struct-name>?`
  - `(define <struct-name> (x y))`
    - `x` and `y` have given this `struct` two field names
  - Define a `<struct-name>` struct using:
    - `(define S1 (<struct-name> x y))`
    - `x` and `y` set values for the field names

Data definition example:

```
(@HtDD Movie)
(define-struct movie (title budget year))
;; Movie is (make-movie String Natural Natural)
;; interp. metadata for a movie
;;         title is the movie's title
;;         budget is the movie's production budget
;;         year is the year the movie was released
(define M-TITANTIC (make-movie "Titantic" 200000000 1997))
(define M-ELEMENT (make-movie "The Fifth Element" 90000000 1997))
(define M-AVATAR (make-movie "Avatar" 237000000 2009))
(define M-AVENGERS (make-movie "The Avengers" 220000000 2012))

(@dd-template-rules compound)
(define (fn-for-movie m)
  (... (movie-title m)     ; String
       (movie-budget m)    ; Natural
```

```
          (movie-year m)))    ; Natural
```

and a function implementing `Movie`:

```
(@HtDF budget?)
(@signature Movie -> Boolean)
;; produce true if movie's budget is < 100 000 000 (USD)
(check-expect (budget? M-TITANTIC) false)
(check-expect (budget? M-ELEMENT) true)
(check-expect (budget? (make-movie "Moonrise Kingdom"
                                   16000000
                                   2012))
              true)

;(define (budget? m) false) ; stub

(@template Movie)
(define (budget? m)
    (< (movie-budget m)
       100000000))
```

Terminology

Module 4a: Self-Reference

CPSC 110

Peyton Seigo

2018-09-26

Module 4a: Self-Reference

Learning goals

- Be able to use list mechanisms to construct and destruct lists.
- Be able to identify problem domain information of arbitrary size that should be represented using lists and lists of structures.
- Be able to use the HtDD, HtDF and Data Driven Templates recipes with such data.
- Be able to explain what makes a self-referential data definition well formed and identify whether a particular self-referential data definition is well-formed.
- Be able to design functions that consume and produce lists and lists of structures.
- Be able to predict and identify the correspondence between self-references in a data definition and natural recursions in functions that operate on the data.

Notes

- Lists
  - A list is an itemization containing `atomic-distinct` and `compound` data.
  - Lists are **self referential**.
- The self-reference template rule puts a natural recursion in the template that corresponds to the self-reference in the type comment.
  - *The `dd-template-type` is called `self-ref`*
- `(check-expect)`s for lists
  - Examples shold include base and self-referential cases.
  - Have one or more tests with a list of 2 or more elements
- Remember, constants support "single-point of control" in that it's very easy to change their values later on!

Terminology

- **Arbitrary-sized information**: information that we don't know the size of in advance.
  - *A program that can display any number of cows is operating with abitrary-sized information.*
- **Self-reference**: the relationship between an itemization's case that refers to the data definition itself and the data definition
  - Causes Natural Recursion in template
- **Well-formed self referential data definition**:
  - At least one base case (allows self referential case to end)
  - At least one self referential case

- **Self reference rule**: if an itemization data definition has a `one-of` case that refers to itself, put a natural recursion in the template that corresponds to the self-reference in the type comment
  - `@dd-template-rules` rule is called `self-ref`
- **Natural recursion**: the relationship within a self-referencing data type's template where the template actually refers to itself!
  - i.e. a template's function named `fn-for-los` will call itself within its own definition using `(fn-for-los (rest los))`
  - Caused by self-reference rule

The cons primitive

The primitive `cons` is a two element constructor that constructs a list:

```
(cons x y) -> list?
  x : any/x
  y : list?
```

`cons` can be used to produce lists with more than one type of data; but we will not do that (our data definitions do not let us talk about that very well).

Other primitives that operate on lists

Lists have functions that are SIMILAR to `struct` selectors:

- `(first <list>)`: first element in list
- `(rest <list>)`: list with front popped off
  - *Note: `rest` expects a non-empty list*
  - `(first (rest L2))`: produces element in `<list>`
    * pops element off the front of `L2`, then gets the first element in the new list
    * `(second <list>)` also exists, but popping and getting the first element as shown above is VERY useful in things like recursion and using accumulators! It's mostly useful because the procedure is generalized.
- `(empty? <list>)`: produce true if argument is the empty list
- `(length <list>)`: evaluates number of items on a list

Lists containing primitive data

Example of a list data definition containing primitive data:

```
(require spd/tags)

(@HtDD ListOfNumber)
;; ListOfNumber is one-of:
;;  - empty
;;  - (cons Number ListOfNumber)
;; interp. a list of numbers
(define LON1 empty)
(define LON2 (cons 12 empty))
(define LON3 (cons 6 (cons 12 empty)))

(@dd-template-rules one-of          ; 2 cases
                    atomic-distinct ; empty
                    compound        ; (cons Number ListOfNumber)
                    self-ref)       ; (rest lon) is ListOfNumber
(define (fn-for-lon lon)
    (cond [(empty? lon) (...)]
          [else
            (... (first lon)
                 (fn-for-lon (rest lon)))]))
```

And a function implementing our new list:

```
(@HtDF sum)
(@signature ListOfNumber -> Number)
;; produce the sum of the given list
(check-expect (sum empty) 0)
(check-expect (sum (cons 5 empty)) 5)
(check-expect (sum (cons 10 (cons 5 empty))) 15)

;(define (sum lon) 0) ; stub

(@template ListOfNumber)
(define (sum lon)
    (cond [(empty? lon) 0]
          [else
            (+ (first lon)
               (sum (rest lon)))]))
```

Module 4b: Reference

CPSC 110

Peyton Seigo

2018-09-30

Module 4b: Reference

Learning goal

- Be able to predict and identify the correspondence between references in a data definition and helper function calls in functions that operate on the data.

Notes

- When a data definition uses the reference rule, the `@dd-template-rules` tag is `ref`

Terminology

- **Reference relationship**: data definition that refers to a different type of data (that's not primitive!)
- **Reference rule**: for data definitions with a reference to another data definitions that you've defined
  - Rule: must wrap calls to referenced definition in that definition's template function (called a **natural helper**)
- **Natural helper**: a referenced data definition's template function due to the reference rule
  - A natural helper in a template says "do something complicated in a helper function that consumes the referred to type. do NOT do it here!"
  - HtDF: create a **helper function** for the natural helper
    - wish list entry: `@HtDF`, `@signature`, purpose, stub, and `!!!`
- **Helper function**: actual function written when doing HtDF
- **complicated? rule**: if it would take more than 1 function that operates on the referenced type, make a helper function instead.

Lists containing non-primitive (user-defined) data

Example of a list data definition containing non-primitive data defined by the user. Pay attention to

- How the self-reference rule applies to `ListOfSchool`
  - the `dd-template-rules` tag is `self-ref`
  - `(rest los)` is non-primitive and is a `ListOfSchool`, so it is wrapped in `ListOfSchool`'s template function
    - this is **Natural Recursion**
    - when writing a function that consumes `ListOfSchool`, this will be a **Recursive Call**
- How the reference rule applies to `ListOfSchool`

- the `dd-template-rules` tag is `ref`
- `(first los)` is non-primitive and is a `School`, so it is wrapped in `School`'s template function
    * this is a **Natural Helper**
    * when writing a function that consumes `ListOfSchool`, you must write a **Helper Function** (unless you're not performing any operations on that data)

- There are **two** data definitions, not just one

```
(@HtDD School)
(define-struct school (name tuition))
;; Bar is (make-bar Natural String)
;; interp. properties of a university
;;          name is abbreviated name of the university
;;          tuition is yearly undergraduate tuition (CAD) of the
   university
(define S-UBC (make-school "UBC" 25000))
(define S-UOA (make-school "UAlberta" 16000))
(define S-UOC (make-school "UCalgary" 8500))

(@dd-template-rules compound)
(define (fn-for-school s)
  (... (school-tuition s)
       (school-name s)))


(@HtDD ListOfSchool)
;; ListOfSchool is one of:
;;  - empty
;;  - (cons School ListOfSchool)
;; interp. a list of schools
(define LOS1 empty)
(define LOS2 (cons S-UBC (cons S-UOA (cons S-UOC empty))))

(@dd-template-rules one-of          ; 2 cases
                    atomic-distinct ; empty
                    compound        ; (cons School ListOfSchool)
                    ref             ; (first los) is School
                    self-ref)       ; (rest los) is ListOfSchool
(define (fn-for-los los)
  (cond [(empty? los) (...)]
        [else
         (... (fn-for-school (first los))
              (fn-for-los (rest los)))]))
```

Module 5a: Naturals

CPSC 110

Peyton Seigo

2018-10-02

Module 5a: Naturals

Learning goals

- Be able to design functions that operate on natural numbers.
- Be able to design a simple alternative representation for natural numbers.

Notes

For the `Natural` data definition in 01-naturals-starter.rkt:

- Template rules wouldn't normally put `n` by itself in the template, but because our funny "compound" data does NOT have a `(first lon)` equivalent, we end up adding `n` ourselves.
  - There must be a way for the first element to **contribute its result**!

For `to-list` with signature `Natural -> ListOfNatural` in 01-naturals-starter.rkt:

- Since ListOfNatural is the produced type (as opposed to the consumed type), and the type is so simple, we do not need to write a template for it.

Terminology

- x

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))
```

Module 6a: Binary Search Trees

CPSC 110

Peyton Seigo

2018-10-12

Module 6a: Binary Search Trees

Learning goals

- Be able to reason informally about the time required to search data.
- Be able to identify problem domain information that should be represented using binary search trees.
- Be able to check whether a given tree conforms to the binary search tree invariants.
- Be able to use the design recipes to design with binary search trees.

Notes

Linear search:

- Best case: `1` operation, constant O(1)
- Worst case: `n` operations, linear O(n)
- Average case: `n/2` operations, linear O(n)

Binary Search Trees:

- A binary tree (every node has 0, 1, or 2 children) with the properties:
  - ALL nodes in left sub-tree have a value less than the parent
  - ALL nodes in right sub-tree have a value greater than the parent
- These rules are invariant; holds true ALL the way down a branch!

Functions using a BST

- Some function may produce `Type or false`.
  - `lookup-key`, a function that searches for a node with a particular key and returns the node's value, may not find the node.
  - Signature for `lookup-key` is `BST Natural -> String or false`

Testing a BST

- We have 2 self-reference cases, so our check-expects should check both cases (i.e. left and right)
- Just like for lists, tests should test "**2-deep**"
  - Test all four scenarios for traversing a BST two levels
  - Left Left, Left Right, Right Left, Right Right

Module 6b: Mutual Reference

CPSC 110

Peyton Seigo

2018-10-15

Module 6b: Mutual Reference

Learning goals

Learn how to use multiple mutually referential types.

- Be able to identify problem domain information of arbitrary size that should be represented using arbitrary arity trees.
- Be able to use the design recipes to design with arbitrary arity trees.
- Be able to use the design recipes with mutually-referential data.
- Be able to predict and identify the correspondence between external-, self- and mutual-reference in a data definition and calls, recursion and mutual-recursion in functions that operate on the data.

Notes

- Mutually recursive data: Arbitrary-arity trees
  - Requires two cycles in the type reference graph
  - Due to arbitrary size in 2 dimensions
- Getting stuck: strategies to get unstuck
  - Make sure you have examples for what you are trying to write.
  - If you missed some examples at the beginning, GO WRITE THEM IN when you get to a situation that isn't covered!
  - Do it on paper.

01-Mutually-Recursive Data (video)

- `ListOfElement`
  - self-reference (SR) cycle: allows directory's list of sub-elements to be arbitrarily long
  - reference to `Element`: mutual reference (MR)
- `Element`
  - reference to `ListOfElement`: mutual reference (MR)
- The **mutual reference cycle** allows each element (or node) to have an arbitrary number of sub-elements (or children)
  - i.e. allows tree to have arbitrary breadth
  - ONLY *Mutual Reference* (MR) if both types reference each other. Otherwise, it is just a reference.
- There are a few "base cases" for this tree for which it stops growing. One or more of these must be the case.

1. When an element has non-zero data. That node cannot have children.
2. When an element has zero data and an empty list.
3. When an element has zero data and a list with elements with non-zero data. The element's children will not have children (no grandchildren for you!).

HTDF for Mutually Recursive Data**

- We don't design a single function. We design a function for EACH type.
- Function naming convention: `<base-fn-name>--<data-type>`
  - All functions have a base name (eg. `sum-data`) with the type that is being operated on as a suffix (eg. `element` or `loe`)
  - eg. `sum-data--element` and `sum-data--loe`
  - All functions are named `base-fn-name` because they are **mutually recursive** & require each other to work
- Functions usually all produce the same data (but there are exceptions)
- `spd/tags` tags
  - HtDF tag at top includes all functions
    * `(@HtDF <fn>--<type1> <fn>--<type2> ... <fn>-<typen>)`
  - Separate signatures for each functon (both above purpose)
  - Separate template tags for each function (above each function)

Why does it work? Because our method is data-driven, we do all the hard work with our data definitions.**

1. Well-formed, self and mutually referential type comments
2. Templates support natural mutual recursion (NMR)
3. Derived functions will
   - have the right structure, and
   - terminate in a base case

Backtracking

Three main things about backtracking:

1. Signature produces a `Type or false`
2. Function body of fn. consuming `ListOfX` has:
   - `(if (not (false? (find--region 1 (first lor)))))`
   - This "if not false?" pattern is important for generic functions.
   - We could instead use `region?`, but this would only work for a tree of `region`s.

3. Backtracking tag: `(if (not (false?` is a structural characteristic of all backtracking problems
    - Add `backtracking` to each function's template tag
    - `(@template Region add-param backtracking)`
    - `(@template ListOfRegion add-param backtracking)`

Terminology

- Arbitary-arity tree: nodes can have an arbitrary number of children
    - Arbitarily deep: an unknown number of levels
    - Aribitarily "wide": an unknown number of children

Off-topic Questions

In Racket, For a search function, we can produce `Value or false` which represents two cases:

1. Success! We find the key and produce its value.
2. Failure. We do not find the key and return `false`.

How might we transfer this to other languages like C++ or Java?

In C++, there is a find function for vector lists. It behaves as such:

1. If found element, returns iterator to it.
2. Otherwise, return iterator to the last element.

Is there a better way? I have often found myself being unsure of how to represent a "failing" case. The solutions I have used are either,

1. Return 0, -1, or some other meaningless value. Give this value meaning where the function is called (i.e. user is responsible for implementation details)
2. Use an `Enum` to give meaning to arbitrary integers.
3. Throw an exception. (I assume this is the best method)

Module 7a: Two One-Of Types

CPSC 110

Peyton Seigo

2018-10-19

Module 7a: Two One-Of Types

Learning goals

- Be able to produce the cross-product of type templates table for a function operating on two values with one-of types.
- Be able to use the table to generate examples and a template.
- Be able to use the table to simplify the function when there are equal answers in some cells.

Function Model: Cross Product of Type Comments Table

- Cases of the `one-of` type comments go along the axes
  - The two arguments don't have to be the same type, but both must be a `one-of`
- Generate tests for each cross product cell in the table

Example for `ListOfString ListOfString -> x`:

los1 (right) los2 (down)	empty	(cons String ListOfString)
empty	true	false
(cons String ListOfString)	true	and firsts are equal; natural recursion

In the video, we only had one test for each box, except for the bottom right. For "both lists are not empty," we generated several more tests to satisfy **2-deep**, los1 longer than los2, los1 shorter than los2, and a few more conditions where the function passes or fails with 2 or 3 deep lists.

Models

- Type comments predict the templates
  - A type comment is a **model** of the functions operating on that type
  - Non-code representation of the program
  - Tells us what the function will look like
- Using a cross product of type comments table: simplifying at a **model level**
  - Simplifying without looking at details of code

Module 7b: Local

CPSC 110

Peyton Seigo

2018-10-20

Module 7b: Local

Learning goals

- Write well-formed local expressions
- Diagram lexical scoping on top of expressions using local
- Hand-evaluate local expressions
- Use local to encapsulate function definitions ("private" helper functions)
- Use local to avoid redundant computation

Local

The `local` function is comprised of *local definitions* and a *body*.

- Must have 0 or more definitions inside square brackets `[...]`
- Must have a body

```
(local [<definition 1> ; local definitions
        <definition 2>
        ...]
  <expression>)        ; body
```

Local Evaluation Rules

Three steps happen at the same time when evaluating a local expresson.

1. Renaming
   - rename definitions and all references to definitions
   - the new name must be globally unique
2. Lifting
   - lift renamed definitions out of the `local`, into top-level scope (not just out of the expression!)
3. Replace entire `local` with renamed body
   - After evaluation, the `local` expression is gone!

See `02-evaluation-rules.rkt` for a step-by-step evaluation.

Encapsulation

Finding good candidates for encapsulation:

1. One function has 1+ helpers closely linked to it
2. Outside program only cares about the main function, not the helpers

When refactoring existing code, make sure to

- Encapsulate
  - Open function with new global name + necessary parameters
  - Wrap old functions in `[]`
  - Add "trampoline" call to the appropriate function
  - Write one `@template` tag with `encapsulated`:
    * `(@template <Type1> <Type2> ... <TypeN> encapsulated)`
- Renaming
  - Rename `check-expect`s
  - Renamestubs
  - `@HtDF` tag
- Delete unnecessary pieces
  - Delete tests for hidden functions (may lose some base cases)
  - Delete signatures that don't apply anymore
  - Delete old stubs

Structure changes. Functionality does NOT change.

Advantages and Disadvantages of Using `local` for Encapsulation

Advantages:

- Templates can be pre-encapsulated, saving time later on
- Template functions inside `local` don't have to be renamed! That's right, you can keep it as `fn-for-element` and `fn-for-loe` (for example).

Disadvantages:

- Cannot write base case tests for helper functions
  - Can only test the whole function
  - However, at this point in the course, we may not need to actually test the absolute base case test first

Terminology

- **Top-level definition**: definition visible to entire program
- **Local definition**: definition restricted to a certain scope
  - Within that scope, a local definition has precedence over any higher-level definitions

- **Lexical scoping**
  - **Scope contours**: boxes drawn around parts of the programming illustrating scopes
  - **Top-level scope**: global scope; scope of the whole program
  - Scope can be imagined as a bunch of nested boxes, or a tree where the top-most node is the global scope and each subnode is a scope within that scope.
- **Encapsulation**: bundling data with functions that operate on that data
- **Refactoring**: changing a program's code/structure without changing the program's behaviour
- **Namespace Management**: way to deal the problem of large programs inevitably using the same names
  - encapsulating many functions away so that the only public functions are ones with unique, descriptive names
  - ensuring other programmers don't call functions they're not supposed to

Module 8: Abstraction

CPSC 110

Peyton Seigo

2018-10-26

Module 8: Abstraction

Learning goals

- Be able to identify 2 or more functions that are candidates for abstraction.
- Be able to design an abstract function starting with 2 or more highly repetitive functions (or expressions).
- Be able to design an abstract fold function from a template.
- Be able to write signatures for abstract functions.
- Be able to write signatures that use type parameters.
- Be able to identify a function which would benefit from using a built-in abstract function.
- Be able to use built-in abstract functions.

Refactoring

- Remember to:
  - Pass new parameter in natural recursive calls
  - Replace point(s) of variance with variable(s)
    - Point of variance could be a value or function call

`(listof T)` instead of `ListOfT`

From now on, anytime you want a `ListOfT` type, you can use `(listof X)` instead of having to write a data definition for it.

The type `(listof T)` means:

```
;; ListOfT is one of:
;;  - empty
;;  - (cons T ListOfT)
;; interp. a list of T

(@dd-template-rules compound self-ref)
(define (fn-for-lot lot)
  (cond [(empty? lot) (...)]
        [else
         (... (first lot)
              (fn-for-lot (rest lot)))]))
```

Working through the recipe

With abstract functions, it gets **harder** as we go back towards the signature.

1. Write abstract function; replace bodies of original function(s)
2. Unit tests
    - Can be abstracted from `check-expect`s of original function(s); copy all original tests and narrow down what you need
    - Insert appropriate additional parameter into tests
    - Revising existing examples may be easier than starting from scratch
3. Purpose
    - Can sometimes be abstracted from orig. purpose(s), but not always
    - Purpose statements can take A LOT of tries to get right. This is normal.
4. Signature
    - Lists: use `(listof <Type>)`
    - Functions: use `(<Type1> -> <Type2>)`
    - Consuming a generic type: use *type parameters* for consumed/produced types
        - Lists: `(listof T)`, a list containing one or more `T` objects
        - Functions: `(T -> U)` such that `T` is some type consumed and `U` is some type produced
        - Example: `(@signature (T -> U)(listof T)-> (listof U))`.
            * The template is `(@template (listof T)add-param)`
            * The definition is `(define (map fn lon)...)`

HtDF: Writing functions that call abstract functions

Writing a function that calls an abstract function like `filter`, `map`, or an abstract function you write yourself.

- Template tag: `(@template use-abstract-fn)`
- Base case test is NOT needed when using built-in abstract functions–their base case is already tested

Functions that are **abstract functions themselves** and **consume** a generic function do not have any special `template` tags. Note that if an abstract function consumes a list, the template will have `(listof <T>)`.

Closures

A closure is a `local` function defined within a function body and **uses a parameter of its enclosing function**. In these cases, you MUST define the function using `local`.

Below, the helper `bigger?` is a closure. It “closes over” the value of `threshold` passed to `only-bigger`.

```
(define (only-bigger threshold lon)
  (local [(define (bigger? n)
            (> n threshold))]
    (filter bigger? lon)))
```

Terminology

- **Abstraction**: generalizing repetitive code (through refactoring)
  - Make programs smaller + easier to read
  - Separates knowledge domains more clearly in code
- **Abstract function**: a helper shared between multiple functions
  - More general than the original code
- **Abstraction from examples**: abstracting/generalizing functions that have already been written
  - Backwards HtDF recipe: *Function definition -> Tests -> Purpose -> Signature*
- **Higher order function**: can a) consume one or more function, and b) produce a function
- **Type parameter**: a name for some generic type; often used in an abstract function’s signature and template tags (`X`, `Y`, `Z`, `T`, `U`, etc.)

Built-in abstract functions

Template for writing a function that calls a built-in abstract function:

```
(@template (listof t) use-abstract-fn)
(define (some-fn lot)
  (<built-in-fn> ... lot))
```

Built-In Abstract Functions ISL and ASL have the following built-in abstract functions.

- `build-list`: make a list of elements based on their index (doesn’t have to be a list of numbers!)
  - `(build-list n identity)` makes a list of naturals for `[0, n)`
- `filter`: list of elements that satisfy a given predicate
- `map`: apply a fn to each element
- `andmap`: analogous to Elixir’s `all?`
- `ormap`: analogous to Elixir’s `any?`

- `foldl`
- `foldr`: reduce a list of elements to a single element
  - similar to Elixir's `reduce`, but `foldr` does not use an [apparent] accumulator
  - HexDocs actually says Reduce is sometimes called `fold`!

`foldr` is the abstract function for the `(listof T)` template:

Signatures for each built-in function:

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

(foldr ... ... lot)
```

- The first `...` in `foldr` is the **combination** in `fn-for-lot`
- The second `...` in `foldr` is the **base case** in `fn-for-lot`

List of signatures **without their function arguments** for comparing to functions that have NOT yet implemented an abstract function:

```
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) ...)
```

Fold functions

Going directly from type comments to abstract functions, rather than writing a bunch of redundant code and abstracting from examples afterwards.

Our fold functions are written from existing templates.

Tips

- For mutual ref templates, assign *type parameters* to the result of each function right away!
  - i.e. for `Element` and `ListOfElement`, let `fn-for-e -> X` and `fn-for-loe -> Y`
  - Makes writing the signature MUCH easier

Practice Midterm Mistakes

CPSC 110

Peyton Seigo

2018-10-09

Practice Midterm Mistakes

Mistakes I made on practice midterms.

Mistakes

- 2017W1-MT1
  - Includes `dd-template-rule` that doesn't apply in that situation (`compound`)
  - Does not write helper function in itemization template with `ref` rule
  - Wrong order for `substring` and `string-ith`: should be `(<primitive> String I1 < I2>)`
  - Code does something other than what the question asked
  - ** Does not pass all arguments in Natural Helper, especially ones that are constant **
- 2017W2-MT1
  - Does half a page of work the question does not ask for
  - Only does half of what question asks for

Style

- Write expression on a new line after the **`else`**
- Use `(zero? n)` instead of `(= n 0)` for recursive Natural functions
- Use given examples/constants!
  - Save time, easier to check, easier to grade

Uncertainties

- =, not =?
- Struct with reference: rules are `compound` and `ref`
- Use primitive type for templates consuming a primitive (`@template String`)
- Include `dd-template-rules` for ALL itemization cases, even if some are the same rule

How to prevent these mistakes

- Double check the connections in code
  - Check where template rules came from
  - Make sure `ref` and `self-ref` generate helper functions
- Memorize order of parameters for primitive functions, AND their names
- Read the question! Make sure you are doing what it asks.
- ** Make sure Natural Helper passes ALL arguments! **

Problem Set Mistakes

CPSC 110

Peyton Seigo

2018-10-02

Problem Set Mistakes

Mistakes I've lost marks for on problem sets.

Mistakes

- PSET2
  - **Style** 1/2: Type names should be with upper Cammel case
    - itemization string cases were lowercase
- PSET3
  - **Stub** 0/2: left stub value as `...`
  - **General style** 1/2: 80 character per line exceed