CS 115
Lecture Notes
Winter 2019

Collin Roberts

January 8, 2019

Contents

1 Lecture 01

Outline

1. Administrivia
2. Introduction to CS 115 - Course Website and Slides 1-9
3. Introduction to DrRacket - Slides 10-20

1.1 Administrivia

1. Labs start **this week**.
2. I will lecture on the blackboard using the course slides as a resource. I will announce pre-reading for each future lecture.
3. Register your iClicker **exactly once**. The course staff will contact you at the end of January if your iClicker is not yet correctly registered.

1.2 Introduction to CS 115 - Course Website and Slides 1-9

1. Refer to the slides.
2. **CQ 1**

1.3 Introduction to DrRacket - Slides 10-20

Programming language design

- Imperative:
  - frequent changes to data
  - examples: machine language, Java, C++
- Functional:
  - examples: Excel formulas, LISP, ML, Haskell, Mathematica, XSLT, R (used in STAT 231)
  - more closely connected to mathematics
  - easier to design and reason about programs

We use **DrRacket**, a functional program, in CS 115. DrRacket is great for teaching, although I have yet to see a real computer system developed in it.

DrRacket is **Turing Complete**, so in theory any computer system could be developed using it. DrRacket provides an easy entry point into coding.
You will work in the imperative language, Python, in CS 116. Real computer systems are written in Python.

Themes of the course

- design (the art of creation)
- abstraction (finding commonality, not worrying about details)
- refinement (revisiting and improving initial ideas)
- syntax (how to say it), expressiveness (how easy it is to say and understand), and semantics (the meaning of what's being said)
- communication (in general)

Functions: A mathematical **function** definition consists of

- the **name** of the function,
- its **parameters** (aka **arguments**) (what the function **consumes**), and
- a mathematical **expression** using the parameters, to define what the function **produces**. The mathematical expression is **evaluated** by **substitution**.

Functions in DrRacket:

- As in the slides, we start with $f(x) = x^2$ and $g(x, y) = x - y$.
- Include exactly one set of parentheses for each function call.
- Write the function first inside the open parenthesis, followed by the arguments.
- Observe that following this syntax removes any ambiguity about the **order of operations**.
- Using this setup, try some examples of your own in DrRacket.

2 Lecture 02

Outline

1. Administrivia
2. Mathematical Functions in DrRacket
3. The DrRacket Environment
4. Defining New Functions in DrRacket

2.1 Administrivia

1. Email Barb Daly (contact information in Personnel page) with your

issues with iClicker registration or Markus setup.

2.2 Mathematical Functions in DrRacket

Familiar Math Notation	DrRacket
`3 - 2`	`(- 3 2)`
`3 - 2 + 4/5`	`(+ (- 3 2) (/ 4 5))`
`(6 - 4)(3 + 2)`	`(* (- 6 4) (+ 3 2))`

- **CQ 2**
- **CQ 3**

2.3 The DrRacket Environment

Built-in Functions - Examples
Integer division (in the positive integers to keep things simple for a start) produces a **quotient** and **remainder**, for example

- Dividing 17 by 5 produces the quotient 3 and the remainder 2 because $17 = 3 \cdot 5 + 2$.
- Dividing 15 by 5 produces the quotient 3 and the remainder 0 because $15 = 3 \cdot 5 + 0$.
- If dividing a by b produces the quotient q and the remainder r, so that $a = q \cdot b + r$, then requiring $0 \leq r < b$ makes the choice of q and r **unique**, so that `quotient` and `remainder` are actually functions.
- `(quotient 75 7)`
- `(remainder 75 7)`

Bad Syntax / Semantics - Examples

- `(* (5) 3)`
- `(+ (* 2 4)`
- `(5 * 14)`
- `(* + 3 5 2)`
- `(/ 25 0)`
- **CQ 4**

2.4 Defining New Functions in DrRacket

- DrRacket enforces correct syntax at all times.
- If you enter an expression which is not syntactically correct, then you will get an error message.

- Every syntactically correct DrRacket function call has the form `(functionname exp1 . . . expk)`, for some function `functionname` and expressions `exp1` … `expk`.

Familiar Math Notation	DrRacket
$f(x) = x^2$	`(define (f x) (* x x))`
$g(x, y) = x - y$	`(define (g x y) (- x y))`

- `define` is a special form; it looks like a DrRacket function, but not all of its arguments are evaluated.
- It binds a name to an expression (which uses the parameters that follow the name).

Identifiers

- To give names to the function and parameters, we use identifiers.
- Syntax rule: an **identifier** starts with a letter, and can include letters, numbers, hyphens, underscores, and a few other punctuation marks.
- It cannot contain spaces or any of ( ) , { } [ ] ‘ ’ “ ”.
- Syntax rule: **function definition** is of the form `(define (id1 . . . idk) exp)`, where `exp` is an expression and each `id` is an identifier.
- **CQ 5**

DrRacket Definitions Window

- can accumulate definitions and expressions
- Run button loads contents into Interactions window
- can save and restore Definitions window
- provides a Stepper to let one evaluate expressions step-by-step
- features include: error highlighting, subexpression highlighting, syntax checking, bracket matching

DrRacket Constants

Familiar Math Notation	DrRacket
$k = 3$	`(define k 3)`
$p = k^2$	`(define p (* k k))`

DrRacket Programs A DrRacket program is a sequence of definitions and expressions.

- The definitions are of functions and constants.
- The expressions typically involve both user-defined and built-in functions and constants.

Programs may also make use of special forms (which look like functions, but don't necessarily evaluate all their arguments).

- **CQ 6**

3 Lecture 03

Outline

1. Administrivia
2. Identifiers and Binding
3. Semantics
4. Values and Substitution

3.1 Administrivia

Anything?

3.2 Identifiers and Binding

1. Two functions can use the same parameter name (e.g. our definitions of f and g from last lecture both used x, with no conflicts).
2. A constant and a parameter can share the same name, for example

   ```
   (define x 3)
   (define (f x) (* x x))
   ```

   but this is not very readable and hence is not recommended.
3. Two constants cannot share the same name, for example

   ```
   (define x 4)
   (define x 5)
   ```

4. A constant and a function cannot share the same name, for example

   ```
   (define x 4)
   (define (x y) (* 5 y))
   ```

5. The name of a function can be the same as a parameter name of another function, for example

   ```
   (define (x y) (* 5 y))
   (define (z x) (* 3 x))
   ```

but this is not very readable and hence is not recommended.

6. The name of a function can be the same as the name of one of its parameters, for example

   ```
   (define (x x) (+ 1 x))
   ```

   but this is not very readable and hence is not recommended.

Remarks:

1. **define** binds an identifier to
   - (a) an expression in the case of a constant.
   - (b) a function declaration (arguments plus expression) in the case of a function.

3.3 Semantics

- **Syntax** is concerned with deciding which expressions are written correctly for DrRacket to process them.
- **Semantics** is concerned with assigning a value to each syntactically correct DrRacket expression.
- We must make sure that our DrRacket programs are unambiguous (i.e. have exactly one interpretation).
- A DrRacket program is a sequence of definitions and expressions.
- Our model involves the simplification of the program using a sequence of steps (i.e. substitution rules).
- Each step yields a valid DrRacket program.
- A fully-simplified program is a sequence of definitions which ends in a **value** (see Definition 3.4.1).

3.4 Values and Substitution

Definition 3.4.1. *A* ***value*** *is an expression which cannot be further simplified.*

- For example, `3` is a value, but `(+ 3 5)` and `(f 3 2)` are not.
- To ensure that we all agree on how to simplify any expression, we adopt the convention that we always simplify the **leftmost** expression requiring simplification first .
- As in the slides, we use ⇒ to separate steps in a simplification.

Examples:

1.
   ```
   (define (f x) (* x x))
   (define (g x y) (- x y))
   (g (f 2) (g 3 1))
   ⇒ (g (* 2 2) (g 3 1))
   ⇒ (g 4 (g 3 1))
   ⇒ (g 4 (- 3 1))
   ⇒ (g 4 2)
   ⇒ (- 4 2)
   ⇒ 2
   ```

2.
   ```
   (define (term x y) (* x (sqr y)))
   (term (- 5 3) (+ 1 2))
   ⇒ (term 2 (+ 1 2))
   ⇒ (term 2 3)
   ⇒ (* 2 (sqr 3))
   ⇒ (* 2 9)
   ⇒ 18
   ```

- **Exercise:** Run the Stepper on these two examples yourself.
- Be prepared to demonstrate tracing on exams.
- The Stepper may not use the same rules as our convention. Write your own trace to be sure you understand your code.
- **CQ 7**

3.5 Goals of Module 1

Refer to Slides 52-53.

4 Lecture 04

Outline

1. Administrivia
2. The Design Recipe - Examples (M2:1-20)
3. Tests/Examples - Check-Expect/Check-Within (M2:21-26)

4.1 Administrivia

1. Your submission for a01 need **not** follow the design recipe.
2. In DrRacket, `#i` indicates an **inexact** value.

4.2 The Design Recipe - Examples

Background: Recall that the **roots** (i.e. the values for x which make the polynomial equal 0) of the quadratic polynomial $ax^2 + bx + c$ are given by the **quadratic formula**

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

1. Solve, using the design recipe:
   Problem: Write a DrRacket function `quadratic-root-right` which consumes real numbers `a` > 0, `b` and `c` and produces the the right-hand root $\frac{-b+\sqrt{b^2-4ac}}{2a}$ of the quadratic polynomial $ax^2 + bx + c$. For now you may assume that $b^2 - 4ac \geq 0$.
   Solution: Solve it from scratch with examples / tests developed by the class. Refer to the associated DrRacket file for the finished version.
2. **Exercise:** Re-do the example for `quadratic-root-left`, and the root $\frac{-b-\sqrt{b^2-4ac}}{2a}$.
3. **Exercise:** Re-do the example with $b^2 - 4ac < 0$.

Examples of Functions that can return inexact values:

1. /, division
2. `sqrt`, square root
3. `expt`, exponentiation, like on a01.
4. later, `sin`, `cos`, etc, trig functions.

- **CQ 1**
- **CQ 2**

4.3 Tests/Examples - Check-Expect/Check-Within

1. Use `check-expect` if DrRacket will produce the expected value exactly.
2. Use `check-within` if DrRacket will produce an inexact value.

We saw both of these in the preceding example!

- **CQ 3**

5 Lecture 05

Outline

1. Administrivia
2. String Functions (M2:28-31)

3. Helper Functions (M2:32-36)
4. Using Constants (M2:37-41)
5. Summary of Module 2 (M2:42-43)

5.1 Administrivia

1. In the example from last time, we can enforce $a > 0$, instead of $a \neq 0$. Then there is no ambiguity about which root is the right hand root.
2. I will post all my .rkt example files on my Instructor Specific page soon.

5.2 String Functions

Examples:

1. **Problem:** Develop a function `start-end-middle`, which will consume a string `s` and construct a new string from it, composed of
   (a) the first character,
   (b) the last character and
   (c) the middle, i.e. the substring lying strictly between the first and the last characters.

   For example, `(start-end-middle "angle")` ⇒ `"aengl"`.

   Solution: Solve with the class; post the DrRacket file on your Instructor Specific Page afterward.

- **CQ 4**

5.3 Helper Functions

Remarks:

1. We already used helper functions in the preceding example.

5.4 Using Constants

Remarks:

1. In CS 115, the first natural number is 0, not 1. You may have used the convention that the first natural number is 1 in previous courses.

Examples:

1. **Problem:** Develop a function `electricity-charge`, which will consume natural numbers `total-kwh` and `high-rate-kwh` and produce the total charge, assuming that high rate kWh are charged at $ 0.15

and low rate kWh are charged at \$ 0.06. Assume that `total-kwh` $\geq$ `high-rate-kwh`. For example, `(electricity-charge 10 5)` $\Rightarrow$ 1.05. **Solution:** Solve with the class; post the DrRacket file on your Instructor Specific Page afterward.

- **CQ 5**

5.5 Summary of Module 2

Refer to slides 42-43.

6 Lecture 06

Outline

1. Administrivia
2. Intro to M3 (M3:1)
3. Boolean Expressions (M3:2-15)
4. Conditional Expressions (M3:16-19)
5. Substitution Rules for `conds` (M3:20-26)
6. Design Recipe Modifications (M3:27-33)

6.1 Administrivia

1. **Reminder:** Like anything that is worth marks, your Labs must be done individually.

6.2 Intro to M3

Refer to slide 1.

6.3 Boolean Expressions

1. Boolean values can seem confusing at first; happily you use them all the time, e.g. "If it rains tomorrow, then I will bring my umbrella." Any phrase that can be either `true` or `false` (and never both) is Boolean.
2. Recall that `string=?` compares two strings and returns `true` if they are equal and `false` otherwise.

3. Recall that a **predicate** is a function that returns a Booelan value in DrRacket.
4. `string=?` is an example of a **predicate** in DrRacket.
5. We adopt the convention of naming our predicates ending in "?", to remind the reader that it produces a Boolean value.
6. `and` and `or` are **special forms**. Not all arguments necessarily need to be simplified in order to evaluate.

Examples in DrRacket:

1. `(> (* 2 5) (- 12 4))` ⇒ `true`
2. `(string=? "dog" "cat")` ⇒ `false`

- CQ2

Example (from slide 14): Simplify
`(and (< 3 5) (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))`
Solution:

```
(and (< 3 5) (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and true (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and (or true (= 1 3)) (or (> 1 5) (> 2 5)))
⇒ (and true (or (> 1 5) (> 2 5)))
⇒ (and (or (> 1 5) (> 2 5)))
⇒ (and (or false (> 2 5)))
⇒ (and (or false))
⇒ (and (or))
⇒ (and false)
⇒ false
```

- CQ2

6.4 Conditional Expressions

Example: Write a function `range-for-input` which consumes a Natural number (`x`) and produces

- `"small"` if $x < 5$,
- `"medium"` if $5 \leq x < 15$ and
- `"large"` if $15 \leq x$.

Solution: Develop in DrRacket. Use your prepared version if you run out of time.

- CQ3

Remarks:

1. `cond` is like any other expression in DrRacket, in that it simplifies to a value. For example,
   (a) `(+ 4 (cond [(even? 3) 1][else 0]))` ⇒ 4
   (b) `(+ 4 (cond [(even? 2) 1][else 0]))` ⇒ 5

6.5 Substitution Rules for `cond`s

Example (from slide 21): Simplify

```
(define n 5)
(cond [(even?  n) "even"] [(odd?  n) "odd"])
```

Solution:

```
(cond [(even?  n) "even"] [(odd?  n) "odd"])
⇒ (cond [(even?  5) "even"] [(odd?  n) "odd"])
⇒ (cond [false "even"] [(odd?  n) "odd"])
⇒ (cond [(odd?  n) "odd"])
⇒ (cond [(odd?  5) "odd"])
⇒ (cond [true "odd"])
⇒ "odd"
```

Remarks:

1. The question in the second clause could equivalently be `else`.
2. On the first line, we substitute only the first `n`, unlike how we would simplify a function call.

- CQ4
- CQ5

6.6 Design Recipe Modifications

- Add Data Analysis and steps for creating the Q & A pairs for any `cond`s.

Remarks:

- `(and c)` and `(or c)` only make sense if `c` is a Boolean expression. For example, `(and 5)` produces a run-time error.

7 Lecture 07

Outline

1. Administrivia
2. Testing Conditional Expressions (M3:34-38)
3. Developing Predicates (M3:39-42)
4. Data Type Sym (M3:43-46)
5. Mixed Data Types (M3:47-53)
6. Changes to Syntax and Design Recipe (M3:54-55)
7. Summary of M3 (M3:56-57)

7.1 Administrivia

1. Stuff.

7.2 Testing Conditional Expressions

Principles:

1. Check every answer.
2. Make the purpose of each question clear.
3. Test all the relevant boundaries.
4. Testing Boolean Expressions:
    (a) `and`
        i. Test one case that evaluates to `true`.
        ii. Test one case that makes each component evaluate to `false`.
    (b) `or`
        i. Test one case that evaluates to `false`.
        ii. Test one case that makes each component evaluate to `true`.

Examples:

1. **Problem:** What minimal set of test cases is needed to fully test our `range-for-input` code from earlier?
    Solution: We need:
    (a) $x = 0$
    (b) $x = 4$
    (c) $x = 5$
    (d) $x = 14$
    (e) $x = 15$
    (f) $x = 25$

so 6 test cases in total.

2. **Problem:** What minimal set of test cases is needed to fully test this Boolean expression?

   ```
   (or (and (symbol=?  x 'a) (symbol=?  y 'b))
       (and (symbol=?  x 'c) (symbol=?  y 'd)))
   ```

   Solution: We need:
   (a) for outer `or`:
      i. both clauses `false` (can use $x =$ `'b`, $y =$ `'a`)
      ii. first clause `true` (can use $x =$ `'a`, $y =$ `'b`)
      iii. second clause `true` (can use $x =$ `'c`, $y =$ `'d`)
   (b) for first clause of outer `or`:
      i. making it `true` (can use $x =$ `'a`, $y =$ `'b`, included already)
      ii. making each part of it `false`
         A. (can use $x =$ `'b`, $y =$ `'b`)
         B. (can use $x =$ `'a`, $y =$ `'a`)
   (c) for second clause of outer `or`:
      i. making it `true` (can use $x =$ `'c`, $y =$ `'d`, included already)
      ii. making each part of it `false`
         A. (can use $x =$ `'d`, $y =$ `'d`)
         B. (can use $x =$ `'c`, $y =$ `'c`)

- CQ 6

7.3 Developing Predicates

- **Example:** Develop the function `before-cat-or-after-dog`, which consumes a string (`s`) and returns `true` if `s` is alphabetically strictly before `"cat"` or after `"dog"`, and `false` otherwise.
  Solution: Develop in DrRacket (with `cond` and without `cond`). Use your prepared versions if you run out of time.

7.4 Data Type Sym

- **Key Point:** Symbols are meaningful to people, not to DrRacket.

7.5 Mixed Data Types

Examples: Which Data Types Are Possible?

1. `-1`: `Num`, `Int` (not `Nat`)
2. `"-1"`: `Str`
3. `5`: `Num`, `Int`, `Nat`
4. `false`: `Bool`

N.B. All could be declared using `anyof`.

- CQ 7
- We can use `cond` to handle `anyof` types, with one Q & A pair per type, before simplifying.
- **Example** (from slide 53) `gen-even?`:

```
(define (gen-even?  info)
        (cond
              [(integer?  info) (even?  info)]
              [(or (equal?  info 'even) (equal?  info "even")) true]
              [else false]))
```

- **Remarks:**
  1. `(even?  info)` is a Boolean expression, which could evaluate to `true` or `false`.

7.6 Changes to Syntax and Design Recipe

1. **Syntax:** an **expression** is:
   (a) a value,
   (b) a single constant,
   (c) a function application,
   (d) a conditional expression, or
   (e) a Boolean expression.
2. **Design Recipe:** See slide 55.

7.7 Summary of M3

1. Refer to slides 56-57.

8 Lecture 08

Outline

1. Administrivia

2. Posn Structures (M4:1-12)
3. Dynamic Typing (M4:13-15)
4. Data Definitions / Structure Definitions (M4:16-18)
5. Design Recipe Modifications (M4:19-20)
6. Templates (M4:21-27)
7. Additions to Syntax/Semantics for Structures (M4:28-29)

8.1 Administrivia

1. Office Hours today in DC 3108, 1:00-1:50 PM.
2. I still need to type my notes for Lecture 06 and Lecture 07. I will get this done ASAP.

8.2 Posn Structures

- The building block of a structure is a **field**.
- `Posn` is built-in in DrRacket. You can use `Posn` with no additional setup.
- Recall that `Posn` is meant to capture the `x` and `y` co-ordinates of a point in the Cartesian plane (draw a diagram of an example).
- Historically, many students have thought that all structures are `Posns`. Do not make this mistake!
- **Example:** Simplify the given expression.
  ```
  ;; requires:  (posn-x myPosn) is non-zero
  (define (slope myPosn) (/ (posn-y myPosn) (posn-x myPosn)))
  (slope (make-posn (posn-y (make-posn 8 2)) (posn-x (make-posn
  5 7))))
  ```
  Solution:
  ```
  (slope (make-posn (posn-y (make-posn 8 2)) (posn-x (make-posn
  5 7))))
  ⇒ (slope (make-posn 2 (posn-x (make-posn 5 7))))
  ⇒ (slope (make-posn 2 5))
  ⇒ (/ 5 2)
  ⇒ 2.5
  ```
- CQ 1
- **Exercise:** Use `slope` as a helper function to write a new function to compute the slope of the line-segment from any `(make-posn x0 y0)` to any `(make-posn x1 y1)`.

- Recall that `Posn` is a structure that stores an `x` and a `y` together. DrRacket allows any values to be stored in `x` and `y`, numeric or otherwise. If you store something non-numeric and then try to process it as if it is numeric, you will get a run-time error. Try an example for yourself in DrRacket.
- CQ 2

8.3 Dynamic Typing

- As in the previous clicker question, DrRacket uses **dynamic typing**. This means that DrRacket determines the type of a value **at run time**.
- DrRacket is **interpreted**, not **compiled**, so there really is no other choice.
- Contracts are comments; they cannot be enforced by DrRacket.
- However we will assume for our purposes in the course that our contracts are always observed by the caller.
- So after adopting the convention for `Posn` structures on slide 15, we know that the `x` and `y` values of a `Posn` will always be numeric.
- For Pros and Cons of dynamic typing: Refer to slide 14.

8.4 Data Definitions / Structure Definitions

- A **Data Definition** is a comment specifying a data type.
- For a structure it specifies the numbers and types of our fields.
- Defining a structure automatically creates:
  - Constructor
  - Selectors
  - Predicate

  Example:

```
(define-struct course (subject number title))
;; A Course is a (make-course Str Nat Str)
;; requires
;;          subject is the subject area code, e.g.  "CS"
;;          number is the catalog number, e.g.  115
;;          title is the course title,
;;          e.g.  "Introduction to Computer Science 1"
```

Then a template for a function consuming a `Course is`

```
;; course-template:  Course -> Any
(define (course-template mycourse)
(...    (course-subject mycourse)
        (course-number mycourse)
        (course-title mycourse)...))
```

8.5 Design Recipe Modifications

- Refer to the slides for full details.

8.6 Templates

- **Example:** Develop a function `update-title` which consumes `oldcourse` (a `Course`) and `newtitle` (a string), and produces a new `Course` with the same `subject` and `number`, and `title` set to `newtitle`.
  Solution:

```
(define (update-title oldcourse newtitle)
        (make-course (course-subject oldcourse)
                     (course-number oldcourse)
                    newtitle))
```

- CQ 3

9 Lecture 09

Outline

1. Administrivia
2. Additions to Syntax/Semantics for Structures (M4:28-29)
3. More Examples (M4:30-34)
4. Using the Design Recipe for Compound Data (M4:35-41)
5. Mixed Data and Structures (M4:42-50)
6. Summary of M4 (M4:51-52)

9.1 Administrivia

1. Office Hours (for my other course) today in DC 3108, 1:00-1:50 PM.
2. I still need to type my notes for Lecture 06 and Lecture 07. I will get this done ASAP.

9.2 Additions to Syntax/Semantics for Structures

- The list of legal values now include structures!
- The simplification rules for the selectors and the type predicate are the obvious things (See slides 28-29).

9.3 More Examples

Example: Use these definitions.

```
(define-struct instructor (last-name
                           first-name
                           office))
;; An Instructor is a (make-instructor Str Str Str)
;; requires
;;    office is the building and room number,
;;       e.g. "DC 3108"

(define-struct TA (last-name
                   first-name
                   loginid))
;; A TA is a (make-TA Str Str Str)
;; requires
;;    loginid is the email prefixt,
;;       e.g. "cbrown" has email address "cbrown@uwaterloo.ca"

;; A Staff is one of:
;; * an Instructor or
;; * a TA

(define-struct course (subject number title staff1 staff2))
;; A Course is a (make-course Str Nat Str Staff Staff)
```

```
;; requires
;;    subject is the subject area code, e.g. "CS"
;;    number is the catalog number, e.g. 115
;;    title is the course title,
;;       e.g. "Introduction to Computer Science 1"

;;Constants
(define emaildomain "@uwaterloo.ca")
(define instructor1
  (make-instructor "Roberts" "Collin" "DC 3108"))
(define instructor2
  (make-instructor "Kahn" "Wassif" "DC 3126"))
(define TA1
  (make-TA "Brown" "Charlie" "cbrown"))
(define TA2
  (make-TA "VanPelt" "Linus" "lvpelt"))
(define test-course1
  (make-course
   "CS"
   115
   "Introduction to Computer Science 1"
   instructor1
   TA1))
(define test-course2
  (make-course
   "CS"
   115
   "The Best Course Ever"
   TA2
   instructor2))
```

1. Develop a template for a consumer of an `Instructor`.
   Solution:

   ```
   ;; instructor-template: Instructor -> Any
   (define (instructor-template myinstructor)
      (...(instructor-last-name myinstructor)
          (instructor-first-name myinstructor)
          (instructor-office myinstructor)
   ```

```
        ...)
```

2. Develop a template for a consumer of a `TA`.
   Solution:

```
;; TA-template: TA -> Any
(define (TA-template myTA)
   (...(TA-last-name myTA)
       (TA-first-name myTA)
       (TA-office myTA)
       ...)
```

3. Develop a template for a consumer of a `Course`.
   Solution:

```
;; course-template: Course -> Any
(define (course-template mycourse)
(... (course-subject mycourse)
     (course-number mycourse)
     (course-title mycourse)
         (cond
            [(instructor? (course-staff1 mycourse)
               (...(instructor-last-name (course-staff1 mycourse))
                   (instructor-first-name (course-staff1 mycourse))
                   (instructor-office (course-staff1 mycourse))
                   ...)]
            [(TA? (course-staff1 mycourse)
               (...(TA-last-name (course-staff1 mycourse))
                   (TA-first-name (course-staff1 mycourse))
                   (TA-loginid (course-staff1 mycourse))
                   ...)]
         )
         (cond
            [(instructor? (course-staff2 mycourse)
               (...(instructor-last-name (course-staff2 mycourse))
                   (instructor-first-name (course-staff2 mycourse))
                   (instructor-office (course-staff2 mycourse))
                   ...)]
            [(TA? (course-staff2 mycourse)
```

```
                   (...(TA-last-name (course-staff2 mycourse))
                       (TA-first-name (course-staff2 mycourse))
                       (TA-loginid (course-staff2 mycourse))
                       ...)]
           )
  ...))
```

4. Use the template to develop a function `first-staff-contact` to return the contact information (office for an instructor; email for a TA) for the first staff member of a consumed Course, `mycourse`.
   Solution: Do this in DrRacket. Use your prepared version if you run out of time.
   Remarks:
   (a) We could make a helper function to produce the email address from the loginid, but we won't. This could help readability but it will not shorten the code in this example.
   (b) In M5, `lists` will provide us a more elegant way to store several course staff members (without needing to know how many there will be up front).

9.4 Using the Design Recipe for Compound Data

9.5 Mixed Data and Structures

- CQ4
- CQ5
- CQ6

9.6 Summary of M4

10 Lecture 10

Outline

1. Administrivia
2. Data Definitions for Lists (M5:2-13)
3. Template for a List of Symbols (M5:14-22)

10.1 Administrivia

1. Office Hours today in DC 3108: 1:00-1:50 PM.
2. I am still behind on typing L06/07 and uploading my clicker files. I will get caught up soon.

10.2 Data Definitions for Lists

A `list` is either

- `empty` or
- `(cons f r)`, where
  - `f` is a value and
  - `r` is a `list`.

Remarks:

1. This definition is **recursive**, i.e. it refers back to itself.
2. It might appear that this definition can continue referring back to itself forever, in the case of an infinite list. However in CS 115 all of our lists will be finite. So in any actual example, when we start peeling off the first element of some list and retaining the rest, the rest will eventually equal `empty`.
3. The empty list `empty` is a built-in constant.
4. The constructor function `cons` is a built-in function.
5. `f` is the first item in the list. `r` is the rest of the list.
6. `first` and `rest` are also built-in functions in DrRacket. They can be called on any non-empty list. Calling either one on an empty list results in a run-time error.

Example (from slide 11): Suppose we have

```
(define mylist (cons 1 (cons 'blue (cons true empty))))
```

Then what expression evaluates to:

1. `1`?
   Solution:
   `(first mylist)`
2. `'blue`?
   Solution:
   `(first (rest mylist))`
3. `(cons true empty)`?
   Solution:
   `(rest (rest mylist))`

4. `true`?
   Solution:
   `(first (rest (rest mylist)))`
5. `empty`?
   Solution:
   `(rest (rest (rest mylist)))`

- CQ 1
- CQ 2

```
;; A (listof Sym) is one of:
;; * empty
;; * (cons Sym (listof Sym))
```

Remarks:

1. Informally: a list of symbols is either empty, or it consists of a first symbol followed by a list of symbols (the rest of the list).
2. This is a recursive definition, with a base case, and a recursive (self-referential) case.
3. Lists are the main data structure in standard Racket.
4. Last week our `Course` structure had `Staff1` and `Staff2` substructures. This worked under the assumption that every course has exactly two course staff. A better approach is to use `(listof Staff)`, so that we can capture any number of staff members for a given course.

10.3 Template for a List of Symbols

See M5:16

```
;; los-template: (listof Sym) -> Any
(define (los-template alos)
   (cond
      [(empty? alos) ... ]
      [else (... (first alos) ...
      ... (los-template (rest alos)) ... )]))
```

Examples:

1. **Problem:** Develop a predicate `list-has-symbol?` which consumes a list (`mylist`) and returns `true` if `mylist` contains at least one symbol and `false` otherwise.
   Solution: See the solution in DrRacket.

2. **Problem:** Give a condensed trace to simplify
   `(list-has-symbol? (cons 1 (cons 'blue (cons true empty))))`
   .
   Solution:

   ```
   (list-has-symbol?  (cons 1 (cons 'blue (cons true empty))))
   ⇒ (or (symbol?  1) (list-has-symbol?  (cons 'blue (cons true empty))))
   ⇒ (or false (list-has-symbol?  (cons 'blue (cons true empty))))
   ⇒ (or (list-has-symbol?  (cons 'blue (cons true empty))))
   ⇒ (or (or (symbol?  'blue) (list-has-symbol?  (cons true empty))))
   ⇒ (or (or true (list-has-symbol?  (cons true empty))))
   ⇒ (or true)
   ⇒ true
   ```

3. **Problem:** Give a condensed trace to simplify
   `(list-has-symbol? (cons 1 (cons 3.5 (cons "dog" empty))))` .
   Solution: Exercise. (You should get `false`.)
4. Develop a function `last-list-element` which consumes a `list` (`mylist`) and returns
   (a) `"empty list"` if `mylist` is `empty`, or
   (b) the last element of `mylist` if `mylist` is non-`empty`.
   Solution: Exercise. We will solve this one in Lecture 12 on Feb 13 if not sooner.

11 Lecture 11

Outline

1. Administrivia
2. Template for a List of Symbols (M5:14-22)
3. Structural Recursion (M5:23-32)
4. Examples (M5:33-42)
5. Wrapping a Function (M5:43)
6. String Functions (M5:44-47)
7. Example: Portions of a Total (M5:48-49)

11.1 Administrivia

1. I am caught up on uploading my iClicker files now.

2. I am not yet caught up on typing my notes from L06/07. I will get this finished over the weekend at the latest.

11.2 Template for a List of Symbols

Example from slides: `my-length`

```
;; (my-length alos) produces the number of symbols in alos.
;; my-length:  (listof Sym) -> Nat
;; Examples:
(check-expect (my-length empty) 0)
(check-expect (my-length (cons 'a (cons 'b empty))) 2)

(define (my-length alos)
        (cond
             [(empty?  alos) 0]
             [else (+ 1 (my-length (rest alos)))]))
```

Condensed trace of `my-length`

```
(my-length (cons 'a (cons 'b empty)))
⇒ (+ 1 (my-length (cons 'b empty)))
⇒ (+ 1 (+ 1 (my-length empty)))
⇒ (+ 1 (+ 1 0))
⇒ 2
```

- CQ 3

11.3 Structural Recursion

- **Idea:** We need recursive programs in order to handle recursively-defined data.
- **Strategy:** Create templates from our data definitions. Then use these templates as starting points to write our functions.
- **Contracts:** Use the `(listof` __________ `)` notation to indicate the type(s) that the list elements can have. E.g.
  1. `(listof Sym)` for symbols
  2. `(listof Num)` for numerics
  3. `(listof (anyof Str Sym))` for strings or symbols
- CQ 4

11.4 Examples

11.5 Wrapping a Function

11.6 String Functions

Example: Develop a function `count-after-cat` which consumes a list of strings (`alos`) and returns the number of elements of `alos` which are alphabetically after `"cat"`.

Solution:

1. Note that this example is similar to `count-apples` in the slides.
2. The string `"cat"` should be a named constant.
3. Thinking about this, we also realize that we should write a helper function to determine whether we should add 0 or 1 for a given list element, after comparing it to the given string (`"cat"` for us), which should be a parameter. This will make our code more readable, and will facilitate re-use for other comparisons. At the end of the example, we created `count-after-fish`, by simply changing `"cat"` to `"fish"` in the constants section.
4. Complete this in DrRacket. Use your prepared version if you run out of time.

11.7 Example: Portions of a Total

- Show and explain the example from slides 48-49 here.
- CQ 5

12 Lecture 12

Outline

1. Administrivia
2. Nonempty Lists (M5: 50-51)
3. Functions with Multiple Base Cases (M5: 52)
4. Structures Containing Lists (M5: 53-55)
5. Lists of Structures (M5: 56-72)
6. Summary of M5 (M5: 73-74)

12.1 Administrivia

1. The iClicker grades on LEARN are up to date now. Please let me know about any outstanding iClicker problems.
2. Office Hours today in DC 3108, 1:00-1:50 PM.
3. I still need to type my notes for Lecture 06 and Lecture 07. I will get this done ASAP.
4. The Mid-Term Exam will cover up to the end of Module 5 (i.e. to the end of Lecture 12).
5. <u>A reminder about how `cons` works:</u> Recall that `(cons f r)` makes sense only if `f` is an element, and `r` is a `list`. For example, processing the expression `(cons 1 2)` will generate a run-time error, since `2` is not a list.
6. Do **not** use the `(list a1` . . . `an)` notation for a05.

12.2 Nonempty Lists

```
;; A nonempty list of numbers (Neln) is either:
;; * (cons Num empty)
;; * (cons Num Neln)
```

- CQ 6

Create the implementation of `max-lon` in DrRacket here.

12.3 Functions with Multiple Base Cases

Develop `all-same?` in DrRacket here.

12.4 Structures Containing Lists

Solve an example like `server-tips` in DrRacket here, only if there is time.

12.5 Lists of Structures

Example: Recall this structure definition from Module 4.

```
(define-struct line (endpt1 endpt2))
;; A Line is a (make-line Posn Posn).
```

Develop the function `midpoints` which consumes a list of `Line`s (`alol`) and returns the corresponding list of their midpoints.
Solution: Develop the solution in DrRacket.

- CQ 7

12.6 Summary of M5

Refer to slides 73-74.

13 Lecture 13

Outline

1. Administrivia
2. Intro to Module 6 (M6:1)
3. Natural Numbers (M6:2-6)
4. Countdown (M6:7-10)
5. Subintervals of the Natural Numbers (M6:11-16)
6. Countdown-to (M6:17-20)
7. Countup (M6:21-24)
8. Countup-to (M6:25-29)

13.1 Administrivia

1. Please do not pass through the instructor's corner while the cabinet door is open. Please go around instead.

13.2 Intro to Module 6

13.3 Natural Numbers

Recall: The Natural numbers are $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. The Natural numbers are defined recursively on slide 2 by

```
;; A natural number (Nat) is either:
;; * 0, or
;; * 1 + Nat
```

13.4 Countdown

Idea: Work on a descending list of Natural numbers $\{n, n-1, \ldots, 1, 0\}$, for some Natural number n.
Remarks:

1. The `add1` and `sub1` functions are built-in in DrRacket. You can just use them whenever you need them.

Problem: Develop the function **countdown-by** which consumes a natural number (`multiplier`) and another natural number (`last-multiple`) and produces the multiples of `multiplier` from `last-multiple * multiplier` down to `0`, in descending order. For example

```
(countdown-by 4 5) ⇒ (cons 20 (cons 16 (cons 12 (cons 8 (cons 4
(cons 0 empty))))))
```

Solution: Develop in DrRacket. Use your prepared version if you run out of time.
Problem: Give a condensed trace to simplify
`(countdown-by 4 5)`

Solution:

```
(countdown-by 4 5)
⇒ (cons (* 4 5) (countdown-by 4 4))
⇒ (cons 20 (countdown-by 4 4))
⇒ (cons 20 (cons (* 4 4) (countdown-by 4 3)))
⇒ (cons 20 (cons 16 (countdown-by 4 3)))
⇒ (cons 20 (cons 16 (cons (* 4 3) (countdown-by 4 2))))
⇒ (cons 20 (cons 16 (cons 12 (countdown-by 4 2))))
⇒ (cons 20 (cons 16 (cons 12 (cons (* 4 2) (countdown-by 4 1)))))
⇒ (cons 20 (cons 16 (cons 12 (cons 8 (countdown-by 4 1)))))
⇒ (cons 20 (cons 16 (cons 12 (cons 8 (cons (* 4 1) (countdown-by
4 0))))))
⇒ (cons 20 (cons 16 (cons 12 (cons 8 (cons 4 (countdown-by 4 0))))))
⇒ (cons 20 (cons 16 (cons 12 (cons 6 (cons 4 (countdown-by 4 0))))))
⇒ (cons 20 (cons 16 (cons 12 (cons 6 (cons 4 (cons 0 empty))))))
```

- CQ 1

13.5 Subintervals of the Natural Numbers

13.6 Countdown-to

13.7 Countup

13.8 Countup-to

Problem: Develop the function **countup-by** which consumes a natural number (`multiplier`) and another natural number (`last-multiple`) and produces the multiples of `multiplier` from `0` up to `last-multiple * multiplier`, in ascending order. For example
`(countup-by 4 5)` ⇒ `(cons 0 (cons 4 (cons 8 (cons 12 (cons 16 (cons 20 empty))))))`
Solution: Develop in DrRacket. Use your prepared version if you run out of time.
Problem: Give a condensed trace to simplify
`(countup-by 4 5)`

Solution:

```
(countup-by 4 5)
⇒ (countup-from-by 4 0 5)
⇒ (cons (* 0 4) (countup-from-by 4 1 5))
⇒ (cons 0 (countup-from-by 4 1 5))
⇒ (cons 0 (cons (* 1 4) (countup-from-by 4 2 5)))
⇒ (cons 0 (cons 4 (countup-from-by 4 2 5)))
⇒ (cons 0 (cons 4 (cons (* 2 4) (countup-from-by 4 3 5))))
⇒ (cons 0 (cons 4 (cons 8 (countup-from-by 4 3 5))))
⇒ (cons 0 (cons 4 (cons 8 (cons (* 3 4) (countup-from-by 4 4 5)))))
⇒ (cons 0 (cons 4 (cons 8 (cons 12 (countup-from-by 4 4 5)))))
⇒ (cons 0 (cons 4 (cons 8 (cons 12 (cons (* 4 4) (countup-from-by
4 5 5))))))
⇒ (cons 0 (cons 4 (cons 8 (cons 12 (cons 16 (countup-from-by 4 5
5))))))
⇒ (cons 0 (cons 4 (cons 8 (cons 12 (cons 16 (cons (* 5 4) empty))))))
⇒ (cons 0 (cons 4 (cons 8 (cons 12 (cons 16 (cons 20 empty))))))
```

Remarks:

1. The helper function was needed in this example, because the specification of the main function omitted a parameter which was required by

the countup template.
2. Hence we included the needed parameter in the helper function, then wrapped it in the main function.

- CQ 2

14 Lecture 14

Outline

1. Administrivia
2. Example: Countdown-To (M6:30-38)
3. Modifications to Templates (M6:39-40)
4. Example: Sorting (M6:41-48)
5. List Abbreviations (M6:49-51)

14.1 Administrivia

1. Office Hours today in DC 3108, 1:00-1:50 PM.
2. No marking of the mid-term exam has been done yet. We will mark the exam on Wednesday, Feb 28.
3. Clicker questions that don't count (about the mid-term).

14.2 Example: Countdown-To

1. **Problem:** Develop the function **countdown-by-to** which consumes a natural number (`multiplier`), a natural number (`last-multiple`) and a natural number (`first-multiple`) and produces the multiples of `multiplier` from `last-multiple * multiplier` down to `first-multiple * multiplier`, in descending order. For example
   `(countdown-by-to 4 5 3)` ⇒ `(cons 20 (cons 16 (cons 12 empty)))`
   Solution: Develop in DrRacket. Use your prepared version if you run out of time.
2. **Problem:** Give a condensed trace to simplify
   `(countdown-by-to 4 5 3)`

   Solution:
   ```
   (countdown-by 4 5 3)
   ⇒ (cons (* 4 5) (countdown-by 4 4 3))
   ```

⇒ `(cons 20 (countdown-by 4 4 3))`
⇒ `(cons 20 (cons (* 4 4) (countdown-by 4 3 3)))`
⇒ `(cons 20 (cons 16 (countdown-by 4 3 3)))`
⇒ `(cons 20 (cons 16 (cons (* 4 3) empty)))`
⇒ `(cons 20 (cons 16 (cons 12 empty)))`

3. **Example:** Do the example `gcd-three` from the slides. Develop in DrRacket. Use your prepared version if you run out of time.

14.3 Modifications to Templates

Refer to slides 39-40.

14.4 Example: Sorting

1. **Example:** `insert`, from the slides, if there is time. Post it after the lecture in any case.
2. **Problem:** Develop the function `insert-non-increasing` to insert `n` into the non-increasing list `alon`.
   Solution: Develop in DrRacket. Use your prepared version if you run out of time.

14.5 List Abbreviations

Refer to slides 49-51. We already used these abbreviations in the previous example.

15 Lecture 15

Outline

1. Administrivia
2. Lists Containing Lists (M6:52-62)
3. Dictionaries (M6:63-66)
4. Different Kinds of Lists (M6:67)
5. Summary of M6 (M6:68-69)

15.1 Administrivia

1. Sorry, I still need to revise my lecture notes for Lecture 14 (including posting the DrRacket examples). I will do this by the end of the day on Friday.
2. The mid-term is marked, but the model solutions still need a few corrections before we release them. The marked mid-term and solutions will be posted on Friday.

15.2 Lists Containing Lists

- CQ 3 (recall you have a DrRacket file to demonstrate the right answer)
- CQ 4
- CQ 5
- The same thing can often be accomplished using a structure or a list of lists.
- Using structures usually makes the solution more readable, and less flexible.

15.3 Dictionaries

A dictionary contains a number of unique **keys**, each with an associated **value**.
Recall these definitions.

```
;; An association (As) is (list Num Str),
;; where
;; * the first item is the key,
;; * the second item is the associated the value.
;; An association list (AL) is one of
;; * empty
;; * (cons As AL)
;; Note: All keys must be distinct.
```

- Example lookup-al, from slide 65!
- CQ 6
- CQ 7

15.4 Different Kinds of Lists

We will work more with lists of lists in Module 10.

15.5 Summary of M6

Refer to slides 68-69.

16 Lecture 16

Outline

1. Administrivia
2. Intro to M7 (M7:1-2)
3. One List Going Along For The Ride (M7:3-6)
4. Processing Two Lists in Lockstep (M7:7-13)
5. Processing Two Lists at Two Different Rates (M7:14-23)

16.1 Administrivia

1. Collin to type ASAP.

16.2 Intro to M7

16.3 One List Going Along For The Ride

16.4 Processing Two Lists in Lockstep

16.5 Processing Two Lists at Two Different Rates

17 Lecture 17

Outline

1. Administrivia
2. Example: nth-occur-suffix (M7:24-31)
3. Example: midpoints of pairs of Posns (M7:32)
4. Example: list=? (M7:33-41)
5. Summary of M7 (M7:42)

17.1 Administrivia

1. I still need to type my lecture notes for Lecture 16. I will get this done ASAP.

17.2 Example: nth-occur-suffix

Roberts' Example: Develop the function `nth-occur-prefix`, as described below.

```
;; (nth-occur-prefix asym n alist) produces sublist
;; ending at the nth occurrence of asym in alist.
;; nth-occur-prefix:
;; Sym Nat (listof Sym) -> (listof Sym)
;; requires:  n >= 1
;; Examples:
(check-expect (nth-occur-prefix 'a 1 empty) empty)
(check-expect (nth-occur-prefix 'a 1 (list 'a 'c)) (list 'a))
(check-expect (nth-occur-prefix 'a 2 empty) empty)
(check-expect (nth-occur-prefix 'a 2 (list 'a 'b 'a 'c)) (list 'a 'b 'a))
(define (nth-occur-prefix sym n alist) ...  )
```

Solution: Develop in DrRacket. Use your prepared solution if you run out of time.

17.3 Example: midpoints of pairs of Posns

17.4 Example: list=?

Develop the function `list=?`, as described below.

```
;; Constants
(define mylist-1 (list 1 2 3 4 5))
(define mylist-2 (list 1 2 3 4 6))
(define mylist-3 (list 1 2 3 4))
(define mylist-4 (list 2 2 3 4 5))

;; (list=?  list1 list2) produces true if
;; list1 and list2 are equal.
```

```
;; list=?: (listof Any) (listof Any) -> Bool
;; Examples:
(check-expect (list=? mylist-1 mylist-1) true)
(check-expect (list=? mylist-1 mylist-2) false)
(check-expect (list=? mylist-1 mylist-3) false)
(check-expect (list=? mylist-1 mylist-4) false)
(define (list=? list1 list2) ... )
```

Solution: Develop in DrRacket. Use your prepared solution if you run out of time.

- CQ 4

17.5 Summary of M7

18 Lecture 18

Outline

1. Administrivia
2. Intro to M8 (M8:1)
3. Binary Arithmetic Expressions
    (a) As Trees (M8:2-5)
    (b) Data Definition (M8:6-8)
    (c) Template (M8:9-10)
4. Binary Search Trees
    (a) Binary Trees (M8:11-17)
    (b) Additional (Ordering) Property for a Binary Search Tree (M8:18-22)

18.1 Administrivia

1. Office Hours today in DC 3108, 1:00-1:50 PM.
2. I still need to type my lecture notes for Lecture 16. I will get this done ASAP.
3. I also still owe you some examples to explain recent clicker question answers. I will get these done ASAP.

18.2 Intro to M8

Remarks:

1. As with lists, the principal goal of M8, namely **binary search trees** are another classic data structure in CS.
2. A Quotation from a CS Prof of mine:

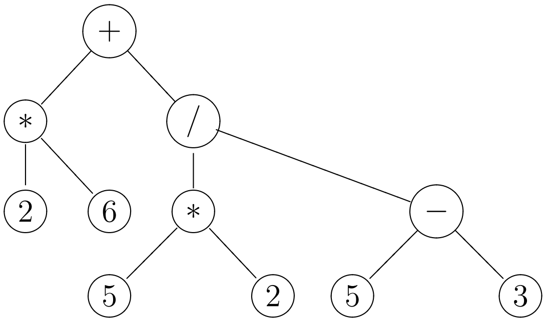
   > Every problem in CS can be boiled down to a **search** or a **sort**.

   We sorted in M6; we will use BSTs to search efficiently in M8.
3. Motivation: How does the fact that the dictionary is sorted in alphabetic order help you to find words more quickly? (We will leverage the answer to this question, when we revisit dictionaries using BSTs.)
4. We build up to the idea step-by-step, starting with **binary trees** (motivating them via **binary arithmetic expressions**).
5. Review the terminology on slide 4 before going on.
6. In a binary tree, every internal node has **at most two children** (i.e. it could have 1 or 2 children).
7. By definition, a leaf has no children.
8. In the trees we use to represent binary arithmetic expressions, every internal node has **exactly two children** (because all the operations we are using are binary, i.e. each operation takes exactly two arguments).
9. Later we will see examples of BSTs (which are binary trees with an additional ordering property), in which internal nodes have 1 or 2 children.

18.3 Binary Arithmetic Expressions

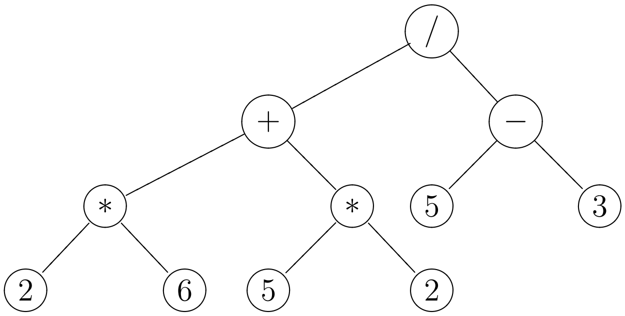
18.3.1 As Trees

1. **Q:** Construct the arithmetic expression that is represented by the tree.



A: $(2 * 6) + ((5 * 2)/(5 - 3))$

2. **Q:** Construct the tree that represents the arithmetic expression:
   $((2 * 6) + (5 * 2))/(5 - 3)$
   A:



- CQ 1

18.3.2 Data Definition

```
(define-struct binode (op arg1 arg2))
;; A Binary arithmetic expression Internal Node (BINode)
;; is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)
;; A Binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode
;; Examples
5
(make-binode '* 2 6)
(make-binode '+ 2 (make-binode '- 5 3))
```

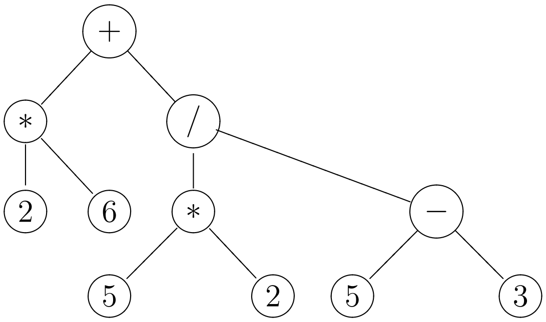
18.3.3 Template

```
;; binexp-template:  BinExp -> Any
(define (binexp-template ex)
        (cond [(number?  ex)...]
              [else (...(binode-op ex)...
                    (binexp-template (binode-arg1 ex))...
                    (binexp-template (binode-arg2 ex))...)]))
```

Examples:

1. On a08 you will develop some programs using this template.

2. **Q:** Construct the DrRacket expression that is represented by the tree.



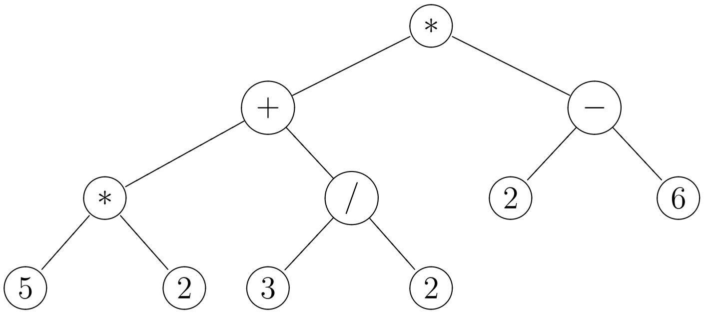
A:

```
(make-binode '+ (make-binode '* 2 6)
(make-binode '/ (make-binode '* 5 2) (make-binode '- 5 3)))
```

3. **Q:** Construct the tree that represents the DrRacket expression.

```
(make-binode '* (make-binode '+ (make-binode '* 5 2) (make-binode '/ 3 2))
(make-binode '- 2 6)))
```

A:



- CQ 2

18.4 Binary Search Trees

18.4.1 Binary Trees

(From slide 13)

```
(define-struct node (key val left right))
;; A Node is a (make-node Nat Str BT BT)
```

```
;; A binary tree (BT) is one of:
;; * empty, or
;; * (make-node Nat Str BT BT)
```

Remarks:

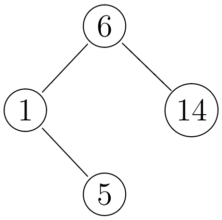
1. A node of a tree stores a key and a value (i.e. an Association of an Association List).
2. When discussing the structure of a BST, it often suffices to show the keys only.

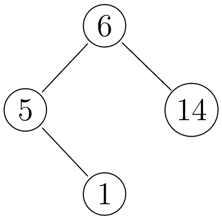
18.4.2 Additional (Ordering) Property for a Binary Search Tree

(From slide 17)

```
(define-struct node (key val left right))
;; A binary search tree (BST) is either
;; * empty, or
;; * (make-node Nat Str BST BST),
;; which satisfies the ordering property recursively:
;; * every key in left is less than key
;; * every key in right is greater than key
```

Examples: **Q:** Which of the following are, and which are not, BSTs, and why?

1. 

   A: Yes
2. 

A: No. The internal node with key 5 has a value (namely 1) on its right subtree which is not greater than 5. The ordering property is not satisfied. This example reminds us that the ordering property must hold at **every internal node**, not only at the root node.

19 Lecture 19

Outline

1. Administrivia
2. Example: `count-leaves`
3. Example: `count-values`
4. Searching in a BST (M8:24-28)
5. Creating a BST (M8:29-30)

19.1 Administrivia

1. I still need to type my lecture notes for Lecture 16. I will get this done over the weekend.

19.2 Example: `count-leaves`

Problem: Develop the function `count-leaves` which consumes a BT (`t`) and produces the number of leaf nodes in `t`.

Remarks:

1. This function consumes a BT.
2. The ordering property of a BST is not yet enforced here.
3. Recall: a **leaf** is a node that has no children (i.e. a node for which both the left and right subtrees are empty).
4. A node that has children is an **internal node**.
5. See slide 16 for the details of the implementation.

- CQ 3

19.3 Example: `count-values`

Problem: Develop the function `count-values` which consumes a BST (`abst`) and a string `v` and produces the number of nodes in `abst` that have value equal to `v`.

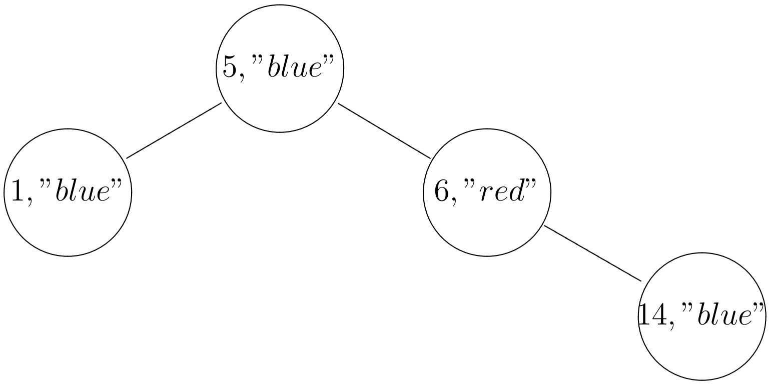
Remarks:

1. Keys are unique in a BST. The ordering property of a BST guarantees that no key can be duplicated.
2. So counting the number of occurrenes of a keys in a BST could only produce 0 or 1.
3. The function `count-values` is less trivial, since duplicate values can be present in a BST.
4. We do **not** use the ordering property of a BST to implement `count-values`.
5. See slide 23 for the details of the implementation.

- A **traversal** is a systematic way of visiting every node of the tree.
- The most commonly used type is called **depth-first**, in which the search tree is deepened as much as possible on each child before going to the next sibling.
- At node N you must do these three things:
  - (N) Process N itself.
  - (L) Recursively traverse N's left subtree.
  - (R) Recursively traverse N's right subtree.

  We may do these things in any order and still have a legitimate traversal. If we do (L) before (R), we call it **left-to-right traversal**, otherwise we call it **right-to-left traversal**.

Example: Let t be the following binary search tree.

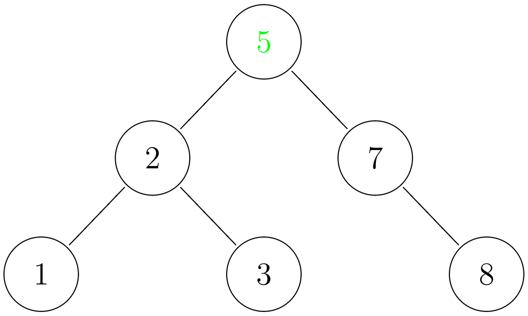


Then

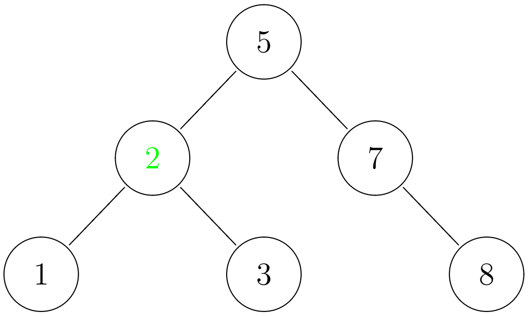
- (count-values t "blue") produces 3, and
- (count-values t "red") produces 1.

19.4 Searching in a BST

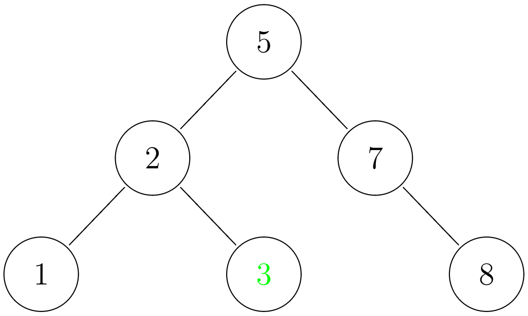
Here are some concrete examples of searching in a BST.



- We start from the root of the tree.
- If we search for 5, then we find it at the root and we are done.



- If we search for 2, then we do not find it at the root.
- Since $2 < 5$, we can ignore the right subtree, and only recursively search the left subtree.



- If we search for 3, then we do not find it at the root.

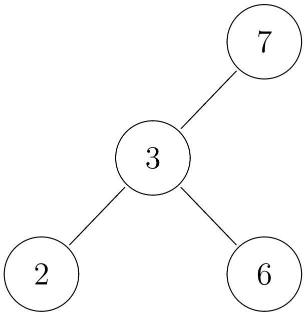
- Since $3 < 5$, we can again ignore the right subtree, and only recursively search the left subtree.
- Since $3 > 2$, we can ignore the left subtree, and only recursively search the right subtree.

The function `search-bst`

1. `(search-bst n t)` produces the value associated with key `n` in the BST `t`, if possible, or `false` if `t` does not contain key `n`.
2. Because of our observations from the examples above, we only need to recursively search **one** subtree when we fail to find the key at the root.
3. See slide 28 for the details of the implementation.

19.5 Creating a BST

Here is the binary tree formed by
(add 6 ”6” (add 2 ”2” (add 3 ”3” (add 7 ”7” empty)))):



- CQ 4

20 Lecture 20 - Abstract List Functions

Outline

1. Administrivia
2. Filter (M9: 1-10)
3. Map (M9: 11-14)
4. Build-List (M9: 15-18)
5. Foldr (M9: 19-35)
6. Summary

20.1 Administrivia

1. Office Hours today in DC 3108, 1:00-1:50 PM.
2. Note the change to the first example for Q2 on a08.
3. If you need someone to talk to about any mental health issues, then you can talk to me, and I will direct you to the right professionals.

20.2 Filter

Problem: Develop the function `sublist-gt-5`, which consumes a list of `Nat` (`alon`) and produces the sublist from `alon` of elements that are strictly greater than 5.
Solution: Solve in DrRacket. Post your solution afterwards.

- **CQ 1**

20.3 Map

Problem: Develop the function `y-equals-negative-x`, which consumes a list of `Num` (`alon`) and produces the list of `Posn` where each `Posn` represents the point on the line $y = -x$ having the x-co-ordinate equal to the element of `alon`.
Solution: Solve in DrRacket. Post your solution afterwards.

- **CQ 2**

20.4 Build-List

Problem: Develop the function `make-list`, which consumes a `Nat` (`n`) and produces the list of `Nat` from 0 up to `n-1`.
Solution: Solve in DrRacket. Post your solution afterwards.
Remarks:

1. The function `build-list` is most useful in conjunction with other abstract list functions.

- **CQ 3**

20.5 Foldr

Problem: Develop the function `all-true?`, which consumes a list of `Bool` (**alob**) and produces `true` if and only if every element of **alob** is true.

Solution: Solve in DrRacket. Post your solution afterwards.
Problem: Trace `(all-true? (list true false true))`
Solution:

```
(all-true? (list true false true))
⇒ (foldr binary-and true (list true false true))
⇒ (binary-and true (foldr binary-and true
                     (list false true)))
⇒ (binary-and true (binary-and false
                     (foldr binary-and true (list true))))
⇒ (binary-and true (binary-and false
                     (binary-and true
                     (foldr binary-and true (list)))))
⇒ (binary-and true (binary-and false
                     (binary-and true true)))
⇒ (binary-and true (binary-and false true))
⇒ (binary-and true false)
⇒ false
```

Exercise: Trace `(all-true? (list true true true))`
Exercise: Develop the analogous function `one-true?` which uses `or` in place of `and`.
Remarks:

1. You might find this example has a useful idea for solving one or more of the problems on a09.

20.6 Summary

Important Quotation from Slide 33: Abstract list functions should be used **judiciously**, to replace **relatively simple** uses of recursion.

- **CQ 4**

21 Lecture 21 - Local Definitions

Outline

1. Administrivia
2. Local Definitions (M9: 36-44)
3. Semantics of Local Definitions (M9: 45-53)

4. Nested Local Expressions (M9: 54-55)
5. Ways to Use Local (M9: 56-63)

21.1 Administrivia

1. Stuff.

21.2 Local Definitions

1. We can define constants and helper functions locally.
2. This turns out to be useful for several reasons.
3. **Rule:** Write purpose, contract and requirements (no examples/tests) for local helper functions.

21.2.1 Locally Defined Constants

Problem from an old Assignment: Heron's formula can calculate the area of a triangle:

$$A = \sqrt{s(s-a)(s-b)(s-c)},$$

where s, which is known as the **semi-perimeter**, is:

$$s = \frac{a+b+c}{2},$$

and a, b and c are the lengths of the three sides of the triangle. Write the function `herons-formula` that consumes the values of a, b and c, and produces the value of A matching the given formula.
Remark: When we have assigned this problem in the past, we have required the student to create a helper (not local) to compute `s`. This is inefficient because it computes `s` (which is constant once `a`, `b` and `c` are chosen), four times.
Solution: Solve in DrRacket. Post your solution afterward.

21.2.2 Locally Defined Helper Functions

Problem: Re-do the `y-equals-minus-x` example from the last lecture, making the helper function local instead of global. This time call your functiony-`equals-negative-x-local`
Solution: Solve in DrRacket. Post your solution afterward.
Remarks:

1. If a helper is really only needed inside one main function, then it can be defined locally.
2. **Exercies:** Re-do the other abstract list examples from last lecture, making the helper functions local.
3. Defining lambda (next lecture) will allow us to streamline this construction even further.

21.3 Semantics of Local Definitions

From slide 52:
An expression of the form
`(local [(define x1 exp1) . . . (define xn expn)] bodyexp)` is handled as follows:
`x1` is replaced with a fresh identifier (call it `x1_0`) everywhere in `exp1` through `expn` and `bodyexp`. `x2` is replaced with `x2_0` everywhere in `exp1` through `expn` and `bodyexp`.

21.4 Nested Local Expressions

Local expressions can be nested, as needed.

- **CQ 5**

21.5 Ways to Use Local

21.5.1 Common Subexpressions

Our first example above already demonstrated this technique.

21.5.2 Improving Efficiency

Our first example above already demonstrated this technique as well.

22 Lecture 22 - `lambda`

Outline

1. Administrivia
2. Using Local for Smaller Tasks (M9: 64-65)
3. Using Local for Encapsulation (M9: 66-73)

4. `lambda` (M9: 74-81)
5. Summary of M9 (M9: 82-84)

22.1 Administrivia

1. Office Hours today in DC 3108, 1:00-1:50 PM.
2. Recall that this Friday (March 30) is Good Friday, so there will be **no labs** that day. The missed Friday schedule (including the labs) will be made up on Wednesday, April 4.
3. For a09, please stick to the instructions for Q1 - use abstract list functions, no explicit recursion is allowed!

22.2 Using Local for Smaller Tasks

Examples (review only if there is time, or come back at the end):
1. `compute-grades`
2. `my-sort`, including `insert`
3. `shorter-than-avg`

22.3 Using Local for Encapsulation

22.4 `lambda`

Remarks:
1. You may have noticed that our examples from the start of M9 are slightly awkward.
2. **Reason:** Before we learn about `lambda`, we have to define our helper functions for the abstract list functions with names, so that we can refer to them by name when we call the abstract list function.
3. If the function is only used **only once** as **input to an abstract list function**, then we can define it most efficiently using `lambda`.

Example: Re-do our earlier `all-true?` example, this time defining the combiner function for `foldr` using `lambda`. Call your new function `all-true-lambda?`
Solution: Do in DrRacket. Post your solution afterward.
- **CQ 6**

When to use `lambda`: Use lambda when the function is
- single use
- reasonably short (2-3 lines)

Exercises:

1. Re-do all earlier examples from M9, making all helper functions locally defined, using `lambda`.
2. If there is time, re-do `sublist-gt-5`, creating `sublist-gt-5-lambda` here.
   Solution: Do in DrRacket. Post your solution afterward.

22.5 Summary of M9

Remarks:

1. Refer to slides 82-84.
2. Observe that the examples, re-done using `lambda`, become much shorter.
3. Shorter code is generally more readable.
4. Mis-using `lambda` (e.g. for a function that is too complicated) can actually make your code less readable. Don't over-use `lambda`!

23 Lecture 23

Outline

1. Administrivia
2. Intro to M10 (M10: 1-2)
3. General Arithmetic Expressions (M10: 3-19)

23.1 Administrivia

1. We'll do our course evaluations during the last lecture on April 3.

23.2 Intro to M10

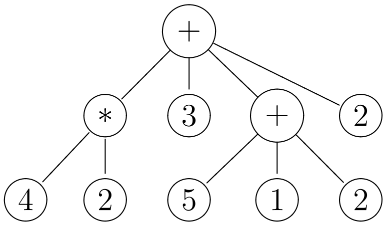
Trees Before M10

1. All **binary** (i.e. each internal node has at most two children).
2. In trees representing arithmetic expressions, each internal node has **exactly** two children (because each arithmetic operation takes exactly two arguments).
3. In M10, we generalize to trees where internal nodes can have **any number** of children.

23.3 General Arithmetic Expressions

1. The arithmetic operations that take more than two operations are `+` and `*`.
2. Hence we will restrict the operations in our trees representing general arithmetic expressions to `+` and `*`.
3. Review the example of mutually recursive data definitions on slide 6!
4. Here is a diagram of the last tree defined on slide 7:

```
(make-ainode '+ (list (make-ainode '* (list 4 2)) 3
                (make-ainode '+ (list 5 1 2)) 2))
```



- **CQ 1**

Slides 9 and 10

1. The purpose of the `eval` function is to evaluate the given arithmetic expression and produce the number that results from this evaluation.
2. The `eval` function depends on the `apply` helper function, which we develop on the next slide.
3. The `apply` function is meant to apply the specified function to the specified list, and return the correct arithmetic result.
4. Note that `apply` needs an identity element is needed and may not be so straightforward for all operations.
    (a) For multiplication, the identity element is 1.
    (b) For addition, the identity element is 0.
5. The functions `eval` and `apply` are two recursive functions that refer to one another. In other words they are our first example of functions having **mutual recursion** (as mentioned on slide 6, referring to the corresponding data definitions).

24 Lecture 24

Outline

1. Administrivia
2. General Arithmetic Expressions (M10: 3-19)
3. Leaf Labelled Trees (M10: 20-31)
4. Summary of M10 (M10: 32-33)

24.1 Administrivia

1. Office Hours today in DC 3108, 1:00-1:50 PM.
2. **Reminder:** Both Lab 12 and a09 will be due **tomorrow night at 11:59 PM**.

24.2 General Arithmetic Expressions

Start of Condensed Trace on slides 12-15

```
(eval (make-ainode '+ (list (make-ainode '* (list 3 4))
                        (make-ainode '* (list 2 5)))))
⇒ (apply '+ (list (make-ainode '* (list 3 4))
                  (make-ainode '* (list 2 5))))
⇒ (+ (eval (make-ainode '* (list 3 4)))
     (apply '+ (list (make-ainode '* (list 2 5)))))
⇒ (+ (apply '* (list 3 4))
     (apply '+ (list (make-ainode '* (list 2 5)))))
⇒ (+ (* (eval 3) (apply '* (list 4)))
     (apply '+ (list (make-ainode '* (list 2 5)))))
⇒ ...
```

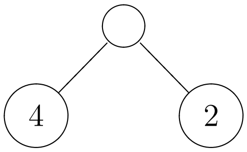
Exercise: Finish the trace yourself. Get the answer 22.
Review the alternate data definition on slides 17-19.

- **CQ 2**

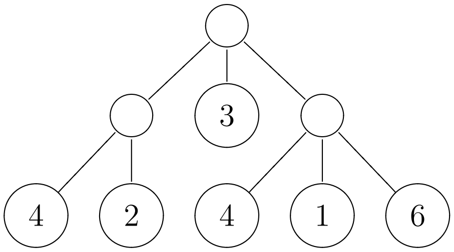
24.3 Leaf Labelled Trees

Examples from slide 24

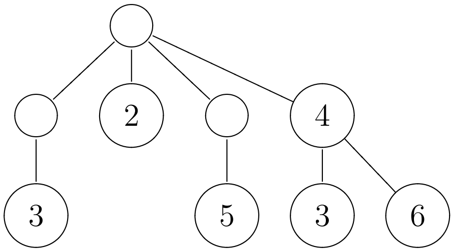
1. `(list 4 2)`



2. `(list (list 4 2) 3 (list 4 1 6))`



3. `(list (list 3) 2 (list 5) (list 4 (list 3 6)))`

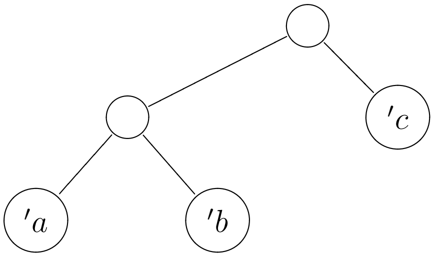


- **CQ 3**

Data Definition for LLTs (slide 25)

```
;; A leaf-labelled tree (LLT) is one of the following
;; * empty
;; * (cons Num LLT)
;; * (cons LLT LLT) where first LLT is nonempty.
```

slide 28 - input to `count-leaves`
`(list (list 'a 'b) 'c)`



Start of Condensed Trace on slides 28

```
(count-leaves (list (list 'a 'b) 'c))s
```

```
⇒ (+ (count-leaves (list 'a 'b)) (count-leaves (list 'c)))
⇒ (+ (+ 1 (count-leaves (list 'b))) (count-leaves (list 'c)))
⇒ (+ (+ 1 (+ 1 (count-leaves (list)))) (count-leaves (list 'c)))
⇒ (+ (* (eval 3) (apply '* (list 4)))
⇒ ...
```

Exercise: Finish the trace yourself. Get the answer `3`.

- **CQ 4**

24.4 Summary of M10

- **CQ 5**

Index