Systematic Program Design

Status Completed

Lecture 0: Introduction

12/25/23

Welcome

- Systematic program design refers to breaking a program down into small problems in order to come up with a well-structured solution.
- The best way to learn programming material is to learn by doing.
- Beginning Student Language (BSL) is a programming language has the functionality of most other languages, but in a simple package that makes it easy to learn.
    - We focus on it in this course because it is simple, and it has the core features of nearly all other languages.

Lecture 1a: Beginning Student Language

12/26/23

Expressions, part 1

- Using BSL, expressions can be written with the operator first.

```
; (<operator> <operands>...)
; Addition
(+ 3 2)
; Multiplication
(* 4 3)
; Sqaure root
(sqrt 16)
```

- Lines can be commented out with `;` at the beginning of a line.

```
; Comment
```

Expressions, part 2

- If a number is irrational, racket will display it with an `#i`, to show that it is an inexact number.

```
; Take the square root of 2
(sqrt 2)

; Racket prints an inexact number
> #i1.4142135623730951
```

Evaluation

- In an expression, the first primitive is the operator, and the following primitives are the operands.
  - Racket evaluates expressions by evaluating all operands to values.
  - Racket then applies primitives to the values.
- Racket goes from the most specific to the least specific expressions based on parentheses.

```
; Evaluation happens left to right, and from the inside to the outside
(+ 2 ( * 3 4) (- (+ 1 2) 3))
(+ 2 12 (- (+ 1 2) 3))
(+ 2 12 0)
14
```

Strings and images, part 1

- In Racket, strings are represented with double quotes.

```
; String
"Hello, World!"
```

- Strings can be concatenated with the operator `string-append`.

```
; Append two strings
(string-append "Hello, " "World!")
```

- String length can be taken with the operator `string-length`.

```
; Evaluates to 5
(string-length "apple")
```

- A substring can be taken with the operator `substring`.

```
; Evaluates to pl
; (substring <string> <start_char> <end_char + 1>)
(substring apple 2 4)
```

Strings and images, part 2

- The image library can be imported using `(require 2htdp/image)`.
- Shapes, such as circles, can be drawn using Racket.

```
; Draw a solid red circle with a radius of 10
(circle 10 "solid" "red")
; Draw an orange "hello" in pt. 24 font
(text "hello" 24 orange)
```

- The operator `above` appends images by stacking them above each other in sequence.

```
; Draw stacked circles of different sizes and colors.
(above (circle 10 "solid" "red")
       (circle 20 "solid" "yellow")
       (circle 30 "solid" "green"))
```

- The operator `beside` appends images by stacking them beside each other in sequence.
- The operator `overlay` appends images by stacking them on top of each other in sequence.

Constant definitions, part 1

- Constants can be defined with the keyword `define`.

```
; Define WIDTH as 400
; (define <name> <expression>)
(define WIDTH 400)

; Evaluates to 1600
(* WIDTH 4)
```

- Racket can define images as constants.

(define CAT )

```
; Rotate the image 10 degrees
(rotate 10 CAT)
```

Function definitions, part 1

- A function definition can be written with the keyword `define`.

```
; Define a function
(define (<name> <params>...)
    <expression>...)

; Call a function
(<name> <params>...)
```

Booleans and “if” expressions, part 1

- Booleans are represented with `true` and `false`.

```
; Evaluates to false
(= 3 5)
; Evaluates to true
(> 5 2)
```

- The operator `string=?` can be used to check the equality of two strings.

Booleans and “if” expressions, part 2-3

- If expressions can be creates using the `if` keyword.

```
; (if <question>
;     <if_true>
;     <if_false>)
(if (string=? "foo" "bar")
        "similar"
        "different")
```

Booleans and “if” expressions, part 3

- Boolean operators like `and`, `not`, and `or` can be used to evaluate expressions.

```
; Evaluates to false
(and true false)
; Evualates to ture
(not false)
; Evulates to true
(or true false)
```

Using the stepper

- The stepper is used to walk through the evaluation of a program to see what is happening step-by-step.
- The stepper is used to understand what is happening in a program an debug it if necessary.

Discovering primitives, part 1

- The helpdesk acts in Dr. Racket as the Racket documentation.

- This is equivalent to the “go to definition” option in VS Code.

▼ Lecture 1b: How to Design Functions

12/26/23

“How to design functions” recipe

- The “How to Design Functions” (HtDF) recipe systematizes the design of a function.
- Every step of the recipe is intended to help with all the steps after it.
- Looking for an inconsistency between the different steps can be enough to find a bug and make it clear what needs to be fixed.

1. We start with the signature, purpose, and stub.
    a. The signature declares what type of data the function consumes and produces.
    b. The purpose is a one line description of what the function produces in terms of what it consumes
        - A function purpose should be specific.
        - If the function produces a Boolean, define what case `true` is.
    c. The stub is a function definition that has:
        i. The correct function name,
        ii. The correct number of parameters,
        iii. And produces a dummy result of the correct type.
        - In Racket, functions that produce a Boolean should have a name that ends in a `?`.

```
; Signature
;; Type ... -> Type
; Purpose
;; 1 line desc.
; Stub
; (define (<func_name> <params>...) <dummy-value>)
```

2. The examples and tests come next.
    - This is equivalent to unit tests.
    - `check-expect` is used instead of `assert` like in Python.
    - Dr. Racket uses a separate test window that will show if the tests run, and if they fail or not.

```
; Examples/tests
(check-expect (<func_name> <args>...) <expected-value>)
...
```

3. After that comes the template/inventory.
    - The template is a function with the right name and parameter(s), along with a placeholder body.

- The purpose of the template is to give an outline of the final function definition.

```
; Template/inventory
(define (<func_name> <params>...)
    (... <params>...))
```

4. Now, the body of the function is coded.

```
; Template/inventory
(define (<func_name> <params>...)
    <expression>...)
```

5. Finally, the tests are run.
    - If the tests pass, that is all.
    - If not, the program is debugged, then tested again.
    - Test-coverage refers too how much of the code is evaluated through tests.
        - There should be enough tests to cover all the paths in a function.
        - Dr. Racket highlights any paths that have not been evaluated through test.

▼ Lecture 2: How to Design Data

12/26/23

"Cond" expressions

- `cond` is a multi-armed conditional that can have any number of cases all at the same level.
    - The `else` path is taken when none of the other conditionals are met.

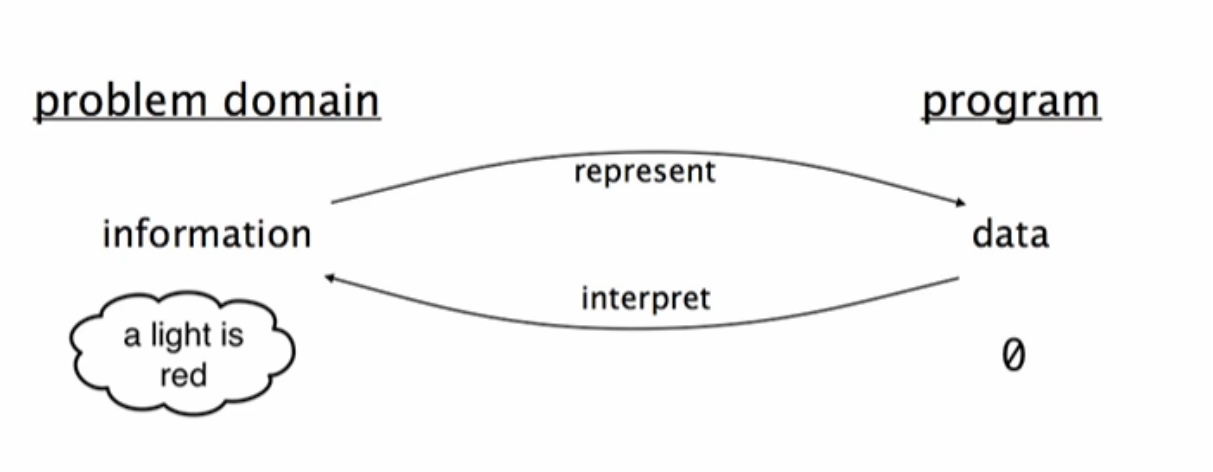
```
; This function uses a cond expression to take
; the absolute value of a number

; (cond [<question_expression> <answer_expression>]
;               ...)

; NOTE that square [] and round () brackets are interchangable,
; but the use of them splits up the problem nicely
(define (absval n)
  (cond [(> n 0) n]
        [(< n 0) (* -1 n)]
        [else 0]))
```

- `#;` can be used to comment out entire expressions and the definitions that follow them.

Data definitions

- In any program, we have what we call the problem domain that contains information.

- The program only contains data.
- Information in the problem domain is represented in the program as data.



- Data definitions describe how one represents information as data.
- Interpretation explains how to interpret data an information, thereby establishing information-data correspondence.

```
; Relate traffic light colors (information) to numbers (data)

;; Data definitions:

;; TLColor is one of:
;;  - 0
;;  - 1
;;  - 2
;; interp. 0 means red, 1 yellow, 2 green


; The TLColor data definition can be used in the signature later

;; TLColor -> TLColor
;; Produce next color of traffic light
```

"How to design data"

- "How to Design Data" (HtDD) gives an outline for systematically creating data definitions.
- Atomic information is information that cannot be broken down any further into constituent components.
- `;;` is by convention used for permanent comments.
- A data definition consists of five parts:
  1. A possible structure definition.

2. A type comment that defines a new type name and describes how to form the data.
3. An interpretation that describes the correspondence between information an data.
4. One or more examples of the data.
5. A template for 1 argument function operating on data of this type.

```
; Example data definition

;; CityName is String
;; interp. the name of a city
(define CN1 "Boston")
(define CN2 "Vancouver")
#;
(define (fn-for-city-name cn)
  (... cn))

;; Template rules used:
;;  - atomic non-distinct: String
```

HtDF x HTDD structure of data orthogonality

- The HtDF and HtDD recipes are mostly orthogonal.
  - This means that they both work nearly the same way for all forms of data.
  - As we learn more forms of data, the function "recipe" will stay mostly unchanged.
- Primitive data include data types like strings, numbers, and images.
- Non-primitive data are other types that are designed with the HtDD recipe, like distinct, interval, enumeration, and itemization.

Interval data definitions

- Interval data definitions are used for information that is numbers within a certain range.
- `[a, b]` mean inclusive, as in the interval contains a and b.
- `(a, b)` mean exclusive, as in the interval does not contain a nor b, but all the numbers between them.

```
; Example interval data definition for seat numbers in a row of a theater

;; SeatNum is Natural[1, 32]
;; interp. seat numbers in a row, 1 and 32 are aisle seats
(define SN1 1) ; ailse
(define SN2 12) ; middle
(define SN3 32) ; aisle
#;
(define (fn-for-seat-num sn)
  (... sn))
```

```
;; Template rules used:
;; atomic non-distinct: Natural[1, 32]
```

Enumeration data definitions

- Enumeration data definitions are used for information that consists of a fixed number of two or more distinct values.
- If the examples are redundant, say so with `<examples are redundant for enumerations>`.

```
; Example enumeration data definition for letter grades

;; LetterGrade is one of:
;;  - "A"
;;  - "B"
;;  - "C"
;; interp. the letter grade in a course
;; <examples are redundant for enumerations>
#;
(define (fn-for-letter-grade lg)
  (cond [(string=? lg "A") (...)]
        [(string=? lg "B") (...)]
        [(string=? lg "C") (...)]))

;; Template rules used:
;;  - one of: 3 cases
;;  - atomic distinict: "A"
;;  - atomic distinict: "B"
;;  - atomic distinict: "C"
```

Itemization data definitions

- Itemization data definitions are used for information comprised of a fixed number of two or more categories, at least one of which is not a distinct data value.
- For itemizations, `else` is able to be used for the last question in `cond`.
- Guards "guard" against other datatypes breaking when being evaluated in a conditional.

```
; A string would break this conditional
(= x 10)
; So we add a guard
(and (number? x) (= x 10))
```

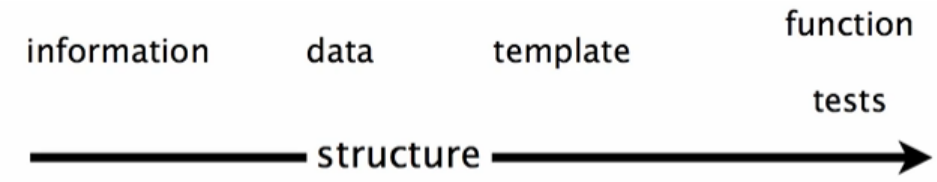
- If a given subclass is the last subclass of its type, we can reduce the test to just the guard.
- If all remaining subclasses are of the same types, then we can eliminate the guards.

```
; Example itemization data definition for a New Year's countdown

;; CountDown is one of:
```

```
;;  - false
;;  - Natural[1, 10]
;;  - "complete"
;; interp.
;;    false          means countdown has not yet started
;;    Natural[1, 10] means countdown is running and how many seconds are left
;;    "complete"     means countdown is over

(define CD1 false)
(define CD2 10) ; just started running
(define CD3 3)  ; almost over
(define CD4 "complete")
#;
(define (fn-for-countdown c)
  (cond [(false? c) (...)]
        [(number? c) (... c)]
        [else (...)]))

;; Template rules used:
;;  - one of: 3 cases
;;  - atomic distinct:     false
;;  - atomic non-distinct: Natural[1, 10]
;;  - atomic distinct:     "complete"
```

Flow of structural information

- Each step in the systematic design process builds off the previous steps.
- Identifying the structure of the information is a key step in program design.
- As data definitions get more sophisticated, one will see that choosing the structure to use is a point of leverage in designing the overall program.



▼ Lecture 3a: How to Design Worlds

12/26/23

The big-bang mechanism

- The big-bang mechanism allows one to make interactive programs in Dr. Racket.
- These programs can have a change in state or change the display based on keyboard and mouse input.
- The expression `big-bang` allows one to create big-bang mechanisms.

- `on-tick` allows one to pass a function that happens every tick.
- `to-draw` allows one to pass a function that happens on every draw call.

- This function is polymorphic, meaning it works for any kind of world state.

```
; Set the initial state X
(big bang <initial_world_state>
; Each time the clock ticks, call func1 with the
; current world state to get the next world state
; X -> X
                   (on-tick <func1>)
; Each time the clock ticks, call func2 with the
; current world state to draw the current world state
; X -> Image
                   (to-draw <func2>))
```

Domain analysis

- When designing a program, one should sketch a mock-up of how the program may work.
- One should identify constant and changing information in the program.

Programming through a main function

- Templates should be used to outline the basic structure of a program before the details are implemented.
  - They break the problem down into steps of the process, not just pieces of the solution.
- Traceability is the ability to see what path each analysis element takes in a resulting program.
- To build an actual program we:
  1. Define constants.
  2. Create data definitions.
     a. These correspond to the changing information in the program.
  3. Create functions.
- Programs always change, so they should be written so that they are as easy to change as possible.

Key handlers

- `on-key` can be used to detect key presses.

```
; Detect if a space bar is pressed, and run the handle-key function
(handle-key c " ")
```

White-box testing

- White-box testing refers to one knowing about the implementation of a function when testing.
- This is the opposite of black-box testing, where one does not need to know the implementation of a function when testing.

▼ Lecture 3b: Compound Data

12/27/23

Compound data

- Compound data refers to data that represents information that consists of two or more naturally connected values.

Defining “structs”

- `define-struct` allows one to define compound data in Racket.
- One can construct or “make” a compound data value using the `-make` prefix.
- A data attribute can be selected from a struct with the `-<data>` suffix.
- The predicate can be used to check if compound is constructed from a specific compound data type with the `?` suffix.

```
; Create a compound data type to describe an <x, y> position


; (define-struct <struct_name> (<data>...))
(define-struct pos (x y))


; Make a position value at <3, 6> and assign that value to P1
(define P1 (make-pos 3 6))


; Select "x" (3) from P1
(pos-x P1)
; Select "y" (6) from P1
(pos-y P1)


; true, is constructed from "pos"
(pos? P1)
; false, is NOT constructed from "pos"
(pos? "Hello"
```

Compound data definitions

- A compound data definition is similar to other data definitions.
  1. A structure definition is written.
  2. A type comment that defines the new compound type.
  3. An interpretation that describes the correspondence between information and data.
  4. One or more examples of that data.
  5. A template for a 1 argument function operating on data of this type.

a. The function should be commented describing what type of atomic data each selector is.

```
; Example data definition for a hockey player compound data type

(define-struct player (fn ln))
;; Player is (make-player String String)
;; interp. (make-player fn ln) is a hockey player with
;;    fn is first name
;;    ln is last name
(define P0 (make-player "Bobby" "Or"))
(define P1 (make-player "Wayne" "Gretzky"))
#;
(define (fn-for-player p)
  (... (player-fn p)    ; String
       (player-ln p))   ; String

;; Template rules used:
;;  - compound: 2 fields
```

- For compound data functions, one should vary all the elements of compound data in tests.
- If you find that a function is doing more than one task, it is best practice to split that function up into multiple functions that each perform one task each.

▼ Lecture 4a: Self-Reference

12/27/23

List mechanisms

- Arbitrary-size data is used for information whose size/quantity is unknown.
- The simplest form of arbitrary-size data is lists of values.
- An `empty` list can hold any type of value, or a mix of values.
  - Types of values will not be mixed often because data definitions do not allow that.
- `cons` can be used to append a value to the beginning of a list.

```
; A list with no values
empty
; A list with one value
(cons "Celtics" empty)
; A list with two values
(cons "Lakers" (cons "Celtics" empty))
```

- `first` produces the first element of a list.

```
; Define a list
(define L0 (cons "Lakers" (cons "Celtics" empty)))
```

```
; Produces "Lakers"
(first L0)
```

- `rest` produces all the elements after the first element of the list.

```
; Define a list
(define L0 (cons "Lakers" (cons "Celtics" empty)))

; Produces (cons "Celtics" empty)
(rest L0)
```

- `first` and `rest` can be chained to access any element in a list.

```
; Define a list
(define L0 (cons "Mavs" (cons "Lakers" (cons "Celtics" empty))))

; Access the second element of the list
; Produces "Lakers"
(first (rest L0))
```

- `empty?` is used to check if a list is empty.

```
; Define a list
(define L0 (cons "Lakers" (cons "Celtics" empty)))

; Empty, false
(empty? empty)
; Not empty, true
(empty? L0)
```

List data definition

- Data definition for lists are always self-referential, meaning they contain their own type in addition to `empty`.
- A well-formed self-referential data definition means a self-referential data definition with a non-self-referential case, or a base case.
  - The base case allows the recursion to stop eventually.
  - In this case, the base case is `empty`.
- Lists are a compound data type.
  - Because of this, they use the compound data predicate for the template function.
  - The `fn-for-list` predicate allows the list to be replaced with a new list omitting the first element.
    - We recursively check through a shortening list in order to check each element of the original list.
  - The `empty?` predicate is used in the template function for any `empty` values.
- List data definitions should contain examples of both the base and self-referential case.

- The `self-reference` template rule is used for lists.

```
; Example of a list data definition

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

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

Positions in list templates

- In the example above:
  - The red highlighted section is where the base case goes.
  - The blue highlighted section is where the contribution from the first element in the list to the overall result goes.
  - The green highlighted section is how the base and recursive case combine.
    - In other words, this is the way in which the final function combines the contribution of the first with the result of the natural recursion.

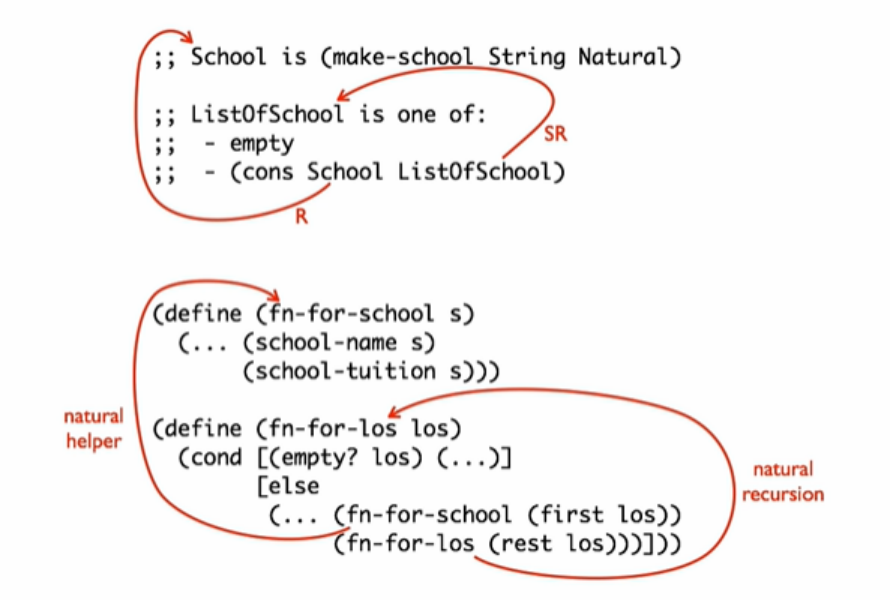
▼ Lecture 4b: Reference

12/27/23

The reference rule

- A reference relationship refers to when one data definition references another data definition.
- In the template, a natural helper is used to relate the two data definitions.
  - A natural help in the template informs the programmer that the functionality should be designated to the helper function that consumes the referred type.
- In the template function, there will be a helper function call to relate the two functions.

```
; Example of referential and self-referential data definition

(define-struct school (name tuition))
;; School is (make-school String Natural)
;; interp. name is the school's name, tuition is tuition in USD
(define S1 (make-school "School1" 27707))
(define S2 (make-school "School2" 23300))
(define S3 (make-school "School3" 28500))
#;
(define (fn-for-school s)
  (... (school-name s)      ; String
       (school-tuition s))) ; Natural

;; Template rules used:
;;  - compound: 2 fields


;; ListOfSchool is one of:
;;  - empty
;;  - (cons School ListOfSchool) ; Reference
;; interp. a list of schools
(define LOS1 empty)
(define LOS2 (cons S1 (cons S2 (cons S3 empty))))
#;
(define (fn-for-los los)
  (cond [(empty? los)
         (...)]
        [else
         (... (fn-for-school (first los)) ; Natural helper
              (fn-for-los (rest los)))]))

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




```
; Functions

;; ListOfSchool -> Image
;; produce bar chart showing names and tuitions of the consumed schools
(check-expect (chart empty)
              (square 0 "solid" "white"))
(check-expect (chart (cons (make-school "S1" 8000)
                           empty))
              (beside/align "bottom" (overlay/align "center" "bottom" (rotate 90 (t
                                                    (rectangle BAR-WIDTH (* 8000
                                                    (rectangle BAR-WIDTH (* 8000
                            (square 0 "solid" "white")))
(check-expect (chart (cons (make-school "S2" 12000)
                           (cons (make-school "S1" 8000)
                                 empty)))
              (beside/align "bottom" (overlay/align "center" "bottom"  (rotate 90 (
                                                    (rectangle BAR-WIDTH (* 1200
                                                    (rectangle BAR-WIDTH (* 1200
                                    (overlay/align "center" "bottom" (rotate 90 (t
                                                    (rectangle BAR-WIDTH (* 8000
                                                    (rectangle BAR-WIDTH (* 8000
                                    (square 0 "solid" "white")))


; (define (chart los) (square 0 "solid" "white")) ; stub

; <use template from ListOfSchool>

(define (chart los)
```

```
  (cond [(empty? los)
         (square 0 "solid" "white")]
        [else
         (beside/align "bottom"
                       (make-bar (first los))
                       (chart (rest los)))]))

; Helper function
;; School -> Image
;; produce the bar for a single school in the bar chart
(check-expect (make-bar (make-school "S1" 8000))
              (overlay/align "center" "bottom" (rotate 90 (text "S1" FONT-SIZE FONT
                                                                (rectangle BAR-WIDTH (* 8000
                                                                (rectangle BAR-WIDTH (* 8000

; (define (make-bar s) (square 0 "solid" "white")) ; stub

; <use template from School>

(define (make-bar s)
  (overlay/align "center" "bottom" (rotate 90 (text (school-name s) FONT-SIZE FONT-
                                                    (rectangle BAR-WIDTH (* (school-tuit
                                                    (rectangle BAR-WIDTH (* (school-tuit
```

- Doing examples force us to figure out what we really want.
- Doing the examples before designing a function can:
  - Help figure out what a function needs to do.
  - Help us remember the names of primitives a function needs.
  - Help us get details like the order of arguments in a function correct.
- It is best practice to, if necessary, copy parts from the examples to the function, and not the other way around.

▼ Lecture 5a: Naturals

12/27/23

Natural numbers

- There are an arbitrary amount of natural numbers, so we can use a well-formed self-referential data definition to describe the type `Natural`.
- `add1` is a primitive in Racket that can be used to “add 1” to any number.
- `sub1` is a primitive in Racket that can be used to “subtract 1” to any number.

```
; Add 1, produce 2
(add1 1)
```

```
; Subtract 1, produce 4
(sub1 5)
```

- Natural numbers can be defined as one of:
  - `0`
  - `(add1 Natural)`

```
; Self-referential data definition for natural numbers

;; Natural is one of:
;;  - 0
;;  - (add1 Natural)
;; interp. a natural number#;
#;
(define (fn-for-natural n)
  (cond [(zero? n)(...)]
        [else
         (... n
              (fn-for-natural (sub1 n)))]))
```

- Data definitions are powerful because we can use them to represent data with any information we want.\

▼ Lecture 5b: Helpers

12/28/23

Introduction

- Large design problems need to be broken into smaller pieces in order to be tractable.

Function composition

- A function should be split into a function composition when it performs two or more distinct operations on the consumed data.
- Function composition should be used when a function must perform two or more distinct and complete operations on the consumed data.
  - In this case, we discard the entire template and replace it with a function composition.
    - The template will be used for the sub-functions that make up the function composition.

```
; Example function composition for laying out images in
; increasing order of size


; 1. First sort the images with "sort-images"
; 2. Next layout the images with "layout-images"
```

```
(define (arrange-images loi)
  (layout-images (sort-images loi))
```

- Making a stub produce its arguments will make the stub more useful in certain cases.
- For a function composition, one does not need to test the base case.
  - This is because the functions composing the function composition test the base case for it through their tests.

Operating on a list

- One needs a helper function to operate on a list.
- One needs a recursive function to operate on arbitrary sized data.

```
; Example of "sort-images" where the list that needs to be sorted is
; of an arbitrary size. Therefore, the helper function "insert" is
; needed to insert (first loi) into the correct place in (rest loi)


;; ListOfImage -> ListOfImage
;; sort images in increasing order of size (area)

(define (sort-images loi)
  (cond [(empty? loi)
         empty]
        [else
         (insert (first loi)
              (sort-images (rest loi)))]))
```

- Assumptions can be written to communicate functionality of a higher-level function to a lower-level function.

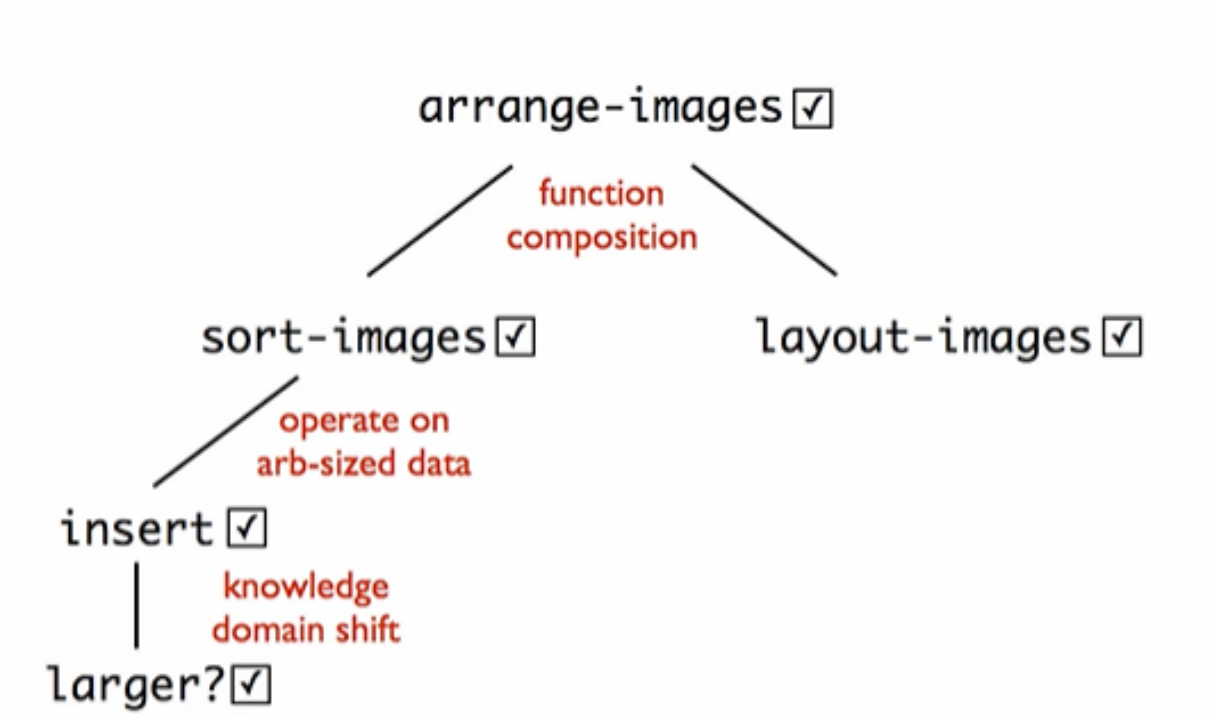
```
; Example assumptions saying that the ListOfImage passed as an argument
; is already sorted by the higher-level function calling "insert"


;; Image ListOfImage -> ListOfImage
;; insert img in proper place in list (in increasing order of size)
;; ASSUME: loi is already sorted

(define (insert img loi) loi) ; stub
```

Domain knowledge shift

- When we shift our knowledge domain, we should use a helper function.
  - A domain knowledge shift refers to when the information we are processing changes considerably.

```
; Example of a knowledge domain shift
; We are shifting from knowledge about sorting lists ->
; Knowledge about finding which image area is larger, denoted by
```

```
; the "larger?" function.


;; Image ListOfImage -> ListOfImage
;; insert img in proper place in list (in increasing order of size)

(define (insert img loi)
  (cond [(empty? loi)
         (cons img empty)]
        [else
         (if (larger? img (first loi))
             (cons (first loi) (insert img (rest loi)))
             (cons img loi))]))
```



▼ Lecture 6: Binary Search Trees

12/28/23

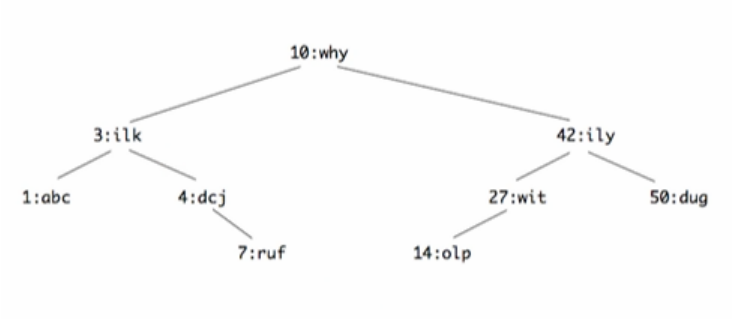
List abbreviations

- Racket can abbreviate lists by switching to the *"Beginning Student with List Abbreviations"* language.
- To add one element to a list, use `cons`.
- `list` should be reserved for creating a fully formed list all at once.
- `append` is used to combine two lists.

```
; Default notation
(cons "a" (cons "b" (cons "c")))

; Abbreviated notation
(list "a" "b" "c")

; Add a single element to a list
(cons "x" (list "a" "b" "c"))

; Append two lists
(append L1 L2)
```

Binary search trees

- Binary search trees (BSTs) split a list of a tree at the middle element along each level of a sorted list.
- An invariant, a rule that doesn't change, across the tree is that.
  - All accounts in a left sub-tree are lesser than the parent node.
  - All accounts in the right subtree are greater than the parent node.
  - The same key never appears twice in the tree.



- The time complexity of a BST is $O(log(n))$.

BST data definition

- Each node in a BST has four parts:
  - left sub-tree
  - right sub-tree
  - key
  - value
- BST left and right sub-trees are an arbitrary size.
- Theis means that BSTs are a compound data definition with two self-referential cycles.
- Invariants should be explicitly stated in data definitions.

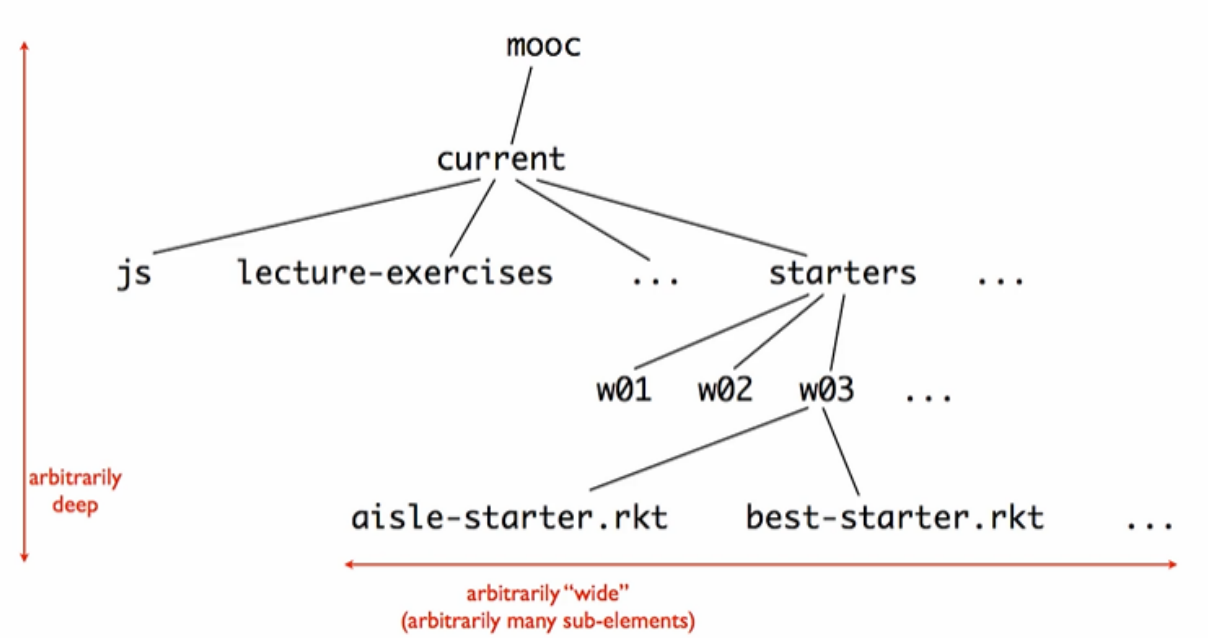
```
; Example of a BST data definition


(define-struct node (key val l r))
;; BST (Binary Search Tree) is one of:
;;  - false
;;  - (make-node Integer String BST BST)
;; interp. false means no BST, or empty BST
;;         key is the node key
;;         val in the node val
;;         l and r are left and right subtrees
;; INVARIANT: for a given node:
;;    key is > all keys in its l(eft) subtree
;;    key is < all keys in its r(ight) subtree
;;    the same key never appears twice in the tree
#;
(define (fn-for-bst t)
  (cond [(false? t)
         (...)]
        [else
         (... (node-key t)               ; Integer
              (node-val t)               ; String
              (fn-for-bst (node-l t))    ; BST
              (fn-for-bst (node-r t)))])) ; BST

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: false
;;  - compound: (make-node Integer String BST BST)
;;  - self-reference: (node-l t) has type BST
;;  - self-reference: (node-r t) has type BST
```

▼ Lecture 7: Mutual Reference

12/29/23

Mutually recursive data

- An arbitrary arity tree is characterized by having two dimensions and requiring two cycles in the type reference graph/comments.
- These trees are arbitrarily wide and deep in two dimensions.
  - Arbitrary in computer science means of an unknown size.
- Each element can have sub-elements or data.



- A self-reference in the list allows the list to be arbitrary in length.
  - This allows for an arbitrary amount of sub-directories.
- The `ListOfElement` has a reference to the `Element` type.
- Each `Element` is a compound type with a reference to the `ListOfElement` .
  - This allows the sub-directories to be arbitrarily deep.
- These references together form a mutual reference.
  - In other words, for this data definition, the self reference allows arbitrary width, and the mutual reference allows arbitrary depth.

```
; Example of arbitrary arity tree data definitions
; The ListOfElement contains a self-reference
; The two data definitions contain a mutual reference


(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
```

Templating mutual recursion

- Remember that the reference rule says to wrap selectors that produce non-primitive types in the appropriate function.

```
; Example of arbitrary arity tree templates
; fn-for-loe conatins a self-reference
; both templates contain a mutual reference


(define fn-for-element e)
  (... (elt-name e))              ; String
       (elt-data e)               ; Integer
       (fn-for-loe (elt-subs e))) ; ListOfElement

(define (fn-for-loe loe)
  (cond [(empty? loe)
         (...)]
        [else
         (... (fn-for-element (first loe)) ; Element
              (fn-for-loe (rest loe)))]))  ; ListOfElement
```

Function on mutually recursive data

- Especially for mutually recursive data, trust the natural recursion from functions.
- A template with the right structure is used, and the recursion always terminates in a base case.
- `or` can be used in function signature if a function can produce multiple types.
  - This is most similar to an exception in other programming languages.

```
; Example of arbitrary arity tree functions
; These functions sum all of that data (integers) in the tree


(define (sum-data--element e)
  (if (zero? (elt-data e))
      (sum-data--loe (elt-subs e))
      (elt-data e)))

(define (sum-data--loe loe)
  (cond [(empty? loe)
         0]
        [else
         (+ (sum-data--element (first loe))
            (sum-data--loe (rest loe)))]))
```

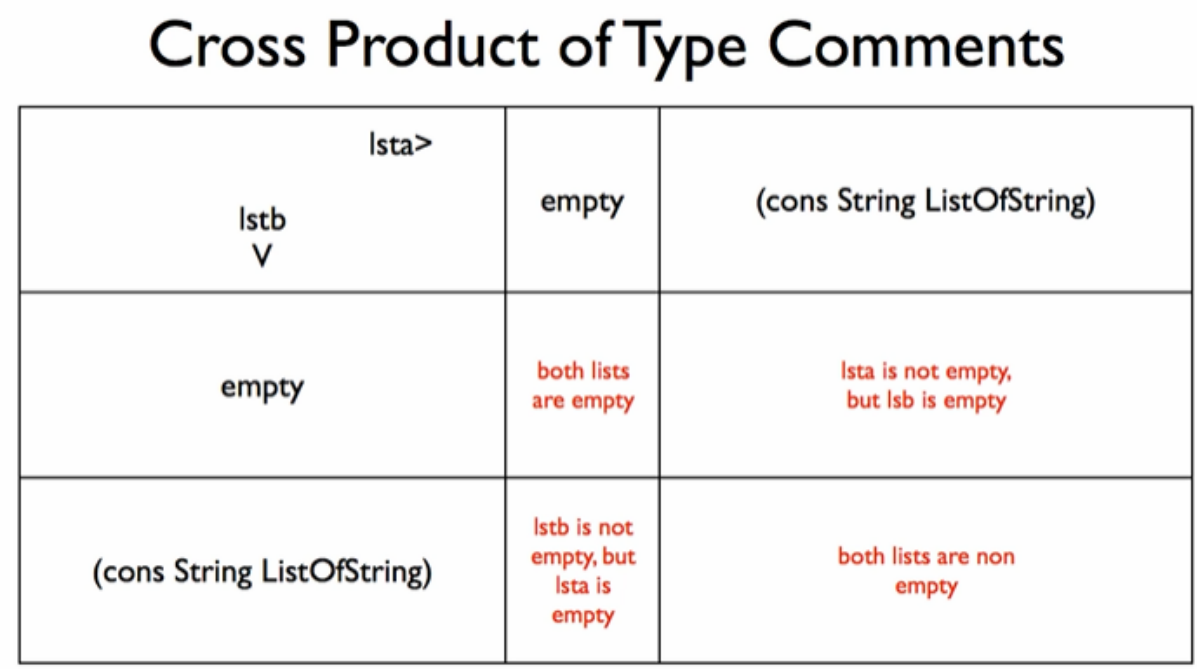
▼ Lecture 8a: Two One-of Types

12/29/23

Cross product table

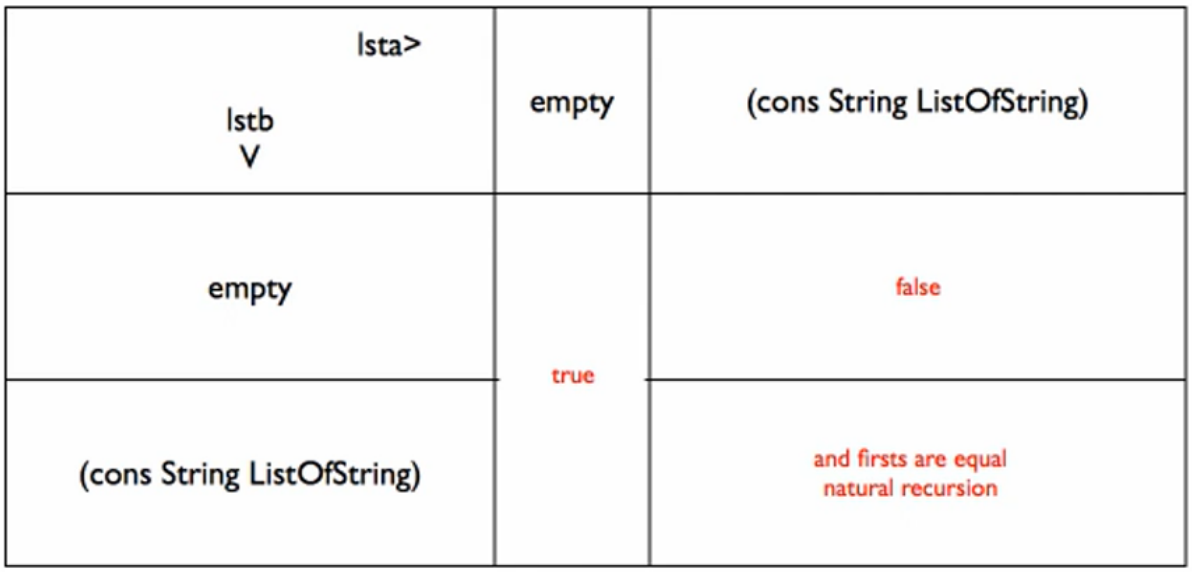
- Type comments can be used to reason about the behavior of a function.
- A model in a less detailed, more abstract, often non-code, representation of a function.
- A cross product is a model used when one has a function that consumes two one-of data types definitions.
  - The two arguments do not necessarily have to be identical, but they must be one-of.
- Making a cross-product table can be helpful to see what combination of they data types a function can consume.
  - This is helpful for knowing what examples/tests to write.

```
; Example list of string data definition
; In this case, a function will need at least 4 tests.


; Data definition:

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

; Function:

(define (prefix? lsta lstb) false) ; stub
```



- This table can be simplified based on the examples/tests.

```
; Example tests for the function to find if lsta is a "prefix" of lstb


;; ListOfString ListOfString -> Boolean
;; produce true if lsta is a prefix of lstb
; top left
(check-expect (prefix=? empty empty) true)
; top right
(check-expect (prefix=? (list "x") empty) false)
; bottom left
(check-expect (prefix=? empty (list "x")) true)
; bottom right
(check-expect (prefix=? (list "x") (list "x")) true)
(check-expect (prefix=? (list "x") (list "y")) false)
(check-expect (prefix=? (list "x" "y") (list "x" "y")) true)
...
```



- This simplified table is a model that can be used to simplify the function.\

▼ Lecture 8b: Local

12/29/23

Forming and intuition

- One thing that separates good programmers from great programmers is taking the time to improve the structure of their code once it is written.
- Racket provides local expressions, which make it possible to have definitions visible only within the local expression.

- Local definitions exist only inside the local expression.
- This is opposed to top-level/global definitions.

```
; Example local expression, produces 3
; "a" and "b" are only valid inside the local


; Local expression anatomy
#;
(local [<definitions>...]
  <expression>)


; definition
(local [(define a 1)
        (define b 2)]
; body
  (+ a b))
```

- Local expressions can also be used to define functions.

```
; Example local expression when the function "fetch" is defined


(local [(define p "accio ")
                ; locally defined function
        (define (fetch n) (string-append p n))]

  (fetch "portkey"))
```

Lexical scoping

- The concept of lexical scoping allows one to answer questions about what definition any reference to an expression refers to.
- The top-level/global scope refers to the scope of the whole program.
- Local scopes override top-level scopes.
  - Expressions values reference the inner-most scope that defines them

```
; Example of a local expression overriding a top-level expression
; In this case, "accio portkey" is printed, as the local expression
; overrides the global expression.


(define p "incendio ")

(local [(define p "accio ")
```

```
        (define (fetch n) (string-append p n))]

  (fetch "portkey"))
```

Evaluation rules

- Evaluation of local expressions is broke into three steps:
  1. Rename the local to a unique name in the program.
  2. Lift the local to the top-level scope.
  3. Replace the local with the body of itself

```
; Example evaluation of a local expression


; Starting local
(local [(define b 2)]
  (* b b)

; Rename local to a unique name
(local [(define b_0 2)]
  (* b_0 b_0)

; Lift the local to the top-level scope
(define b_0 2)
(local []
  (* b_0 b_0)

; Replace the local with the body of itself
(define b_0 2)
(* b_0 b_0)

; Evaluate normally
(* 2 2)
4
```

Encapsulation

- Encapsulation means bunding data together into a single unit to simplify a program.
- In Racket, this can be done by wrapping multiple forms of data in a local expression.

```
; Example of two functions encapsulated into one function


(define (sum-data e)
  (local [(define (sum-data--element e)
            (if (zero? (elt-data e))
```

```
                (sum-data--loe (elt-subs e))
                (elt-data e)))

          (define (sum-data--loe loe)
            (cond [(empty? loe) 0]
                  [else
                   (+ (sum-data--element (first loe))
                      (sum-data--loe (rest loe)))]))]
    (sum-data--element e)))
```

- Some good candidate functions for encapsulation are functions that has one or more helper functions closely linked to it.
- Refactoring means changing a program's code/structure without changing the program's behavior.
  - Encapsulation is an example of refactoring.
- Encapsulation is vital when multiple programmers work on the same large project, as it allows them to choose whatever names for data or functions they want locally.

Avoiding re-computation

- Local expressions can be used to avoid recomputing results.
  - This can be used to improve program efficiency, especially for recursive algorithms.
- They do this by pre-computing a result, preventing the need to compute it multiple times in a program.

▼ Lecture 9: Abstraction

12/30/23

Abstraction

- Abstraction is a technique for managing the complexity of a program by reducing redundancy.
- Paradoxically, abstract functions should be developed in the opposite order, meaning that you should do:
  1. Function
  2. Examples/tests
  3. Purpose
  4. Signature
- To abstract two highly repetitive functions:
  1. Copy one of the functions.
  2. Give the function a more general name
  3. Add and use a parameter at the varying position.
- Then replace the specific expressions with calls to the abstract function using the varying value.

```
; Example of a function that maps a function onto a list of numbers
; "fn" is the varying parameter
```

```
(define (mapping fn lon)
  (cond [(empty? lon) empty]
        [else
         (cons (fn (first lon))
               (map2 fn (rest lon)))]))


; Now "mapping" can be used for any mathematical operation


(define (squares lon) (mapping sqr lon))
(define (square-root lon) (mapping sqrt lon))
```

- The above is a higher-order function, meaning is can consume other functions as arguments.
- Examples/tests can also be abstracted from the specific functions.
- In the signature of a higher order function, the type of a function parameter is that function's signature wrapped in parentheses
  - Because a function can operate on any datatype, a type parameter is used instead.
  - A type parameter is typically any uppercase letter, such as X or T.

```
; Example of a function abstraction that maps a list of types
; according to a function argument

; This function has to consume a list of a type and produces
; a list of a type, therefore instead of just
; writing "Function" for the signature, we write (X -> Y) instead

; Because the type is unknown, "X", "Y", or any other uppercase
; letter is used in its place


;; (X-> Y) (listof X) -> (listof Y)
;; given fn and (list x0 x1 ...) produce (list (fn y0) (fn y1) ...)
(mapping fn lon)
...
```

Built-in abstract functions

- Some abstract functions are so useful, that they are built into Racket and other languages.
- Two of these built in abstract functions are `map` and `filter`.
  - `map` maps a function to a list.
  - `filter` filters a list based on a predicate.
  - `build-list` is used to build a list of the first n natural numbers mapped to a function.

        - `identity` produces what a function consumes.
    - `foldr` applies a procedure to the elements of lists.

12/31/23

Closures

- Sometimes the function passed to an abstract function doesn't exist yet.
- In those cases, one should define it with a local expression.

```
; Example where the "wide?" fn does not already exist,
; so instead it is locally defined inside the abstract fn


(define (wide-only loi)
  (local [(define (wide? i)
            (> (image-width i)
               (image-height i)))]
  (filter wide? loi)))
```

- This is necessary to do when the body of a function you want to pass to an abstract function refers to a parameter of the enclosing abstract function.
- The terminology for this locally defined function is a closure.

```
; The parameter "w" is a parameter of the abstract function
; "wider-than-only" , and is used in the fn "wider-than?"
; Therefore, "wider-than?" cannot be defined outside the abstract
; function, and must be defined locally with a local expression

; "wider-than?" is a closure, which "closes over" the value of w


(define (wider-than-only w loi)
  (local [(define (wider-than? i)
            (> (image-width i) w))]
  (filter wider-than? loi)))
```

Fold functions

- Abstract functions can be produced directly from templates.
    - This can be useful for types involving mutual reference.
- A fold function is an abstract function based directly on the template function.
    - An additional parameter is added to the fold function for each set of `...`.

```
; Function template
```

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


; "fn" and "b" are added as parameters to the fold function


(define (fold fn b lox)
  (cond [(empty? lox)
         b]
        [else
         (fn (first lox)
             (fold fn b (rest lox)))]))
```

▼ Lecture 10a: Generative Recursion

01/01/24

Generative recursion

- Structural recursion is recursion that terminate in a base case.
  - The data passed into a recursive call is a sub-section of the data being passed to the current call.
- Generative recursion is similar to other types of recursion, expect in the fact one needs to develop a proof for the function being written.
  - The data passed into a recursive call is generated, rather than being a part of the data passed to the current call.
- The HtDF generative recursion recipe differs from the standard HtDF recipe.
- There is a standard template for generative recursion functions.

```
; This is the standard template for
; generative recursion, in Racket syntax

; The "trivial?" predicate refers to the point at which recursion
; is halted, similar to the base case in structural recursion

; The "next-problem" refers to the recursive case that works in
; the direction of the trivial answer


(define (genrec-fn d)
  (cond [(trivial? d) (trivial-answer d)]
```

```
        [else
         (... d
              (genrec-fn (next-problem d)))]))
```

- With generative recursion we can no longer rely on type comments and template rules to guarantee that the recursion will end.
- This is because they are not operating based on a well-formed self-referential data definition.
- Generative recursion has a three-part termination argument that includes a(n):
  - Base case
  - Reduction step
  - Argument that repeat application of the reduction step will eventually reach the base case.
- Arguments are not necessarily simple or even provable (yet).

▼ Lecture 10b: Search

01/01/24

Lambda expressions

- A locally defined function that is only used in one place can be refactored with a lambda expression.
- The keyword `lambda` signals to create an anonymous function.
  - An anonymous function is an inline function without a name.
- The lambda function first takes one or more element(s) to operate on and defines what is done to it/them.

```
; The top function can be simplified to the
; bottom function using a lamba expression

; Anatomy of a lambda function
; (lambda <args>... <expression>)


; local expression implementation
(define (only-bigger threshold lon)
  (local [(define (pred n)
            (> n threshold))]
    (filter pred lon)))


; lambra expression implementation
(define (only-bigger threshold lon)
    (filter (lambda (n) (> n threshold))
            lon))
```

Search spaces

- For some problems, one can generate a search space of all possible solutions, and then pick then search through that space to pick the solution that is most appropriate.
- We generate an arbitrary arity tree of the search space, and then do back-track search over it to find one or more solutions.
  - We “walk down” the tree, and back track if a dead-end is reached.
- `list-ref` is used to index an element from a list based on the element’s position in that list.

```
; Index the third element of L


(define L (list 3 6 0 -3))
; Produces 0
(list-ref L 2)
```

Template blending

- One can blend multiple function templates using a technique called template blending.
- For example, if one needs to design a function that generates an arbitrary-arity tree and does a backtracking search over it, one needs to combine the templates for:
  - Generative recursion
  - Arbitrary-arity trees
  - Backtracking search

▼ Lecture 11: Accumulators

01/02/24

Accumulators

- Structural recursion does not have the ability to keep track of when a function has been in traversal or how much work is remaining to be done.
- Accumulators solve these problems in three different ways.

Context preserving

- Context preserving accumulators preserve context that would otherwise be lost in structural recursion.
- It is best practice to make a signature, an invariant, and one or more examples for the accumulator.

```
; Example function using an accumulator "acc" to
; keep track of the current list index
; The current list index in this case is the "context"
; that is being "preserved"


;; (listof X) -> (listof X)
;; produce list consisting of only the 1st, 3rd, 5th... elements of lox
```

```
(define (skip1 lox)
  ;; acc: Natural
  ;; 1-based position of (first lox) in lox
  ;; (skip1 (list "a" "b" "c") 1)
  ;; (skip1 (list     "b" "c") 2)
  ;; (skip1 (list         "c") 3)
  (local [(define (fn-for-lox lox acc)
            (cond [(empty? lox)
                   empty]
                  [else
                   (if (odd? acc)
                       (cons (first lox)
                             (fn-for-lox (rest lox) (add1 acc)))
                       (fn-for-lox (rest lox) (add1 acc)))]))]
    (fn-for-lox lox 1)))
```

Tail recursion

- Pending computations generated by recursion are stored in the stack.
  - This can be very expensive to the computer memory.
- The result of an expression in the tail position of a function will be the result of the function.
  - Essentially, it is the position of the return value.

```
;; The (+ ... ...) primitive call is in tail position

;; The recursive call (sum (rest lon)) that is NOT in tail
;; position is what causes the build-up on the stack


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

- A recursive call that is in the tail position is what tail recursion is.
  - You remove the operator wrapping the recursive call, and instead move that operator to the accumulator inside the recursive call.

```
; The accumulator moves the recursive call to the tail position
; by putting the addition inside the recursive call rather than outside


;; (listof Number) -> Number
;; produce sum of all elements of lon
```

```
(define (sum lon0)
  ;; acc: Natural
  ;; the sum of the elements of lon0 seen so far
  ;; (sum (list 2 4 5)  0)
  ;; (sum (list 2 4 5)  0)
  ;; (sum (list   4 5)  2)
  ;; (sum (list     5)  6)
  ;; (sum (list      ) 11)
  (local [(define (sum lon acc)
            (cond [(empty? lon)
                   acc]
                  [else
                   (sum (rest lon) (+ acc (first lon)))]))]
    (sum lon0 0)))
```

Worklist accumulators

- When adding an accumulator to mutually recursive functions, it needs to be added to all the functions.
- For tail recursion in mutually recursive functions, each mutually recursive call must be in tail position.
- A worklist accumulator allows for tail recursion of an arbitrary arity tree.
  - Each child node calls its siblings, cutting down on back tracking substantially.
  - The accumulator keeps track of all the nodes we've seen but haven't visited.

```
; Example of a worklist accumulator
; The worklist accumualtor "todo" keeps track of all the
; wizards we have seen, but have not visited in the search yet

; appendings (wiz-kids) to the front of "todo"
; allows for a depth-first search

; In an alternate scenerio, append "todo" to the front of (wiz-kids)
; allows for a breadth-first search


;; Wizard -> Natural
;; produces the total number of wizards in the consumed tree
;; (including the root)
(define (count w)
  ;; rsf is Natural; the number of wizards seen so far
  ;;
  ;; todo is (listof Wizard); the wizrds we have
  ;; seen but not visited yet
  (local [(define (fn-for-wiz w todo rsf)
            (fn-for-low (append (wiz-kids w) todo) (add1 rsf)))
          (define (fn-for-low todo rsf)
            (cond [(empty? todo)
```

```
                    rsf]
                   [else
                    (fn-for-wiz (first todo) (rest todo) rsf)]))]
    (fn-for-wiz w empty 0)))
```

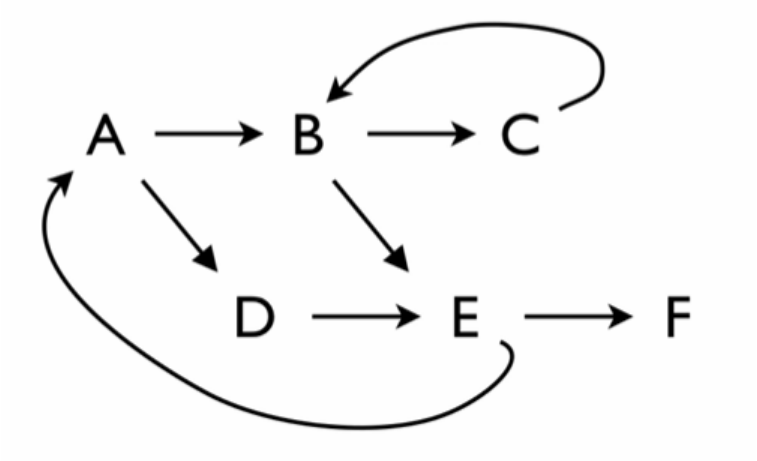
- Seeing code in terms of a blended templates makes it easier to understand and write.

▼ Lecture 12: Graphs

01/02/24

Graphs

- Graphs are different from trees in the fact that they can:
  - Have cycles between two or more nodes.
    - In the figure below, B leads to C lead to B.
    - A graph with cycles is cyclic, and one without cycles is acyclic.
  - Have multiple nodes leading into a single node.
    - In the figure below, both B and D lead into E.



- A directed graph is a graph with arrows that only go in one direction.

Constructing cyclic data

- In Racket, the `shared` keyword allows one to define data in a circular structure.
  - This allows `struct`s to contain each other.

```
;; "-0-" is the name of the circular structure.
;; "A" contains "B" contains "C" contains "A", starting at "A"


(define H0
  (shared ((-A- (make-room "A" (list -B-)))
```

```
           (-B- (make-room "B" (list -C-)))
           (-C- (make-room "C" (list -A-))))
    -A-))
```

- Cyclical structure doesn't show up in text well.
  - This is a property of cyclical structure, not the programming language's representation of it.
  - Racket uses the above format to make forming cyclic graphs easier.
    - Then that is turned into a less readable structure to humans, but a more readable structure for computers.

Templating cyclic data

- Previous templates can be blended together to create a template for cyclic data

```
; Template for navigating a graph


;; templates:
;;  - structural recursion
;;  - encapsulate with local
;;  - tail-recursive w/ worklist
;;  - context-preserving accumulator for what rooms
;;    have already been visited
;;  - details

(define (fn-for-house r0)
  ;; todo: (listof Room); a worklist accumulator
  ;;
  ;; visited: (listof String); context preserving accumulator,
  ;; names of rooms already visited
  ;; ASSUME: all room names are unique
  (local [(define (fn-for-room r todo visited)
            (if (member (room-name r) visited)
                (fn-for-lor todo
                            visited)
                (fn-for-lor (append (room-exits r) todo)
                            (cons (room-name r) visited))))

          (define (fn-for-lor todo visited)
            (cond [(empty? todo)
                   (...)]
                  [else
                   (fn-for-room (first todo)
                                (rest todo)
                                visited)]))]

    (fn-for-room r0 empty empty)))
```

Closing thoughts

- Systematic design uses higher-level design concepts like type comments, templates, and template blending to break down complicated problems to a point where one can solve them simply.