

CS 145 Lecture Summary

2019: Gian Cordana Sanjaya

2020: Rivers Chen, Benjamin Wong

CS 145 Instructional Support Assistant

Fall 2020

Notes

1. **For future ISAs: Please do not share this PDF outside the CS 145 course.**
2. In Racket interactions shown in this PDF, at the beginning of each line:
 - `>` means user-written code.
 - `<` means a produced result.
 - A boxed expression means a user input through functions such as `read` and `read-char`.
 - A purple-coloured expression means output through functions such as `display`.

Contents

1	(September 8, 2020)	6
1.1	Introduction	6
1.2	Racket	6
1.3	Assignment 0	8
2	(September 9, 2020)	9
3	(September 10, 2020)	10
4	(September 14, 2020)	11
4.1	Memory Analysis of <code>sumto</code>	11
4.2	Tail Recursion	11
4.3	Comparing different versions of <code>sumto</code>	13
5	(September 15, 2020)	14
6	(September 16, 2020)	15
7	(September 20, 2020)	17
7.1	Conditional Expression	17
7.2	Abstract Data Type	17
7.3	Implement Bunch	18
8	(September 21, 2020)	20
8.1	A Concrete Data Structure	20
8.2	Revised Bunches	20
8.3	Trees	21
8.3.1	Anatomy of a Tree	22
8.3.2	Counting Trees	22
8.3.3	Extremal Trees	23
9	(September 23, 2020)	24
9.1	Examples	24
9.2	Height Decorated Trees	25
10	(September 27, 2020)	26
10.1	Introduction to Binary Search Tree	26
10.2	BST Specific Algorithms	26
11	(September 29, 2020)	28
12	(October 1, 2020)	30
12.1	Lists	30

13 (October 5, 2020)	32
13.1 Built-in Functions For Lists	32
13.2 Reverse Lists	33
13.3 Ordered Lists	33
13.4 List vs. (Decorated) Trees	34
13.5 More Racket feature: <code>random</code> and <code>racket/base</code>	36
14 (October 6, 2020)	37
14.1 Motivation	37
14.2 Lambda Expression	37
14.3 Abstract List Functions	38
15 (October 10, 2020)	40
15.1 Modules and <code>time</code>	40
15.2 Order Notation	41
16 (October 19, 2020)	43
16.1 Formal Big-O	43
16.2 Big-O proofs	43
16.3 Big-O Identities	43
16.4 Big-Omega and Big-Theta	44
17 (October 20, 2020)	45
17.1 Building More Trees	45
17.2 Recursion on Trees	45
17.3 Tail-Recursive/Singly-Recursive Sum Function	46
18 (October 20, 2020)	47
18.1 Introduction to Information Hiding	47
18.2 Example: BST	47
19 (October 26, 2020)	49
19.1 Total Order	49
19.2 Small Note on <code>eq?</code>	50
20 (October 27, 2020)	52
21 (October 27, 2020)	53
21.1 Insertion Sort	53
21.2 Selection Sort	53
21.3 BST sort	54
21.4 Merge Sort	54
21.5 Quick Sort	55

22 (November 2, 2020)	56
22.1 Generating Functions	56
22.2 The <code>generate</code> Function	57
23 (November 2, 2020)	58
23.1 Infinite Sequences	58
23.2 The Stream ADT	58
24 (November 5, 2020)	60
24.1 Deferred Evaluation	60
24.2 Input/Output	61
24.3 Aside: Models of Computation	61
24.4 Lazy Racket	62
25 (November 10, 2020)	63
25.1 Lambda Calculus	63
25.1.1 notation	63
25.1.2 rules	63
25.1.3 α - β reduction	64
25.2 Fully Reduce λ -expression	65
25.3 Relation to (Lazy) Racket	66
26 (November 11, 2020)	67
26.1 Translating to λ -calculus Expressions	67
26.2 Building ADT with λ -calculus	67
26.2.1 Booleans	67
26.2.2 Cons	68
26.2.3 Natural Numbers	68
26.3 Non-recursive <code>define</code> with λ -calculus	68
27 (November 11, 2020)	68
27.1 Recursion	68
27.2 Natural Numbers	69
28 (November 17, 2020)	70
28.1 Reading Characters in Racket: <code>read-char</code>	70
28.2 Reading various Racket types: <code>read</code>	71
28.3 Outputting Racket values: <code>display</code>	72
28.4 <code>IOStream.rkt</code>	73
29 (November 19, 2020)	75
29.1 Assignment 11 Version of <code>Gen</code>	75
29.1.1 <code>Gen</code> usage	75
29.1.2 Other functions and variables in <code>Gen.rkt</code>	76

30 (November 19, 2020)	78
30.1 Assignment 11 Part (b): Rock Paper Scissors	78
31 (November 23, 2020)	79
31.1 Introduction to RAM	79
31.2 Build a RAM Machine	79
31.2.1 Build RAM in Racket	79
31.2.2 Build RAM Physically	80
31.3 Assignment 12	80
31.4 Representing Data in RAM	81
32 (November 25, 2020)	82
32.1 Ram Operations	82
32.2 Lists in RAM	83
32.3 Linked-lists	83
33 (November 25, 2020)	85
34 (November 26, 2020)	86
35 (December 3, 2020)	88
36 (November 28, 2020)	90

Lecture 1 (September 8, 2020)

The lecture begins with some introduction to the course. It uses a programming language called Racket, and it starts in teaching dialect level.

1.1 Introduction

- You can enroll in CS 146 if you get good marks in CS 145, similar to enrolling in CS 136 if you get good marks in CS 135.
- You can also enroll in CS 146 with instructor consent if you get a high mark (over 80%) in CS 135.
- The due date for dropping down from CS 145 to CS 135 or CS 115: October 21, 2020.
- Grading for this term: assignments, worth 60%, 2 (assignment like) midterms, each worth 10%, and final exam, worth 20%.

This course is (obviously) assignment driven.

- This course does not require prior programming experience. You can find some brief guidelines on Racket in the book HtDP (How to Design Programs).
- The main site of communication is Piazza
- Contact information can be found on the course homepage.

1.2 Racket

- **Racket** is a dialect of Scheme, which is a dialect of Lisp, which was originally created to develop artificial intelligence. It is particularly well suited for creating data structures, etc.
- There are several things you can do on Racket. See the following:
 - Click "Run" to run a program.
 - Type in 42 in Racket and run it. It will compute and produce 42.
 - Syntax for addition in Racket: (+ 42 10) instead of (42 + 10) or 42 + 10 and it will produce 52. Similar with * / -.
 - You can actually type in multiple expressions at once, for example both (+ 42 10) and (* 1 2 3). It will evaluate to 52 and 6. Try typing in all the expressions you can find below, on the last page of the section.
 - Other data types in Racket: symbols, start with '. For example: 'hello, 'lol. Try the following:

```
(= 'hello 'hello)
```

It does not work. Use `symbol=?` instead, i.e., `(symbol=? 'hello 'hello)`.

- Other important example:

```
(if (< 1 2) (+ 3 4) (+ 5 6))
```

Notice that when ran in DrRacket the block `(+ 5 6)` goes black.

- Question: Is Racket whitespace-sensitive?

Answer: Yes

- Racket has *stepper*, which shows how Racket evaluates expressions. For example, try stepping through

```
(+ (+ 1 2) (+ 3 (+ 5 6))).
```

- You can also define functions in Racket. For example, try adding the following line:

```
(define (dubble x) (+ x x))
```

Then, you can also call them. For example, `(dubble 10)` and `(dubble (dubble 10))`.

- Assignments are on the course website:

<https://www.student.cs.uwaterloo.ca/~cs145/>

Students will submit solutions to the Marmoset system.

- This course is balanced between theory and practice. Syntax is a tiny part of the course. The big part will be correctness proof, discovering algorithms, and discussing strategies.
- For now, use the Beginning Student dialect.

Some codes you can try on Racket

```

(- 1 2 3)
(* 3 2 ) ;; extra space
(* 2424749748 2456532 123598210 1000001
  ) ;; new line before closing bracket
(/ 6 3)
(/ 6 5) ;; fraction
(/ 1 3) ;; non-terminating decimal
(expt 2 3)
(expt 2 .5) ;; non-integer power
(sqrt 2) ;; equivalent of above
(sqrt -1) ;; complex numbers
(< 20 30)
(< 1 2 3 4 5 6 7)
(<= 1 2 3 4 5 6 7)
(= 1 1 1 1 1)
(= 1 2 3 4 5)
(number? 3)
(number? (+ 1 2))
(exact? (/ 1 2))
(real? (/ 1 2))
(complex? (/ 1 2))
(exact? (sqrt 2))
(exact? (sqrt 4))
(expt (sqrt 2) 2) ;; not exact!

```

1.3 Assignment 0

Assignment 0 is demonstrated in the video.

Note that the release token is regenerated every 12 hours for each assignment entry.

Lecture 2 (September 9, 2020)

This lecture introduces more data types: rational numbers, irrational numbers, symbols, strings, and Boolean values; and also other features of Racket.

Also see the previous lecture summary.

We have seen .

- The beginning student language by default represents all numbers as decimals. This behaviour can be changed in Showdetail -> Fraction style in the Choose Language Panel (However, it can cause trouble with Marmoset submissions).
- Once a computation involves an inexact number, all following results will be inexact. In other words, once it is impossible to “recover” the exact form of the number.

Example: `(sqr (sqrt -2))`

- There is a **stop** button at the top right corner of DrRacket. (Ctrl-B)
- For Boolean values, `#t`, `#true`, and `true` are all equivalent. Similar for false.
- A special form called `if` is introduced. The format is

`(if predicate then-expression else-expression)`

`and` and `or` are also special forms.

- In DrRacket, after you run a program, the parts of the code that are never reached will be highlighted. This can be observed with, for example, `(if (< 2 3) 1 (+ 2 3))`
- It's recommended to checkout <https://docs.racket-lang.org/> The Racket Documentation for reference.

Lecture 3 (September 10, 2020)

This lecture introduces recursion in Racket.

Joke:

Definition: Recursion — See *Recursion*.

Broadly speaking, recursion is to use itself. More specifically, a program applies itself to a “smaller” problem is called a recursive function.

- Example code:

```
(define (sumto n) (if (= n 1) 1 (+ n (sumto (- n 1)))))
```

- Try stepping through the following function calls: `(sumto 1)`, `(sumto 10)`, `(sumto 0)`
- Note that the evaluation process of `(sumto 1)` does not use itself, this (argument 1) is called the trivial case. There always has to be a trivial case of a recursion.
- Another example: Fibonacci Number

```
(define (fib n)
  (if (= n 0) 0
      (if (= n 1) 1
          (+ (fib (- n 1)) (fib (- n 2))))))
```

Note that it calls itself twice.

- If you consider the evaluation process with stepper, you will find the above definition of `fib` does a lot of work!

Lecture 4 (September 14, 2020)

4.1 Memory Analysis of `sumto`

This lecture introduces an alternative version of `sumto`

- Recall the previous definition of `sumto`:

```
(define (sumto n)
  (if (= n 1) 1
      (+ (sumto (- n 1)) n)))
```

(The order of the two arguments of the last `+` are different, but since addition is commutative here the programs should be equivalent with the version given in the last lecture summary)

- Try to compute `(sumto 10000000)`. The program will stop because the memory limit is exceeded, assuming that you have not increased the memory limit.
- We want to know the number of steps (through stepper), and the size of the derivation, which is roughly the biggest intermediate expression.
- We can roughly view the stepper process as a series of substitution. It takes the original expression, and substitute a new expression, until it gets the final answer.
- Racket repeatedly takes an expression, does a calculation, and substitute the new expression into the original one.
- The amount of computer memory needed is roughly the largest (by the maximum number of opening parenthesis) intermediate expression.
- Try using stepper to calculate a number of different applications of `sumto`, you would find that the number of opening parenthesis increases as the argument increases.
- In general the size of the largest intermediate step is given by:

```
size(1) = 5
size(n>1) = size(n-1) + 1
```

- In fact, `size(n) = n + 4`

4.2 Tail Recursion

- We propose another function

```
(define (sumbetween a b acc)
  (if (> a b) acc
      (sumbetween (add1 a) b (+ acc a))))
```

The `acc` stands for accumulator.

- Then we define

```
(define (sumto n)
  (sumbetween 1 n 0))
```

- We find that even `(sumto 100000000)` can be computed without exceeding memory limit.
- Using the stepper, we find that the maximum size of intermediate expression does not grow.
- This is called Tail Recursion.
- In tail recursion, whenever you enter a recursive application of the function, it is always the last thing you do.
- Previously, when we enter an recursive case, we calculate `(sumto (- n 1))`, and add `n` to it.
- In this example, `sumbetween` is called a *helper function*, and actually solves a more general problem than `sumto`.
- The key to tail recursion is that we do not have to save or process the value of the recursive application of the function. In this case, we save pass the value through accumulator so that no further manipulation of the returned value is required.

4.3 Comparing different versions of `sumto`

We can even define a new `sumto`:

```
(define (sumto n) (/ (* n (+ n 1)) 2))
```

Here is a comparison between all versions of `sumto` we have so far:

	sumto	sumbetween	closed form
result	$result[1] = 1$ $result[n > 1]$ $= result[n - 1] + n$	$result[1] = 1$ $result[n > 1]$ $= result[n - 1] + n$	$\frac{n(n+1)}{2}$
step	$5(n - 1) + 3$	$5(n - 1) + 9$	4
size	$n + 4$	5	3

Lecture 5 (September 15, 2020)

This lecture tries to analyze a mysterious function (`crunch x`)

```
(define (crunch x)
  (if (< x 10) 1
      (add1 (crunch (/ x 10)))))
```

- (`crunch n`) seems to give the number of digits of n in decimal representation.
- We then state two facts (in an informal notation):
 - For $n \in \{0, 1\}$, $(\text{crunch } n) = 1$.
 - For $n = 10^k$,
($k \geq 1$), $(\text{crunch } n) = 1 + (\text{crunch } n/10) = k + 1$.
- Using the two facts we want to prove the value of (`crunch n`) in the general case, when n is a (non-negative) rational number.
 - For $0 \leq n < 10$, $(\text{crunch } n) = 1$
 - For $10 \leq n < 10^2$, $(\text{crunch } n) = 2 = k + 1$
 - For $10^k \leq n < 10^{k+1}$, ($k > 1$),
 $(\text{crunch } n) = 1 + (\text{crunch } n/10) = k + 1$
- Why? The base case is obvious. When $10^k \leq n < 10^{k+1}$ for $k > 1$, we know that if $n \in [10^k, 10^{k+1})$, then $10^{k-1} \leq \frac{n}{10} < 10^k$, which is in the range for which we already know the answer of.
- We then know fully about (`crunch x`): given a non-negative rational number, it gives the number of digits of its integer part in decimal representation.
- It is important to write such descriptions for each function in one's code, for the sake of sanity.

Lecture 6 (September 16, 2020)

This lecture introduces basic data structure in Racket.

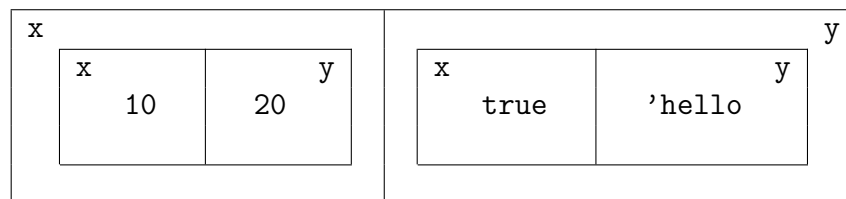
- So far, all functions we have discussed calculates only one value. In a lot of cases, we want to be able to compute two values at once. For real numbers, we can switch into complex numbers to do it. But how can we do that in a more general way?
- In Beginning Student, we are provided the `make-posn` function. It is called a constructor which is a special kind of function, it does nothing besides storing its arguments and return a data structure. Example: `(make-posn 10 20)`.
- We could define a variable to contain a structure object. Example: `(define wilma (make-posn 10 20))`. Then, `wilma` will evaluate to `(make-posn 10 20)`.
- The true power of `make-posn` is that it can contains data structures of it self: consider the following definitions:

```
(define fred (make-posn true 'hello))
(define betty (make-posn wilma fred))
```

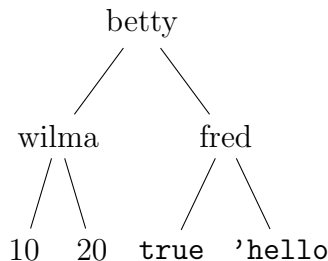
Then, evaluating `betty` produces

```
(make-posn (make-posn 10 20)
            (make-posn true 'hello))
```

- We could also consider the box representation of `posn` to understand it more easily.



- Writing in the box diagram is very cumbersome, we can use a much abbreviated form:



- Other available functions for handling `posn`:
 - `posn?` (Predicate function). `(posn? wilma)` produces `true`.

- Field selectors: `posn-x` and `posn-y`. `(posn-x wilma)` produces 10.
- We can also define our own data structures:

```
(define-struct triple (fred wilma pebbles))
```

Then Racket automatically provides us with `make-triple`, `triple-fred`, `triple-wilma`, `triple-pebbles`, and `triple?`

Lecture 7 (September 20, 2020)

This lecture introduces another Racket syntax called `cond`, as well as an *abstract data type* (ADT) called *bunch*.

7.1 Conditional Expression

- `cond` stands for *conditional* expression.
- The basic form of `cond` is

```
(cond
  [p1 e1]
  [p2 e2]
  :
  [pn en])
```

- A `cond` consists of a sequence of boolean-expression-Racket-expression pairs. `cond` evaluates each of the boolean expressions in order until it finds one that evaluates to `true`, while discarding all the expressions associated to those boolean expressions that are false. Racket then returns the value of the corresponding expression.
- If none of the boolean expressions is true, then Racket will produce an error. Therefore, `pn` is often replaced by `true`.
- `cond` is a generalized version of `if`, it just makes a series of conditional pairs much simpler to write.

7.2 Abstract Data Type

- An ADT is defined based on the available operations of the data structure. The actual implementation does not matter. We can use Racket to implement an ADT, as well as with many other programming languages.
- A bunch is basically a *multiset* of Racket values.
- A multiset is basically a set that allows duplicate elements.
- We define some operations for a bunch (using a Racket-like syntax)
 - `emptyb`: an empty bunch.
 - `(singb e)`: creates a *singleton* bunch for a given element.
 - `(joinb b1 b2)`: join two bunches
 - `(countb e b)`: count the number of instances for a given element.
 - `(empty? b)`: determine whether a given bunch is empty. It is only true if `b` is the empty bunch.

- `(remb e b)`: remove all instances of the given element in the bunch.
- `(elemb b)`: return an *arbitrary* element from the bunch. Undefined if the bunch is empty.
- Note that we have not discussed how are these operations implemented, nor what exactly a bunch is in a Racket program. These choices are in fact arbitrary when defining an ADT.
- Using these operations we can form other functions over bunches. For example, a function that counts the number of distinct elements in a bunch: by removing an arbitrary element one at a time with `elemb` and `remb`.
- Also note that `remb` returns a new bunch, with no instance of `e` in it.

Example in Racket:

- Suppose we do:


```
(define x1 emptyb)
(define x2 (singb 'fred))
(define x3 (singb 'wilma))
(define x4 (joinb x2 x3))
```
- Then we should expect


```
> (emptyb? x1)
< true
> (countb 'wilma (joinb x3 x3))
< 2
> (elemb x4)
< 'fred ;; or it could be 'wilma
> (elemb (remb 'fred x4))
< 'wilma
```
- We can use these operations to find out what a bunch contains by repeatedly using `elemb`, `remb`, and `countb`.

7.3 Implement Bunch

An example implementation looks like this

```
(define-struct nothingb ())
(define-struct pairb (a b))
(define emptyb (make-nothingb))
(define (singb v) v)
(define (joinb a b) (make-pairb a b))
(define (countb e b)
  (cond
    [(nothingb? b) 0]
```

```

      [(pairb? b) (+ (countb e (pairb-a b))
                     (countb e (pairb-b b)))]
      [(equal? e b) 1]
      [true 0]))
(define (remb e b)
  (cond
    [(nothingb? b) b]
    [(pairb? b) (joinb (remb e (pairb-a b))
                       (remb e (pairb-b b)))]
    [equal? e b) emptyb]
    [true b]))
(define (elem b)
  (cond
    [(pairb? b)
     (if (emptyb? (pairb-a b))
         (elemb (pairb-b b))
         (elemb (pairb-a b)))]
    [true b]))
(define (emptyb? b)
  (cond
    [(nothingb? b) true]
    [(pairb? b) (and (emptyb? (pairb-a b))
                     (emptyb? (pairb-b b)))]
    [true false]))

```

- Note the use of `cond` in `countb`. The third case represents the singleton case.
- For `elemb` the `true` case is for both singleton and `emptyb`, even though the function is undefined when given an empty bunch.
- Note that our implementation forbids bunches containing other bunches.
- Another issue with our implementation is that when removing elements in a bunch it leaves us an empty structure consists of `pairbs`.

Lecture 8 (September 21, 2020)

Recall the bunch ADT from last lecture. This ADT is defined by its operations as described in lecture 7.

8.1 A Concrete Data Structure

After discussing the bunch ADT, we can begin to formulate one possible concrete implementation. We define bunches as follows.

Definition: a bunch is implemented as one of three things:

1. An instance of `(make-nothingb)`.
2. A value other than an instance of `(make-nothingb)` or `(make-pairb...)`.
3. An instance of `(make-pair a b)` where `a` and `b` are implementations of bunches.

We can compare the abstract “pen and paper” bunch with the concrete and abbreviated bunch notations.

Abstract	Concrete	Abbreviated
\square	<code>(make-nothingb)</code>	\circ
$\boxed{42}$	42	42
$\boxed{25} \boxed{50}$	<code>(make-pair 25 50)</code>	<code>[.25 [.50]]</code>

Warning! This implementation admits multiple representations of the same bunch. For example:

$$[.50 [.]] \qquad 50 \qquad [. [.50]]$$

All of the above represent the singleton bunch containing 50.

This ambiguity creates more cases for us to deal with when we program. For example, the implementation of `emptyb?` requires recursion despite accomplishing a relatively simple task. We would like to minimize this ambiguity by introducing a new implementation of bunches that is less flexible.

8.2 Revised Bunches

Consider the following *new* concrete implementation of bunches.

A bunch is one of:

- A `(make-nothingb)`.

- A Racket value other than a `(make-nothingb)` or a `(make-pairb...)`.
- A `(make-pairb a b)` where `a` and `b` are non-empty bunches.

The key change here is that empty bunches are not permitted inside of `(make-pairb ...)` calls.

To see how this changes our common functions, we'll look at each function and the different cases with which we must deal.

- `(emptyb? b)`
 - $b = \circ \implies \text{true}$
 - $b \neq \circ \implies \text{false}$
- `(elemb b)`
 - $b = \circ \implies \text{undefined}$
 - $b = [.x [.y]] \implies (\text{elemb } x)$
 - $b \neq \circ \text{ and } b \neq [.x [.y]] \implies b$
- `(joinb a b)`
 - $a = \circ \implies b$
 - $b = \circ \implies a$
 - $a \neq \circ \text{ and } b \neq \circ \implies [.x [.y]]$

As you can see, the implementation of `emptyb?` is much simpler.

8.3 Trees

In this section, we turn our attention to trees. Our bunches from lectures 7 and 8 are an example of a tree. Any data structure built using structs is a tree. Trees are often restricted in form and obey specific rules.

An *undecorated binary tree* is either:

1. An empty represented by `'()`. Or,
2. A node containing two trees. The two child trees are often called the left and right subtrees.

Tree Concept	Abbreviation	Possible Racket Implementation
empty	ϵ	<code>empty</code> or <code>'()</code>
$[\epsilon [\epsilon]]$	\circ	<code>(make-node empty empty)</code>
$[.x [.y]]$	$[.x [.y]]$	<code>(make-node x y)</code>

8.3.1 Anatomy of a Tree

Now we develop a language with which to talk about trees.

- A *leaf* is a node with no subtrees.
- The *root* of a is a node that is not a subtree. Each non-empty tree has exactly one root.
- A *path* is a sequence of nodes which are connected by adjacency.
- The *length* of a path is the number of nodes in that path.
- The *depth* of a node is the length of the path which has endpoints at the root and the node itself.
- The *height* of a tree is zero for an empty tree, and the maximum length of a path beginning at the root. Note that the empty tree has height zero.
- The *size* of a tree is the number of nodes in that tree.

We can also think of size and height recursively. Suppose that t is a tree which is either empty or which has left subtree a and right subtree b .

The **size** of a tree t is:

1. 0 if t is the empty tree. Or,
2. $1 + \text{size}(a) + \text{size}(b)$ when t is non-empty.

The **height** of a tree is:

1. 0 if t is the empty tree. Or,
2. $1 + \max\{\text{height}(a), \text{height}(b)\}$ when t is non-empty.

8.3.2 Counting Trees

We can look at the possible trees by classifying them by height:

height	trees	number
0	ϵ	1
1	\circ	1
2	$/ \ \backslash \ \wedge$	3
\vdots	\vdots	\vdots

We could build a similar table for size rather than height.

8.3.3 Extremal Trees

We want to determine what the largest and smallest (by height) trees are for a given size.

Maximal trees are trees with maximum height. They are trees in which all of the nodes lie on a single path from the root. It is not surprising that the maximum height of a tree with n nodes is n .

Minimal trees are trees which have the smallest height for a given number of nodes. Minimal trees require more subtlety than maximal trees. However, we will see that creating minimal trees is worth the effort. For a size n , if one can find an h such that $2^{h-1} \leq n < 2^h$ holds, then h is the height of any minimal tree with n nodes.

Full trees are a special kind of minimal tree. A full tree of height h has the maximum number of nodes among all trees of its height. A full tree is a tree in which every node has zero or two subtrees, and the path length from the root to each node with empty subtrees is the same. A full tree with height h necessarily has $2^h - 1$ nodes.

Lecture 9 (September 23, 2020)

Let us consider *decorated binary trees*. We build on our theory of undecorated binary trees by adding one or more values to each node. Later on in the course, we will use these new trees to talk about binary search trees, heaps, and tries.

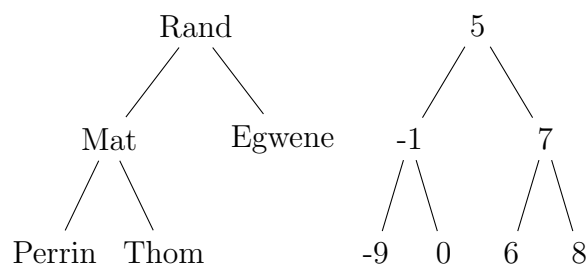
9.1 Examples

We can decorate the nodes with their size, height, or both:

<i>Decorated by:</i>	Size	Height	(Size, Height)
<i>Example:</i>	<pre> 5 / \ 1 3 / \ 1 1 </pre>	<pre> 3 / \ 1 2 / \ 1 1 </pre>	<pre> (5,3) / \ (1,1) (3,2) / \ (1,1) (1,1) </pre>

These examples are all amenable to simple Racket implementations via `define-struct`. For example: `(define-struct node (size height left right))`, `(define-struct node (left size right))`, or `(define-struct node (left right height))`.

The above examples use the values added to store information about the tree. We can also use decorated binary trees to store other elements in a tree structure.



9.2 Height Decorated Trees

If we need to access the height of our trees often, it may be useful to turn `(height t)` into a constant time function by modifying our data structure. We accomplish this with the ADT definition of *Height Decorated Trees*. The operations for HDTs are:

- `(hdt-height t)` Calculates the height of an HDT t in a constant number of steps.
- `(hdt-left t)` Returns the left subtree of an HDT t .
- `(hdt-right t)` Returns the right subtree of an HDT t .
- `empty` The empty HDT
- `(empty? t)` Returns `true` if and only if t is the empty HDT
- `(make-hdt a b)` Creates an HDT with left and right subtrees which are the HDTs a and b .

One possible concrete definition of this ADT can be made in Racket using `(define-struct node (l r h))`.

Lecture 10 (September 27, 2020)

10.1 Introduction to Binary Search Tree

The lecture introduces Binary Search Tree (BST), a special kind of tree.

- A binary search tree is a decorated tree that represents an (ordered) set.
- More specifically, a set is also an ADT, and a binary search tree can be used to implement an (ordered) set.
- A BST is either empty or a node, where its left and right are subtrees, and a key k , satisfying the following property:
 - The left subtree is a BST with every key less than k ,
 - The right subtree is a BST with every key greater than k .
- Because of the two above property, each key in the tree is unique.
- We sometimes call the properties of BST the *invariant*.
- We can easily come with an algorithm that inserts an element to a BST Recursively go to the node depending on whether the current key is less than or greater than the element we are inserting, until we find an empty spot to create a new node with the element. Note that we have to re-create the node that we “walked along”. An important observation is that the number of nodes we have to recreate is related to the height of the tree.

Given input x and tree t

- If t is empty, the result is just x
- If t is a node with key y with left and right subtrees A and B , then
 - * If $x < y$, then the result is a new tree with the same key y and right subtree B , but with A replaced by A' , which is a BST that contains x .
 - * If $x > y$, then the result is a new tree with the same key y and left subtree A , but with B replaced by B' , which is a BST that contains x .
 - * If $x = y$, then the result is just t .

10.2 BST Specific Algorithms

- The following functions whether x is an element of t , but it has to visit every node in the tree. We can have a more efficient algorithm for BSTs.

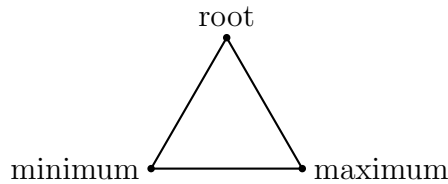
```
(define (elem? x t)
  (cond
    [(empty? t) false]
    [(= x (node-key t)) true]
    [true (or (elem? x (node-left t)) (elem? x (node-right t)))])
```

- We define a function (`elemb? x t`) that takes an element x and tree t .
 - If t is `empty`, then the result is `false`.
 - If t is a node with key y with left and right subtrees A and B , then
 - * If $x < y$, then the result is (`elemb? x A`)
 - * If $x > y$, then the result is (`elemb? x B`)
 - * If $x = y$, then the result is `true`.
- `elemb?` takes much less step to compute, because at each step it knows which subtree the element (potentially) lies in. A BST with height 23 can store 2^{24} nodes, `elem?` will have to check each of these nodes, while `elemb?` only takes about 23 recursive calls.

Lecture 11 (September 29, 2020)

This lecture continues BST specific algorithms.

- Sometimes we draw a tree as a triangle to signify the minimum and maximum elements.



- In the following functions, t is a BST of the form



where A and B are the left and right subtree of t with its key y .

- A function (`minb t`) that is the minimum element of a BST.
 - If t is empty, then there is no minimum element, in which case the function is undefined.
 - If t is not empty, then all nodes in the left subtree is less than the root, so we can look for the minimum element in the left subtree.
 - If t is not empty but the left subtree is empty. Then the root should have the minimum element.
- A function (`deleteminb t`) that is the tree that is equivalent to t except that the minimum element in t is missing.

case	result
$t = \epsilon$	undefined
$A = \epsilon$	y
$A \neq \epsilon, B \neq \epsilon$	$(deleteminb A)$

- (`delb x t`) that is the tree equivalent to t but with the element x missing. Here we denote the maximum element of A by α , and the minimum element of B by b (if they exist).

case	result
$t = \epsilon$	ϵ
$x = y, A = \epsilon$	B
$x = y, B = \epsilon$	A
$x = y, A \neq \epsilon, B \neq \epsilon$	Promote b or α to the root of t
$x \neq y, x < y$	$(make-node y (delb x A) B)$
$x \neq y, x > y$	$(make-node y A (delb x B))$

- We can use the `random` function to randomly choose which subtree to promote to prevent the trees from growing taller when repeatedly delete and insert elements from a tree.

Other functions:

`(bst? b)`, the predicate function for BST. Needs a helper function `(bst-range? t a b)` that checks if `t` is a BST with all its keys between `a` and `b`.

```
(define (bst-range? t a b)
  (cond
    [(empty? t) true]
    [(and (< a (node-key t)) (> b (node-key t)))
     (and (bst-range? (node-left t) a (node-key t))
           (bst-range? (node-right t) (node-key t) b))]
    [true false]))
(define (bst? b) (bst-range? b -inf.0 +inf.0))
```

Lecture 12 (October 1, 2020)

Introduction to Lists

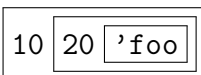
- Remember that `empty`, `null` and `'()` represent the same thing.

12.1 Lists

- Lists can be seen as a special case of max-height trees.
- We can consider `(define-struct cons (car cdr))` and see a list as either empty or `cons`, where its `car` is any element and `cdr` is a list.
- It is not quite like that in actual, because the “constructor” is actually `(cons a b)`, the “field selectors” are `(car lst)` and `(cdr lst)`, and the “predicate” is `list?` (or `pair?`, which does not work in Beginning Student).
- To spell it out, A list is one of

- `empty`
- a pair `(cons A B)` where
 - * A can be *anything*.
 - * B is a list.

- In real Racket, `(cons 'foo 22)` works. It doesn't work in Student Language though. In Student Language, the second element must be a list.
- We could also consider a box diagram for list. For example, the above one is represented

by: 

- We can define functions that generate lists for us

```
;; the first n non-negative integers in reverse order
(define (firsts n)
  (if (zero? n) empty
      (cons (sub1 n) (firsts (sub1 n)))))
```

- Beginning Student prints long lists in an unreadable way. We can use Beginning Student *with List Abbreviations*. `(list 10 20 'foo)` is the same as `(cons 10 (cons 20 (cons 'foo '())))`.
- There also also two aliases `first` and `rest` for `car` and `cdr` respectively.
- Typing codes such as `(car (cdr ...))` becomes annoying over time. We can use `(cadr ...)`, `(caddr...)`, etc.
- `(second lst)` and `(third lst)` are two aliases of the two previous functions, respectively.

- You can use any combination of **a** and **d** in the name until you do not know what you are doing.

Ex. (**cadadr** ...) gives you the second element of the second element of the list.
(**cdadr** ...) gives you the cdr of the second element of the list. These functions only work if the second element of the list is itself a list.

- Other predicate functions on lists: **empty?**, **list?**, **null?**, **cons?**.

Lecture 13 (October 5, 2020)

This lecture continues discussion about list.

13.1 Built-in Functions For Lists

- Other built-in functions on lists:

- `(length lst)`, the length of `lst`. It is defined in Racket, but we can define it as the following:

```
(define (Length l)
  (if (empty? l) 0 (add1 (Length (cdr l)))))
```

- `(member x lst)` returns true if `x` is an element of `lst`, false otherwise.

```
(define (Member x lst)
  (cond
    [(empty? lst) false]
    [(equal? x (car lst)) true]
    [true (Member x (cdr lst))]))
```

- Two more functions over lists.

```
(define (sum x)
  (if (empty? x) 0 (+ (car x) (sum (cdr x)))))
(define (double x)
  (if (empty? x) 0
      (cons (* 2 (car x)) (double (cdr x)))))
```

- `(append a b)`, appends `a` to the front of `b`.

```
(define (Append a b)
  (if (empty? a) b (cons (car a) (Append (cdr a) b))))
```

- Notice that the running time of `append` is proportional to the length of `a`, the length of `b` is *irrelevant*.
- It is very easy to add things at the beginning of the list, but to add anything to the end of the list, we have to first go through the whole list.

Other list functions:

- `(list-ref l n)`, the `n`th element of `l`. Here, indexing starts at 0.

```
(define (List-ref l n)
  (if (zero? n) (car l) (List-ref (cdr l) (sub1 n))))
```

- `list*`, a list constructor function. `(list* 10 20 (list 40 50))` is equivalent to `(list 10 20 40 50)`.

13.2 Reverse Lists

A more in-depth discussion of `reverse`:

- Consider a implementation of `reverse` that both reverse a list.

```
(define (Reverse-slow x)
  (if (empty? x) x
      (append (Reverse-slow (cdr x)) (list (car x)))))
```

- The first version `Reverse-slow` append the first element to the result of reversing the rest of the list.
- If we use stepper to trace `Reverse-slow`, each use of `append` only counts as one step. However, we know that the steps `append` takes is proportional to the length of the list. If we replace `append` with a function that we wrote by ourselves, we would see that the step count in stepper increases.
- Reversing a list of 3 elements takes 55 steps. If we triple the size of the list, it will take 419 steps. This is an indication that the function is slower then linear. In fact, it takes quadratic time.
- Consider an alternative implementation.

```
(define (Reverse x)
  (Reverse-helper x empty))
(define (Reverse-helper x acc)
  (if (empty? x) acc
      (Reverse-helper (cdr x) (cons (car x) acc))))
```

You can assume this is equivalent to Racket's builtin `reverse`, and the number of steps is a linear function of the size of `x`.

13.3 Ordered Lists

- We define an *ordered list* to be a list of which elemnts are strictly increasing.

- ```
(define (ordered? x)
 (cond
 [(empty? x) true]
 [(empty? (cdr x)) true]
 [(< (car x) (cadr x)) (ordered? (cdr x))]
 [true false]))
```

- It would be nice if we can turn any list into an ordered list. Here is a particular implementation of `order`.

- ```
(define (ins-ordered e x)
  (cond
    [(empty? x) (list e)]
    [(< e (car x)) (cons e x)]
    [(> e (car x)) (cons (car x) (ins-ordered e (cdr x)))]
    [true x]))
(define (order x)
  (if (empty? x) x (ins-ordered (car x) (order (cdr x)))))
```
- Note that this implementation removes duplicate elements from the list.

13.4 List vs. (Decorated) Trees

- Remember lists are sort of like unary trees. Lists can be thought of a binary tree that has only the right subtree.
- Both of lists and trees can act as sets, they can be searched, ordered. We can also insert and delete elements from it, etc.
- One kind of ordered trees is a BST.
- List is a simpler structure, it's much easier to search, insert and remove elements.
- Trees sometime allow more efficient algorithms to be implemented. Eg. Insert and delete from BST is proportional to the *height* of the tree.
- Consider `(insert-bst e b)` that is linear in the height of `b`.
Suppose we have a very long list of numbers and insert them into a tree with `insert-bst`.
- The resulting shape of the BST depends on the order in which you insert the elements. You can get anything from a min-height tree to a max-height tree.

```
(define l1 (list 1 3 4 2))
(define l2 (list 4 3 2 1))
(define l3 (list 1 2 3 4))
```

- If you insert elements in order, it will be a max-height tree.
- For min-height trees: if we always insert the median first, it will be a min-height tree.
- `(list 1 2 3 4 5 6 7)`.

If we insert in the order of 4 2 6 1 3 5 7 0 it will be a min-height (full) tree.

- However this is harder in practice, because finding the median of the list is not that easy. Another approach is that you find the “almost median”, by just picking an random element.
- For a very large list, the resulting tree should be very close to a min-height tree. As the probability of shuffling the list to be an ordered list is extremely low ($2 \times n!$).

- In case you can convince yourself that this method is good enough at least within your lifetime. There is one more problem: humans are terrible at making up randomness. We can use Racket's builtin function `random` that generates a pseudo-random number.

13.5 More Racket feature: random and racket/base

- `(random n)` produces a random number between 0 and $n - 1$.
- Consider the following function:

```
(define (ran-list n)
  (if (zero? n) empty
      (cons (random 1000000)
            (ran-list (sub1 n)))))
```

- Then, we write the following definitions:

```
(define a1 (ran-list 4))
(define a2 (ran-list 4))
(define a3 (ran-list 4))
```

After running the program, we check if `a1`, `a2`, and `a3` are equal. They turn out to be different. Note that, after they got evaluated by running the program, they become fixed for the interactions.

- Meanwhile, suppose we implement an equivalent of `ran-list2` below:

```
(define (ran-list2 n)
  (if (zero? n) empty
      (append (ran-list2 (sub1 n))
              (list (random 1000000)))))
```

We can see that the number of function applications of `ran-list2` is huge; in fact, you can draw a table and see.

Lecture 14 (October 6, 2020)

The lecture focuses on lambda expressions (in Racket). Lambda expressions is enabled for Intermediate Student with `lambda`.

14.1 Motivation

- Some functions we write have striking similarities. The structure of the codes look very similar.
- Also, when we use apply arguments to a function, we sometimes have to evaluate the value of the arguments first. But we never evaluate the “value” of the function.
- ```
(define (doit n i f)
 (if (zero? n) i
 (f n (doit (sub1 n) i f)))))
```
- `n` is the *size* of the result, `i` is the *identity* value (the value in the trivial case), `f` is *what should be done* to the result of the recursive call. Note the position of `f` implies it must be a function.
- `(doit 10 0 +)` is equivalent to `(sumto 10)`.  
`(doit 10 1 *)` is equivalent to `(factorial 10)`. `(doit 10 empty cons)` gives a list from 10 to 1.
- To get an increasing list instead, we need another function `snoc` (`cons` backwards).  

```
(define (snoc a b) (append b (list a)))
```

  
`(doit 10 empty snoc)` would give a list from 1 to 10.
- ```
(doit 10 empty (lambda (a b) (append b (list a))))
```


We can just pass the content (value) of `snoc` to `doit` directly. `lambda` is this “constructor” that allows you to build *anonymous* functions.
- We can also do `(λ (a b) (append b (list a)))`. Use `Ctrl-\` to type the `λ`. (Lambda comes from lambda calculus invented by Alonzo Church, and is the building block for lisp and its derivatives).

14.2 Lambda Expression

- The general syntax for lambda is `(lambda (arguments) (body))`.
- We have a new predicate `(procedure? f)` that determines if `f` is a function (lambda). `(procedure? +)` gives `true`.
- Functions are now no different from other values.

- In the stepper, every name of functions is first substituted by the actual lambda expression, and then the arguments are applied.
- `(define (foo x) (+ x x x))` is actually `(define foo (lambda (x) (+ x x x)))` which associates the lambda (that triples its only argument) to the name `foo`.
- Most importantly, you can pass functions as parameters, and you can return functions as a value.
- ```
(define (addn n) (lambda (x) (+ x n)))
> (addn 10)
< (lambda (a1) ...)
> (define add10 (addn 10))
> (add10 2)
< 12
```

`addn` returns a function that takes one argument, and add  $n$  to that argument.

### 14.3 Abstract List Functions

- `(build-list n f)` gives a list of `(list (f 0) (f 1) ... (f (- n 1)))`
- `(build-list 3 add1)` gives `(list 1 2 3)`.
- `(map f lst)` applies `f` to every element of `lst`.
 

```
- (map sqr (build-list 10 add1))
- (map (lambda (x) (* 2 x)) (build-list 10 add1))
```
- `(foldl f i l)`. `foldl` takes a list `l`, takes a function `f` that takes 2 arguments, and an initial value. It then uses the combining function to build an accumulator, that traverses through the list.
 

Eg. `(foldl + 0 (list 1 2 3))` applies `+` to 0 and 1, then that result to 2, then to 3, etc.
- Go back to the `doit` example, we want to have a different version of `doit` that applies a function to all of the elements in a list.
 

```
(define doit-list
 (lambda (lst i f)
 (if (empty? lst) i
 (doit-list (cdr lst)
 (f (car lst) i)
 f))))
```

Example:

- `(doit-list (build-list 10 add1) 0 +)`
  - `(doit-list (build-list 10 add1) 0 *)`
  - `(doit-list (build-list 10 add1) empty cons)`
  - `(doit-list (build-list 10 add1) empty snoc)`
- It turns out that `doit-list` is just `foldl`. We can use `foldl` to write `sumto`, `factorial`, and many other functions with `build-list`.
  - One more list function: `foldr`. Essentially the same as `foldl`, but “fold” the list in the opposite direction.

## Lecture 15 (October 10, 2020)

### 15.1 Modules and time

- A module is a file containing Racket code that *provides* definitions to a client.
- A client is a Racket program that requires definitions provided by a module.
- Some examples:

foo.rkt

```
#lang racket
(provide number)

(define number 42)
(define question 'huh)
```

client.rkt

```
(require "foo.rkt")

(+ number 30)
```

- Another useful tool for debugging is the function `time`, which can be found on the module `racket/base`. To access it, add `(require racket/base)` into your program.
- `(time expr)` evaluates `expr` and records the amount of time needed to evaluate it. More specifically, it records the CPU time, real time, and garbage collection (gc) time needed.
- A useful function for observing the effect of doubling the input size:

```
(define (timer f n)
 (map (lambda (x) (time (car (f (build-list x add1)))))
 (build-list n (lambda (x) (expt 2 x)))))
```

- To use a module, first make sure you have the racket file that provides the module's location. Then, in the *client* racket program use `(require "modulename.rkt")` to use the module. Then all expressions in the required file(s) will be evaluated. The language of the client program is not restricted.
- By default any definition in the module is not visible in the client file. To change that, use `(provide ...)` to provide the definitions so client can use them directly.
- Modules can be used to provide an ADT, so that only certain functions are exposed to clients. Meanwhile, clients should not have any interest in how the module functions are implemented.



## 15.2 Order Notation

- So far we have encountered steps, clock time, CPU time, GC time, etc. These concepts can be unified in an abstract level.
- Suppose that we are given a function or program, and we want to calculate its running time, given an input. Suppose that for some  $n$ , its running time is  $T(n)$ .
- Now, we need to be explicit of what  $n$  is, and in what unit  $T(n)$  is. Some possible examples for  $n$  would be `(length 1)`, where `1` is a list, or  $n$  could be the input itself, or maybe it could be a value based on the input. For  $T(n)$ , its unit could be the number of steps needed, the CPU or real time, or the number of function applications, for example.
- The exact value of  $T(n)$  could be very complicated (and it would) if the given function is more complicated. So, we could instead simply analyze its behaviour as  $n$  grows large. This is what the order notation is used for. A particularly important one here is the Big O notation.
- The simplest example for a linear function is  $T(n) = n$ . In Big-O notation we write it as  $\mathcal{O}(n)$
- We could say any of the following, but the meaning is the same:
  - $h(n) \in \mathcal{O}(g(n))$ .
  - $h(n)$  is  $\mathcal{O}(g(n))$ ,
  - $h(n) = \mathcal{O}(g(n))$ ,
- $\mathcal{O}(g(n))$  is actually defined to be a set of functions, which makes the last notation makes less sense.
- We also use  $\mathcal{O}(f(n))$  to represent a function that is in the set of  $\mathcal{O}(f(n))$ .
- Some algebra on Big O notation:
  - $\mathcal{O}(h(n)) + k = \mathcal{O}(h(n))$  and  $c \cdot \mathcal{O}(h(n)) = \mathcal{O}(h(n))$  for any constants  $k$  and  $c$ .
  - $\mathcal{O}(k) = \mathcal{O}(1)$  for any constant  $k$ ; it is the set of all functions bounded by a constant.
  - $\mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$ .
  - $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ .
  - $\mathcal{O}(g(n)) + \mathcal{O}(h(n)) = \mathcal{O}(h(n))$  if  $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$
- An example on Big O: Say that  $h(n) = 3n + 10 + n^2$ . Then,  $h(n) = \mathcal{O}(3n) + \mathcal{O}(10) + \mathcal{O}(n^2) = \mathcal{O}(n) + \mathcal{O}(1) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$ .

- A false proposition:  $O(1) + O(1) = O(1)$ , therefore by induction  $\sum_{i=1}^n O(1) = O(1)$ .  
And by the same logic  $O(f(n)) = O(1)$  for any  $f$ .  
 $\sum_{i=1}^k O(f(n)) = O(f(n))$  is only true when  $k$  is a *constant*.
- Some ordering on Big O:

$$O(1) \subseteq O(\log n) \subseteq O(\sqrt{n}) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^{1.5}) \subseteq O(2^n)$$

In general, it can be proved that for any constants  $k \leq m$ ,  $O(n^k) \leq O(n^m)$  and  $O(k^n) \leq O(m^n)$ .

## Lecture 16 (October 19, 2020)

### 16.1 Formal Big-O

- We consider Big-O notation from a mathematical perspective. For additional reading see the reference on order notation and the preamble of assignment 6.
- $\mathcal{O}(f(x))$  is a set of functions that can be represented in set-theoretic notation as follows:
- $\mathcal{O}(f(x)) = \{g(x) | \exists c > 0. \exists x_0. \forall x \geq x_0. g(x) \leq cf(x)\}$
- We say that  $g(x) \in \mathcal{O}(f(x))$  if  $\exists c > 0. \exists x_0. \forall x \geq x_0. g(x) \leq cf(x)$ .

### 16.2 Big-O proofs

Two proofs were presented. One is a proof that  $10x^2 + 20 \in \mathcal{O}(x^2)$  the other is a proof that  $x^2 \notin \mathcal{O}(x)$ .

Proof 1: See lecture video

Proof 2:

Let  $c$  and  $x_0$  be given. Take  $x = \max\{x_0, c + 1\}$

| step# | step                         | reason                       |
|-------|------------------------------|------------------------------|
| 1     | $x \geq c + 1$               | property of max              |
| 2     | $x^2 \geq x(c + 1) = xc + x$ | know that $x \geq c + 1 > 0$ |
| 3     | $x^2 \geq xc + x > xc$       | $x > 0$                      |
| 4     | $x^2 > cx$                   | combine 2 & 3                |

Hence for any choice of  $c$  and  $x_0$  we can find an  $x \geq x_0$  such that  $x^2 > cx$ . We conclude that  $x^2 \notin \mathcal{O}(x)$ .

### 16.3 Big-O Identities

We present some common and useful identities to help us manipulate Big-O expressions:  
The following are equivalent:

1.  $a\mathcal{O}(f(x)) + b$
2.  $\mathcal{O}(f(x))$

The following are equivalent:

1.  $\mathcal{O}(f(x)) + \mathcal{O}(g(x))$
2.  $\mathcal{O}(f(x) + g(x))$
3.  $\mathcal{O}(\max\{f(x), g(x)\})$
4.  $\mathcal{O}(f(x)) \cup \mathcal{O}(g(x))$

5.  $\mathcal{O}(f(x))$  if  $\mathcal{O}(f(x)) \supseteq \mathcal{O}(g(x))$
6.  $\mathcal{O}(g(x))$  if  $\mathcal{O}(g(x)) \supseteq \mathcal{O}(f(x))$

The following are equivalent:

1.  $\mathcal{O}(f(x)) * \mathcal{O}(g(x))$
2.  $\mathcal{O}(f(x)g(x))$
3.  $\mathcal{O}\left(\sum_{i=0}^{\mathcal{O}(f(x))} \mathcal{O}(g(x))\right)$

## 16.4 Big-Omega and Big-Theta

- Knowing that Big-O is the upper bound (worst case) running time it is natural to ask if we care about lower bounds of run-time.
- Big-Omega is the equivalent lower bound. We can define Big-Omega in a similar way to how we defined Big-O. Alternatively we can reuse our previous definition and say:
- $g(x) \in \Omega(f(x))$  means that  $f(x) \in \mathcal{O}(g(x))$
- Finally, we define Big-Theta notation.  $\Theta(f(x))$  is both an upper and lower bound.
- $g(x) \in \Theta(f(x))$  means that  $g(x) \in \mathcal{O}(f(x))$  and  $g(x) \in \Omega(f(x))$

## Lecture 17 (October 20, 2020)

We discuss trees and various recursion techniques on trees. All trees in this lecture are made using `(define-struct node (l k r))`. Notice that the key is in the middle.

### 17.1 Building More Trees

- We make an insert function `ins` that consumes a number and a bst and inserts the given number into the bst. (As usual)
- The following two definitions allow us to build generic trees.
- `(define N 10)`
- `(define x (foldl ins empty (build-list N (λ (x) (random N)))))`

The above ideas may useful for testing. Try to see if you can use `foldl` and `build-list` to test your functions without working too hard.

### 17.2 Recursion on Trees

- We can sum the elements of a tree:
- ```
(define (sum t)
  (if (empty? t) 0 (+ (sum (node-l t)) (node-k t) (sum (node-r t)))
      ))
```
- In the spirit of A5, we can make an ordered list of the elements in a similar manner using `append` instead of `+`:
- ```
(define (lst t)
 (if (empty? t) empty (append (lst (node-l t)) (list (node-k t)) (lst
(node-r t))))
))
```
- Both of the above functions are slow for large inputs.
- We can speed up `lst` by using `cons` instead of `append`.
- `append` and `+` are both associative. When using `cons` we do not have this luxury so we must be careful to apply it in the correct order.
- Below we use a helper function to make a more efficient `lst`.
- ```
(define (lst1 t) (lst1-helper t empty))
```

- ```
(define (lst1-helper t acc)
 (if (empty? t)
 acc
 (lst1-helper (node-l t) (cons (node-k t)
 (lst1-helper (node-r t) acc)))
))
```
- Although we do use two recursive calls, we reduce the size of the tree so as to use only one call to `cons` per node.
- Another way to build a tree for testing is to call:
- ```
(define y (foldl (λ (x y) (make-node y x empty))
  empty (build-list 1000000 add1)))
```
- Notice the similarity in structure of `sum` and `lst`.

17.3 Tail-Recursive/Singly-Recursive Sum Function

The lecture ends with a demonstration of a tail-recursive version of the above `sum` function. For the exact implementation refer to the lecture video. We will describe the intuition and strategy here.

- Idea: Use a tail-recursive helper function.
- Plan: one parameter will be a list of all the trees that we need to sum. We'll call this our worklist. The second parameter will be an accumulator that stores the cumulative sum.
- First of all, our function `sum` will consume a tree `t` and pass `(list t)` and 0 as initial values to our helper function.
- Case 1: If the worklist is empty, we can return the accumulator.
- Case 2: If the first tree in the worklist is empty we can call our helper on the `cdr` of the worklist and the same accumulator.
- Case 3: We now know that the first tree in the worklist is a node. We call out helper function with the left and right subtrees added to the list and the key added to the accumulator.
- After the above case we have reduced the size of the trees in the worklist.

As an exercise, try to write this function (or a function of the same form) without referring back to the implementation in the lecture.

Lecture 18 (October 20, 2020)

18.1 Introduction to Information Hiding

Why would we want to hide information?

- Every module hides a secret(s).
- To preserve an invariant property. (As with the BST)
- Prevent malicious usage.
- Prevent inadvertent violations.
- To simplify our programs.

Q: How does hiding our information in modules simplify our programs?

A: Two lines of code in separate modules cannot interact with each other.

Splitting a large program into many modules can asymptotically cut down on the number of possible interactions. This means that our program will require asymptotically less reading to understand.

18.2 Example: BST

Recall from last lecture and prior our familiar BST using the implementation (`define-struct node (l k r)`).

We wrote functions:

- `ins` which inserts a number `x` into a BST `t`.
- `sum` adds up all of the numbers in a BST `t`
- `lst` converts a BST `t` into an ordered list.

If we save these definitions in a full Racket file called `bst-adt.rkt` we can use (`require "bst-adt.rkt"`) in another Racket program to get access to any of the provided definitions. Below are some examples of what we can do with our new module:

- Use (`provide ins sum lst`) to provide users of access to these functions.
- The above code does not allow users access to `make-node` which they could use to break our BST ADT implementation.
- They could break the ADT by building a node that does not have BSTs as subtrees.
- A more subtle way to break the ADT is to violate the invariant by build a decorated tree that is not a BST.
- We can provide access to the field selectors by using (`provide node-k node-l node-r`)

- An alternative approach is to give these definitions an alias and provide that.
- For example: `(define node-key node-k)` and then `(provide node-key)`
- Information hiding prevents the use of counterfeit structs.

Lecture 19 (October 26, 2020)

The lecture mostly talks about total order and struct as wrappers.

19.1 Total Order

- BST is essentially an ordered-set. The definition for a BST involves defining a relation between the elements. For example, the usual \leq for real numbers. But we could also define other relations that works on other elements. ($>=$, `string<=?`, etc)
- A relation \leq on a set S is called total order if:
 - Reflexive property: For all $x \in S$, $x \leq x$.
 - Total property: For all $x, y \in S$, at least one of $x \leq y$ or $y \leq x$ is true.
 - Anti-symmetric property: For all $x, y, z \in S$, if $x \leq y$ and $y \leq x$, then $x = y$.
 - Transitive property: For all $x, y, z \in S$, if $x \leq y$ and $y \leq z$, then $x \leq z$.
- We can derive some relations from the total property.

Relation	Definition
$x \geq y$	$y \leq x$
$x = y$	$x \leq y$ AND $y \leq x$
$x > y$	NOT $x \leq y$
$x < y$	$y > x$
$x \neq y$	$x < y$ OR $y < x$

- A relation $<$ on a set S is called strict total order if:
 - For all $a, b \in S$, either $a < b$ or $b < a$ or $a = b$.
 - For all $a, b, c \in S$, if $a \leq b$ and $b \leq c$, then $a \leq c$.
 - For all $a, b \in S$, if $a < b$, then both $b < a$ and $b = a$ does not hold.
- We could also derive relations based on strict total order.

Relation	Definition
$a \geq b$	NOT $a < b$
$a > b$	$b < a$
$a \leq b$	$b \geq a$
$a = b$	NOT ($a < b$ OR $b < a$)
$a \neq b$	$a < b$ OR $b < a$

- Note that \leq is merely a symbol we use to represent the total order. For example we can define it to be

```
(lambda (a b)
  (cond
    [(< (abs a) (abs b)) true]
    [(> (abs a) (abs b)) false]
    [true (<= a b)]))
```

the “absolute value order”. Then under this order elements in \mathbb{Z} would be

$$0, -1, 1, -2, 3, \dots$$

- If we define another strict total order

```
(lambda (a b)
  (< (abs a) (abs b)))
```

Then -10 and 10 would be the “equal” according to this order.

- In our `bst-adt.rkt` file we used the regular `<` for the order. But we can define another total order lambda and replace it by the new function. Then `ins` would work according to the new total order.
- If we modify `ins` to accept a third argument, we can pass the total order in as an lambda to avoid changing the module all the time.
- To prevent stupid users and malicious users from breaking up the BST, we introduce a new way of using `define-struct`.
- We use a *wrapper* to pack the total order and the BST that follows the total order together.

```
(define x
  (foldl
    (lambda (e t) (ins e t))
    (newbst >) ; a new empty bst with total-order >
    (build-list 10 (lambda (x) (- 5 (random 10))))))
```

creates a random BST according to `>`.

19.2 Small Note on eq?

- When a structure or a lambda is defined, they obtain a “serial number.”
- `eq?` produces true if and only if the two inputs have the same serial number.
- Some example:

```
(define q (make-posn 1 2))
(define r (make-posn 1 2))
> (eq? q r)
< false
```

- ```
(define foo (lambda (x) x))
(define bar (lambda (x) x))
(define foofoo foo)
```

foo and bar are not equal, but foofoo and foo are equal.

## Lecture 20 (October 27, 2020)

Racket's builtin sort and sorting compound data.

- Racket has a built in function (`sort lst <`) which consumes a list and a comparing function `<`, and sort the list according to `<`.
- `sort` will keep any duplicate element, but the order of the duplicate elements in the resulting list is not specified.
- To sort compound data, we often need to define our own total order.
- Lexicographic order: compare each element in the list respectively, the first different element that is smaller, or the first list that reaches its end, is smaller.

This comparing method is common for strings, it's sometimes just called alphabetical order.

```
(define (lex< a b)
 (cond
 [(empty? a) (not (empty? b))]
 [(empty? b) false]
 [(< (car a) (car b)) true]
 [(< (car b) (car a)) false]
 [true (lex< (cdr a) (cdr b))]))
```

- We can even generalize `lex<` to use a custom comparing function `<`, so the element in the list can be anything that has a total order that we defined.

We can just change the above definition to (`lex< a b <`) and add the parameter in the recursive call as well.

- Now to use `sort` with this new `lex<`, we need to wrap it with another function.

```
(define (gen-lex <)
 (lambda (a b) (lex< a b <)))
```

This is a lambda function that returns another lambda function, which takes two parameters instead of three, as required by `sort`.

## Lecture 21 (October 27, 2020)

Sorting Algorithms.

### 21.1 Insertion Sort

- It inserts each element of the list into the accumulator by sorted insertion. Using a pair of accumulators **old** and **new**, the following table represents the algorithm.

| Step count | old                       | new                       |
|------------|---------------------------|---------------------------|
| 0          | (30 70 10 80 60 50 40 20) | ()                        |
| 1          | (70 10 80 60 50 40 20)    | (30)                      |
| 2          | (10 80 60 50 40 20)       | (30 70)                   |
| 3          | (80 60 50 40 20)          | (10 30 70)                |
| 4          | (60 50 40 20)             | (10 30 70 80)             |
| 5          | (50 40 20)                | (10 30 60 70 80)          |
| 6          | (40 20)                   | (10 30 50 60 70 80)       |
| 7          | (20)                      | (10 30 40 50 60 70 80)    |
| 8          | ()                        | (10 20 30 40 50 60 70 80) |

- Invariants:
  - old** and **new** contains all the elements to be sorted.
  - new** is ordered.
- Each step  $i$  takes  $O(i)$  time, and we have  $n$  steps where  $n$  is the length of the list.
$$\sum_{i=1}^n O(i) = O(n^2).$$
 So the whole algorithm runs in  $O(n^2)$  time.

### 21.2 Selection Sort

- Table for selection sort:

| Step count | old                       | new                       |
|------------|---------------------------|---------------------------|
| 0          | (30 70 10 80 60 50 40 20) | ()                        |
| 1          | (30 70 10 60 50 40 20)    | (80)                      |
| 2          | (30 10 60 50 40 20)       | (70 80)                   |
| 3          | (30 10 50 40 20)          | (60 70 80)                |
| 4          | (30 10 40 20)             | (50 60 70 80)             |
| 5          | (30 10 20)                | (40 50 60 70 80)          |
| 6          | (10 20)                   | (30 40 50 60 70 80)       |
| 7          | (10)                      | (20 30 40 50 60 70 80)    |
| 8          | ()                        | (10 20 30 40 50 60 70 80) |

- Invariants:
  1. `old` and `new` contains all the elements to be sorted.
  2. `new` is ordered.
  3. Every element `x` of `old` and every element `y` of `to` satisfies `x < y`.
- Now, each step takes  $O(n - i + 1)$  time. The total running time is  $O(n^2)$ .

## 21.3 BST sort

- We have learned in A5 that we can use BST to sort a list.
- For inserting to BST, each step takes the following amount of time:
  - Worst case:  $O(i)$
  - Average case for random input:  $O(\log i)$
  - AVL worst case:  $O(\log i)$
- However BST does not allow duplicate elements. To work around this, we can modify the BST to hold pairs, the number and the number of times the element occurs. Another method is to implement a tie-breaker, append every element with a unique number. Either method requires us to modify the total order rules.

## 21.4 Merge Sort

- Table for bottom-up merge sort:

| Step count | <code>acc</code>                          |
|------------|-------------------------------------------|
| 0          | ((30) (70) (10) (80) (60) (50) (40) (20)) |
| 1          | ((30 70) (10 80) (50 60) (20 40))         |
| 2          | ((10 30 70 80) (20 40 50 60))             |
| 3          | ((10 20 30 40 50 60 70 80))               |

- Invariant for bottom-up merge sort:
  - `acc` is a list of lists, each contains the elements to be sorted.
  - Each sublist of `acc` is ordered.
- The total running time for each step is  $O(n)$  since merging two ordered lists takes  $O(m)$  time where  $m$  is the sum of the length of the two lists. There are  $\lceil \log_2 n \rceil$  steps, so the total running time is  $O(n \log n)$ .
- Another variant is the top-down merge sort. It is more easily explained through normal recursion.

- If the list contains at most one element, produce itself. Otherwise, we divide the list into half, apply merge sort into each half, and then merge them.
- Racket function `sort` is a merge sort.

## 21.5 Quick Sort

- If the input list `l` is empty, output is obviously empty.
- Otherwise, pick a (pseudo-)random element `e` from `l`. Then, form two lists, where the first one is a list with elements `y` from `l` satisfying `y < e`, and the other one is a list with all other elements from `l`, except `e`.
- Finally, apply quicksort on each of the sublists and append them together with `e`.
- Average running time is  $O(n \log n)$ , but worst-case running time is  $O(n^2)$ .
- Suppose that we want to list the first 10 Fibonacci numbers. We can do `(build-list 10 fib)` and provide an implementation of `fib` for the Fibonacci numbers.
- One possible way to do it is the following:

```
(define (fib n)
 (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
```

However, this takes exponential amount of steps. In fact, it is  $O(F_n)$ , where  $F_n$  is the  $n^{\text{th}}$  Fibonacci number.

- We can make it better using accumulative recursion.

```
(define (fib n) (fibhelp 0 1 n))
(define (fibhelp a b n)
 (if (<= n 0) a (fibhelp b (+ a b) (sub1 n))))
```

- This `fib` variant now takes  $O(n)$ . However, the `build-list` will take  $O(n^2)$ .
- We can do better for generating the Fibonacci numbers from scratch. The following does not work, but it can be modified to work.

```
(foldl (λ (a b) (list (cadr b) (+ (car b) (cadr b))))
 (build-list n add1) (list 0 1)))
```

## Lecture 22 (November 2, 2020)

### 22.1 Generating Functions

In this lecture we are concerned with generating functions and their applications.

**Warning:** these are not the same generating functions as in mathematical enumeration!

We have already seen some examples:

- `build-list`
- `foldl`
- `map`

We can explore our familiar `sumto` in the context of generating functions:

- Recall the simple recursive `sumto`.
- Recall the tail-recursive `sumto`.
- We can use a `sumbetween` that only accepts one list parameter.
- Recall the `sumto` implementation using `build-list` and `foldl`.
- We can make a new `sumto` using `generate`:

```
(define (sumto-fast4)
 (generate
 (list 0 n 0);; initial state
 (λ (st) (> (car st) (cadr st)));; stopping condition
 (λ (st) (list (add1 (car st)) (cadr st));; how to get the next value
 (+ (car st) (caddr st))))
 (λ (st) (caddr st))))
```
- The above version of `sumto` outsources the recursion of the `sumto` below to `generate`.

Aside: `define` can be used inside a function body to create a local definition. This may make programs easier to read by giving values meaningful names.

- What happens if we need to use a function that only accepts and outputs one parameter?
- We can combine all of our parameters into a list.
- We could also use a structure to hold the information.
- A local `define` can simplify programs that hold all of their data in a single data structure.
- We can apply local `define` to `sumbetween` by assuming that it accepts an input state that is a list containing a, b, and acc.



- ```
(define (sumbetween st)
  (define a (car st))
  (define b (cadr st))
  (define acc (caddr st))
  (if (> a b) acc
      (sumbetween (list (add1 a) b (+ a acc))))))
```
- ```
(define (sumto n)
 (sumbetween (list 0 n 0)))
```

## 22.2 The generate Function

One of the most important generating functions this week is **generate**. This should be rather unsurprising considering the name. **generate** and functions in the same family will be provided in future assignments. Hence it is important to master the general version below.

- ```
(define (generate first done? step final)
  (if (done? first) (final first) (generate (step first) done? step
      final))))
```
- Intuitively, this function will repeatedly apply **step** to **first** until **done?** is true. After that, **final** is applied to the result.
- We can outsource common recursions to **generate**.
- The **sumto** example is above.
- **(factorial n)** and **(fib n)** can be created in a similar manner.

Lecture 23 (November 2, 2020)

23.1 Infinite Sequences

We have spent much time discussing lists of finite length. In Racket, lists of infinite length do not present any immediate problems unless we attempt an operation that requires accessing all of the elements such as `length` or `reverse`. We will refer to infinite lists as infinite sequences or sequences.

- Our first example of an infinite sequence is the sequence of Fibonacci numbers which we call `fibseq`.
- We use an ADT called `fibseq`. The key operations are as follows:
 1. `fibseq` is a sequence of Fibonacci numbers.
 2. `(fibcar f)` gives the first element of the sequence.
 3. `(fibcdr f)` gives the rest of the sequence.
- We can implement this in Racket by using the data definition (`define-struct f (fib nxt)`) to store the current and next Fibonacci numbers. `fibcar` and `fibcdr` simply return or compute the next values.
- Notice that our infinite sequence does not actually take up an infinite amount of memory or operations. We only need to provide instructions on how to get an arbitrary element. The memory and operations are only evaluated up until the point we need.

23.2 The Stream ADT

It may have been disappointing to discover that our previous implementation of an infinite sequence only worked for the Fibonacci numbers. The Stream ADT is capable of handling much more general sequences.

- An implementation can be found in `stream.rkt`.
- A description of the ADT can be found in the preamble to A9. **It is important to read and understand this ADT description!**
- The idea here is that `(stream-generate first done? step final)` can be used to construct a variety of streams.
- Example: Fibonacci numbers using `stream-generate`. We use the same struct definition as above.

- ```
(define fibseq
 (stream-generate
 (make-f 0 1)
 (λ (st) false)
 (λ (st) (make-f (f-nxt st) (+ (f-fib st) (f-nxt st))))
 (λ (st) (f-fib st))
))
```
- This ADT could be completed by providing `fibcar` and `fibcdr`.
- ```
(define fibcar stream-car)
```

 and

```
(define fibcdr stream-cdr)
```

Lecture 24 (November 5, 2020)

This lecture covers deferred evaluation, input/output, and introduces Lazy Racket.

24.1 Deferred Evaluation

- Problem: How could we build special forms such as `if` if they were not provided to us?
- Solution: We could use λ to defer evaluation of an expression such as `expr`.

Examples:

- `(define x expr)` evaluates the expression as usual.
- `(define (x y) expr)` does not evaluate `expr` until `x` is called on any parameter as in: `(x 'wilma)`. The technique of wrapping an expression in a function to delay evaluation is called *promising*.
- The function does not need to accept a parameter. `(define (xx) expr)` is only evaluate when `(xx)` is called.
- The definition does not need to be a function. `(define xxx (λ () expr))`
- The function does not need to be named. `(λ () expr)`

Promises can be used to define `myif` which computes the same result as `if`.

- First try: `(define (myif t a b) (if t a b))`.
- This definition requires `t`, `a`, and `b` to be evaluated before substitution. Hence it does not work in the same way as the builtin `if`.
- Second try: `(define (myif1 t a b) (if t (a) (b)))`
- This version uses promises to defer evaluation. It works in the same way as the builtin `if`. However, `myif1` expects `a` and `b` to be promises.

Promises and deferred evaluation are explained by way of several Racket examples. Notable among them are:

- `(define (zap x) (x x))`
- `(define (ident x) x)`
- Consider the output of `((zap ident) 10)`.
- Consider the output of `(zap zap)`.
- The above program written using lambdas is: `((lambda (x) (x x))(lambda (x) (x x)))`.

Next we see a discussion of the Stream ADT in the context of lambdas.

- Streams are lazy lists.
- In a stream created by `stream.rkt` only the first element is computed and the rest are promised.
- This is done by wrapping the rest of the list in a lambda, and evaluating it only when we call `stream-cdr`.
- This is how we can achieve $\mathcal{O}(1)$ time in A9.

24.2 Input/Output

- Recall that `(random n)` returns a different value every time it is called. This can be interpreted as returning the `stream-car` of a pseudo-random stream of numbers.
- `(read)` will read from Racket's input.
- Whenever `(read)` is evaluated, Dr Racket will prompt the user for input.
- Notice that numbers are read as numbers and lists are interpreted as lists of symbols.
- `(display expr)` returns `void` and causes the side effect of printing `expr` to the standard output in purple.

In full Racket there is nothing stopping us from writing programs with multiple expressions when this does not create any ambiguity. The output is the last expression to be evaluated. Hence this does not add anything to our programs except where side effects can occur.

- `(begin expr1 expr2 expr3)` will evaluate all of the expressions that it consumes.
- Function bodies may include more than one expression.
- The test-value pairs inside of a `cond` statement can be used to evaluate multiple expressions.

24.3 Aside: Models of Computation

- Racket is a derivative of Scheme which is itself a derivative of Lisp. Lisp is based on the computational model of λ -calculus.
- Alonzo Church developed λ -calculus.
- Alan Turing created another computational model called the Turing Machine.
- Later on in this course we will discuss the RAM model of computation.

24.4 Lazy Racket

- There are two versions of Lazy Racket in Dr Racket.
- With determine language from source, we can enter Lazy Racket by using `#lang lazy`.
- We can also enter a slightly different version of Lazy Racket directly by selecting it under “Experimental Languages”. This version has a stepper, but is not the version used in the assignments.
- Lazy Racket is similar to full Racket, but it uses *Lazy Evaluation*.
- The computations of the results of functions and lambda expressions are delayed whenever possible.
- This means that we can define our own `if` without having to use promises because Lazy Racket will not compute the values before substitution as in our familiar `if`.
- All lists behave as streams. In fact, lists no longer need to be finite.
- For example: `(define r (cons 3 (cons 6 r)))` works in Lazy Racket.
- Try evaluating the following expressions in lazy Racket.

```
(define y (cons 1 y))
(car y)
(cadr y)
(caddr y)
(define (f x) (map add1 x))
(define z (cons 1 (f y)))
(car z)
(cadr z)
```

Lecture 25 (November 10, 2020)

The lecture starts a new material about Models of Computation and lambda calculus.

25.1 Lambda Calculus

25.1.1 notation

- A *lambda expression* is one of:
 - A variable: x, y, a, b, \dots ,
 - An application: $(e1\ e2)$, where $e1$ and $e2$ are lambda expressions.
 - An abstraction: $(\lambda x.e)$, where e is a lambda expression.
- We define the expressions to be left-associative, and omit the parentheses when there is no ambiguity.

Some shorthands we will use:

- e stands for (e) .
 - $\lambda x.e$ stands for $(\lambda x.e)$.
 - $e1\ e2\ e3$ stands for $(e1\ e2)\ e3$.
 - $\lambda x.e1\ e2$ stands for $\lambda x.(e1\ e2)$.
- Some examples of lambda expressions:

x
 $x\ x$
 $x\ y\ z$
 $\lambda x.x$
 $\lambda x.x\ x$
 $\lambda x.(\lambda y.x)$
 $(\lambda x.x).(\lambda y.y)$

To compute any lambda expression, we repeatedly substitute using two rules α -equivalence and β -reduce, until we have the desired result. We need two “helper functions” Free and Substitute to use these two rules.

25.1.2 rules

- Free variables are variables not bound by an abstraction. Suppose that $FV[e]$ denotes the set of free variables of a lambda expression e . Then,
 - If e is a variable x , $FV[x] = \{x\}$.

- If e is an application $e_1 \ e_2$, then

$$FV[e_1 \ e_2] = FV[e_1] \cup FV[e_2]$$

- If e is an abstraction $\lambda x. e'$, then

$$FV[\lambda x. e'] = FV[e'] \setminus \{x\}$$

- Now, we talk about substitutions. Some notations for substitutions:

$$e_1[x \leftarrow e_2]$$

$$e_1[x := e_2]$$

$$e_1[e_2 \rightarrow x]$$

$$e_1[x \rightarrow e_2]$$

$$e_1[x / e_2]$$

$$e_1[e_2 / x]$$

We will use $e_1[x \leftarrow e_2]$. Informally, we can describe the above as "Replace all free variable x in e_1 by e_2 ".

- To formalize it, we have a set of rules.

- For a variable x , $x[x \leftarrow e]$ becomes e .
- $(e_2 \ e_3)[x \leftarrow e_1]$ becomes $(e_2[x \leftarrow e_1] \ e_3[x \leftarrow e_1])$.
- $(\lambda x. e_2)[x \leftarrow e_1]$ becomes $\lambda x. e_2$. It does not really do anything.
- $(\lambda y. e_2)[x \leftarrow e_1]$ gives $\lambda y. (e_2[x \leftarrow e_1])$ if $x \neq y$ and $y \notin FV[e_1]$.

* The criterion for the last substitution is made to prevent "capture". An example of capture: $(\lambda x. x \ y)[y \leftarrow x]$ is substituted to $\lambda x. x \ x$.

* In original Lisp, this is an issue. According to the designer, John McCarthy, this is "merely a bug."

25.1.3 α - β reduction

- Now

- α -equivalence: Two lambda expressions $(\lambda x. e_1)$ and $(\lambda y. e_2)$ are said to be α -equivalent if $y \notin FV[e_1]$ and $e_2 = e_1[x \leftarrow y]$. We also denote this by \equiv_α .
- β -reduction: $(\lambda x. e_1) \ e_2$ β -reduces to $e_1[x \leftarrow e_2]$, provided that $x \notin FV[e_2]$. We also denote this by \Rightarrow_β .

- We say a lambda expression is a reducible expression (redex) if it can be β -reduced, with or without the help of α -conversion. We denote this by $\Rightarrow_{\alpha\beta}$. Not all lambda expressions are redexes.

- Some examples with these:

$$\begin{aligned}
 (\lambda x. a \ x \ b) \ (\lambda y. y) &\Rightarrow_{\beta} a \ (\lambda y. y) \ b \\
 (\lambda x. a \ x \ b) \ (\lambda y. x) &\equiv_{\alpha} (\lambda q. a \ q \ b) \ (\lambda y. x) \\
 &\Rightarrow_{\beta} a \ (\lambda y. x) \ b
 \end{aligned}$$

- For a lambda expression e :
 1. If $e = x$, it is not a redex.
 2. If $e = e1 \ e2$, and both $e1$ and $e2$ are redexes, then we can either $\Rightarrow_{\alpha\beta} e1$ or $e2$ first.
 3. Or we can also $\Rightarrow_{\alpha\beta} (e1 \ e2)$ altogether.
- If we always choose the third method, we *may* eventually get to a fully reduced form (a lambda expression with no redexes). Some expressions are always reducible.

25.2 Fully Reduce λ -expression

- Whether a given λ -expression has a fully reduced form is a undecidable question.
- Given that the λ -expression *has* a fully reduced form. *Normal Order Evaluation* can always get to that form.
- For normal order, we always reduce the left-most redex.
- For example, consider the following expression:

$$(\lambda x. x \ x) \ ((\lambda y. y) \ q)$$

- $((\lambda y. y) \ q)$ is a redex, while the whole thing is also a redex.
- Using normal order, the expression reduces to

$$((\lambda y. y) \ q) ((\lambda y. y) \ q)$$

which is then reducible to $q \ ((\lambda y. y) \ q)$, and finally $q \ q$.

- We also have applicative order, where we always reduce the inner most redex first. In this example the expression reduces to

$$(\lambda x. x \ x) \ q$$

which then is reducible to $q \ q$.

- In this case the two orders give us the same fully reduced form. Given a fully reducible λ expression, the normal order evaluation will *always* find it, but not applicative order.
- A λ -expression with no fully reduced form.

$$(\lambda x. x \ x) \ (\lambda x. x \ x)$$

- Another example

$$(\lambda x. y) ((\lambda x. x \ x) (\lambda x. x \ x))$$

25.3 Relation to (Lazy) Racket

- We can easily translate a λ expression into Racket.
- However Racket uses applicative order to evaluate expressions, if we want it to be evaluated with normal order, we use Lazy Racket.
- As you can see it is a little bit slower to do the same computation multiple times. Lazy Racket use normal order strictly with an additional tweak.
- The trick is to use memoization to simultaneously β -reduce all the same redex to avoid repeated computation.
- It turns out that lambda calculus is all we need for computation, no special form, `define`, `cons`, or all the extra things in the normal Racket.

Lecture 26 (November 11, 2020)

The lecture continues on lambda expressions. It talks about how we can convert human concepts into lambda expressions.

26.1 Translating to λ -calculus Expressions

- As a continuation of our hypothetical circumstances, we now have to build everything using lambda expressions. We start by trying to convert Racket expressions into lambda calculus expressions.

Racket Expression	λ -calculus Translation []
<code>'x</code>	<code>x</code>
<code>x</code>	<code>x</code>
<code>(R1 R2)</code>	<code>[R1] [R2]</code>
<code>(λ (x) R)</code>	<code>$\lambda x. [R]$</code>
<code>(λ (x y) R)</code>	<code>$\lambda x. \lambda y. [R]$</code>
<code>(λ (x) (λ (y) R))</code>	<code>$\lambda x. \lambda y. [R]$</code>
<code>(f R1 R2)</code>	<code>[f] [R1] [R2]</code>
<code>((f R1) R2)</code>	

- A few things to note
 - λ -calculus does not distinguish between bound and free variables like Racket requires
 - λ -calculus only have functions taking one argument. We convert any function (application) asking for more than one argument to “curried form” (named after Haskell Curry). In the above example $\lambda x. \lambda y. [R]$ is ambiguity-free.

26.2 Building ADT with λ -calculus

26.2.1 Booleans

- We define the two Boolean values to be

`true` `$\lambda x. \lambda y. x$`
`false` `$\lambda x. \lambda y. y$`

- Why? Consider the Racket expression `(if B 'y 'n)`, if B is true then 'y is returned, otherwise 'n.
- Consider `[B] ['y] ['n]` which is `((B y) n)` in λ -calculus expression, if B is `true` (in λ -calculus), then y is returned, otherwise n.
- We then define `if` in λ -calculus to be `$\lambda b. \lambda t. \lambda f. b \ t \ f$` , which essentially just takes the Boolean value and apply it with `t` and `f`.

26.2.2 Cons

- We then define `cons`, the selectors,

<code>(cons A B)</code>		<code>((λa.λb.λs.s a b [A]) [B])</code>
<code>(car C)</code>		<code>(λa.a true) [C]</code>
<code>(cdr C)</code>		<code>(λa.a false) [C]</code>
<code>empty</code>		<code>λa.true</code>
<code>(empty? C)</code>		<code>[C] (λa.λb.false)</code>

- The `s` in `cons` is a selector, which makes building `car` and `cdr` very easy.
- Note in our implementation of `empty?`, if we apply `empty` to `empty?`, then it will evaluate to `true`. However, it will return false if we pass a `cons` to `empty?`.
- By having `cons`, we can build all kinds of compound data structures.

26.2.3 Natural Numbers

Deferred to next lecture.

26.3 Non-recursive define with λ -calculus

- For some non-recursive definition (`define x R`), and some expression `S` that uses `x`, we can write it as `(λx.[S]) [R]` (potentially α -conversion is involved).

Lecture 27 (November 11, 2020)

This lecture continues the definition of various ADTs in λ -calculus, and talks about recursion. For simplicity, functions used are in uncurried form, but in Assignment 10 all functions are in curried form.

27.1 Recursion

- Consider the function which checks if every element in a list is `True`. The problem is that we cannot define it like this in λ -calculus because all functions are anonymous, so it cannot refer to itself directly.

```
(define And-list
  (lambda (lst)
    (If (Empty? lst) True
        (If (Car lst) (And-list (Cdr lst)) False))))
```

- We can do better by making a copy of itself as a parameter.

```

(define And-list-helper
  (lambda (self lst)
    (If (Empty? lst) True
        (If (Car lst) (self self (Cdr lst)) False))))
(define And-list1
  (lambda (lst) (And-list-helper And-list-helper lst)))

```

- If we curry `And-list-helper` and name it `And-curry-helper`, then we can define


```
(define And-curry-list (And-curry-helper And-curry-helper))
```
- If we just beta-reduce `And-curry-helper`, we don't need the helper lambda and write everything in `And-curry-list`.
- However this is cumbersome to do for every recursion and of bad style. We can generalize by using one of the fixed-point combinators — Y combinator.

```

• (define Generic-curry-nohelper ;; Y combinator
  (lambda (f)
    ((lambda (self) (f (self self)))
     (lambda (self) (f (self self))))))

(define And-generic
  (Generic-curry-nohelper
   (lambda (And-list)
     (lambda (lst)
       (If (Empty? lst) True
           (If (Car lst) (And-list (Cdr lst)) False))))))

```

27.2 Natural Numbers

- There are many ways to define natural numbers in λ calculus.
 - Church numerals — not covered in CS145.
 - Unary — n is a list of length n .
 - Binary — n is a list of binary digits (in our case `True` and `False`) representing base-2 numbers.
- Regardless of which representation we choose, we can define the following functions:
 - 0 — the smallest natural number.
 - `add1` — add 1 to a given number.
 - `sub1` — subtract 1 to a given number.
 - `add` — add two numbers
 - `mult` — multiply two numbers

Lecture 28 (November 17, 2020)

The lecture starts with a teaser about the game Simon and how computer works in the real world.

- In old days, a computer is always taking in a stream of user input, do the computation, and print a stream of output on its screen. Thus completing a read-eval-print loop (REPL).
- We use a terminal, which is a command-line interface to interact with the computer. The lecture's demonstration can be done in any *nix (including MacOS) system, or you can install Windows Subsystem for Linux (WSL) to try it on your own.
- When we press on the keyboard, the keyboard sends a distinct sequence of 0 and 1's (also called bits) to the computer for each key. Similarly, the computer sends a stream of bits to the screen to output characters (and perhaps colours). We need an *encoding scheme* for the sequences.
- One popular and widely used encoding scheme for characters on the keyboard is the ASCII format. ASCII is a 7-bit format, which means it can represent $2^7 = 128$ characters.
- For English 128 is more than enough, but it is not for other languages (European, CJK, Arabic). Unicode was created. It is a 32-bit format, currently it consists of 143,859 characters (according to Wikipedia).

28.1 Reading Characters in Racket: `read-char`

- A new type in Racket is introduced, a character type is represented with the some characters proceeded with `#\`.
- We have Racket functions `char?`, `char->integer`, etc to deal with character types.
- A string is essentially a sequence of chars. We have `string->list`, `list->string`, and many other functions to be found in the Racket documentation.
- `(read-char)` reads a character and produces the same character.
- Only the first character is produced if multiple keystrokes are given. Furthermore, subsequent `(read-char)` simply reads the next character put before.
- Some examples, each with separate program run:

1. (read-char)

```
qrs
< #\q
> (read-char)
< #\r
> (read-char)
< #\s
```

2. (define x (build-list 10 (λ (x) (read-char))))

```
abcdefghi
> x
< (#\a #\b #\c #\d #\e #\f #\g #\h #\i #\newline)
```

3. (define x (build-list 10 (λ (x) (read-char))))

```
abc
def
ghi
> x
< (#\a #\b #\c #\newline #\d #\e #\f #\newline #\g #\h)
```

- At the right of the input window, there is a yellow button named EOF, this signals a special value to Racket, such that it will consider it to be the end of all input. Note that EOF is not a character.

28.2 Reading various Racket types: read

- (read) reads from user input and make an object based on one of Racket data types.

- The following is an example of such interaction in strict Racket.

```
> (read)
12356327
< 12356327
> (read)
foo
< 'foo
> (read)
(a b c d e)
< '(a b c d e)
> (read)
(a b c (d e f) (g (h i)) j)
< '(a b c (d e f) (g (h i)) j)
> (define x (build-list 10 (λ (x) (read))))
1 2 3 4 5 f g h 9
> x
< '(1 2 3 4 5 f g h 9)
```

- What about lazy Racket?

```
(define Q (build-list 10 (λ (x) (read))))
```

It does not really read things. The value of `Q` is deferred.

- Only when we proceed to view the element of `Q`, `(read)` is called.
- However, before we do that, `(length Q)` gives 10! Racket can calculate a list of promises without evaluating each element.

28.3 Outputting Racket values: `display`

- We have already seen `display` in previous lectures. It also works with char types.

•

```
> (display #\x)
< x
> (display #\u03bb)
< λ
```

- We can use `write-char` to output a single char. `newline` to output the `#\newline` character.

- Racket has a special function called `begin`, which allows us to take any amount of displays (and any other side effects). With the above definitions, try the following:

```
(begin (display x) (display #\newline) (display y))
```

- Now, consider the following function:

```
(define (foo)
  (display x)
  (newline)
  (cond [true (display y) 42]))
```

When we call `(foo)`, we get the following display:

```
3
λ42
```

- Suppose that we have the following definition instead:

```
(define a (build-list 5 (λ (x) (read))))
```

How about the following interactions:

```
123 4 5 6 88
> a
< '(123 4 5 6 88)
> (map display a)
12345688'(#\<void> #\<void> #\<void> #\<void> #\<void>)
```

- Oops... we should have done the following instead (continuing interactions):

```
> (void (map (λ (x) (display x) (newline)) a))
< 123
< 4
< 5
< 6
< 88
```

28.4 IOStream.rkt

- The assignment provides a file called `IOStream.rkt`. It contains functions called `instream` and `outstream`.
- The function `instream` reads from input until `#\<eof>` (end of file) object is reached.

- `#\<eof>` has a predicate function `eof-object?`.
- `instream` is essentially an infinite list of `(read)`.
- The function `outstream` produces outputs from a stream (list).
- Suppose that `s` is a stream. If `s` is empty, `(outstream s)` produces `(void)`. Otherwise, it outputs all elements of `s`, one per line.
- Try the following: `(outstream instream)` and `(outstream (map add1 instream))`.

Lecture 29 (November 19, 2020)

The lecture talks about Assignment 11 and the modules available for it: `IOStream.rkt` and `Gen.rkt`.

29.1 Assignment 11 Version of Gen

29.1.1 Gen usage

- `Gen` in Assignment 11 works a bit differently with `stream-generate` from Assignment 9.
- `Gen` takes 3 parameter: `input`, `state`, and `step`. The parameter `step` itself must be a function that takes 3 inputs: `in`, `state`, and `cont`.
- It applies a continuation-passing style of programming. Here, our continuation simply stores the value in a list. Then, it passes back to `Gen` to do more generation. More about it can be read in the following webpage.

https://en.wikipedia.org/wiki/Continuation-passing_style

- An example:

```
(define x
  (Gen 0 1 ;;input and state
    (lambda (a b cont) ;;step
      (cont b (+ a b) (list b))))))
```

It outputs the first ten Fibonacci numbers.

- Another example:

```
(define (addlist inp)
  (Gen inp 0
    (lambda (inp sum cont)
      (if (empty? inp) empty
          (cont (cdr inp)
                (+ (car inp) sum)
                (list (+ (car inp) sum)))))))
```

```
> (outstream (addlist (list 1 10 20)))
< 1
< 11
< 31
```

- Since `cont` uses `append`, `step` can pass a list of more than one element to `cont`, so that the stream will just output multiple elements at once. It can also pass empty, so that `cont` will not output anything. As long as the list is of constant length, `append` will only take constant time.

29.1.2 Other functions and variables in `Gen.rkt`

- The function `history` takes a list as an input and produces a list of lists. For example, `(history '(a b c d))` produces:

```
'(() (a) (b a) (c b a) (d c b a))
```

- We could also make a version of `history` such that the resulting list from `(history '(a b c d))` does not produce `'(d c b a)`.

```
(define (history s)
  (Gen s empty
    (λ (s t cont)
      (if (empty? s) empty
          (cont (cdr s) (cons (car s) t (list t)))))))
```

- Try the following code:

```
(outstream (map list '(a b c d) (history '(a b c d))))
```

- The function `Flatten` is provided in `Gen.rkt`. It is built-in in regular (strict) Racket, but it is not provided in lazy Racket.
- More examples of using `Gen` to create streams:

1. `Trues`, an infinite stream of `true`s.

```
(define Trues
  (Gen 0 0
    (λ (inp state cont)
      (cont 0 0 (list true)))))
```

One can also use `(define trues (cons true trues))` or `(define trues (cycle true))`.

2. `mymap`, roughly the same as `map` for one stream.

```
(define (mymap f s)
  (Gen s 0
    (λ (stream state cont)
      (if (empty? stream) empty
          (cont (cdr stream) 0
                (list (f (car stream)))))))
```

3. **stutter**, repeats every element of the input stream. As an example, (**stutter** '(a b c)) produces '(a a b b c c). Note the last argument of **cont** is a list of two elements.

```
(define (stutter s)
  (Gen s 0
    (λ (stream state cont)
      (if (empty? stream) empty
          (cont (cdr stream) 0
                (list (car stream)
                      (car stream)))))))
```

4. **map2**: map for two streams. Both streams must be either finite with same length, or infinite.

```
(define (map2 f s t)
  (Gen s t
    (λ (inp1 inp2 cont)
      (if (empty? inp1) empty
          (cont (cdr inp1) (cdr inp2)
                (list (f (car inp1) (car inp2))))))))
```

5. **less**, oustreams 5 elements and asks if the user wants to continue oustreaming more elements. Each time the user continues, it oustreams 5 more elements and asks the same question. Stops when the user wants to stop or the stream is exhausted. The utility with the same name exists on most *nix systems.

```
(define (less s)
  (Gen s (list 5 instream)
    (λ (inp1 inp2 cont) ...)))
```

- Algorithm for the step on **less**:

- We represent the state as (**cons** *n* **instream**) for each step.
- If $n > 0$, then the next output is (**car** *s*), the next input is (**cdr** *inp*), and the new *n* value is $n - 1$.
- If $n > 0$, then the next output is "Continue? (y/n)", the next input is *inp*, and the new *n* value is -1 .
- If $n = -1$, the next step depends on user input. It does so by invoking (**cadr** **state**). If the user says no, **less** terminates.
- Otherwise, when $n = -1$ and the user says yes, the step sets *n* to 5 and calls the next step.

Lecture 30 (November 19, 2020)

This lecture continues lecture 29.

- Another function similar to `less` is `more`. In the original *nix system, `less` was written as an enhanced version of `more` (hence the saying “less is more”), that does not have anything to do with our version.
- `more` is just like `less`, except that you can specify an input stream and output stream.

30.1 Assignment 11 Part (b): Rock Paper Scissors

- `rps` produces a random infinite list of `'rock`, `'paper`, and `'scissors`.
- The function `rps-winner` is also provided, and determines who wins the round based on the input moves.
- A more useful function is `rps-game`, which shows the winner and what the two players plays.
- In Game theory, a “perfect player“ will always play randomly. This will always converge to a draw over all against any other player, given enough rounds are played. The problem is that true randomness cannot be achieved by any player, thus making it theoretically possible to dominate another player in RPS.
- In assignment 11, you can view the history of your opponent as a list from the most recent to the oldest. Your task is to program a player to consistently beat your opponent (wins more times then lose) given a large number of rounds.

Lecture 31 (November 23, 2020)

The lecture talks about a new model of computation: RAM. It also talks a bit about Assignment 12.

31.1 Introduction to RAM

- In lambda calculus, computations are done by α -conversions and β -reductions. Just like lambda calculus, Random Access Memory (RAM) is a model of computation.
- A theoretical RAM is an ADT equipped with two operations:
 - (`RAM-store R A N`): stores a value N at the address A in the RAM R .
 - (`RAM-fetch R A`): produces the value from address A in RAM R (it is an error if the address has not a value stored previously).

where the address is usually an natural number.

- A *Random Access Machine* (RAM) is another ADT, which is an abstract computational device that *uses* Random Access Memory. The RAM model of computation assumes
 1. `RAM-fetch` and `RAM-store` are $\mathcal{O}(1)$.
 2. $+$, $-$, $/$, \times are all $\mathcal{O}(1)$
- In the real world, these assumptions are false (as seen in A10). The best we can do is $\mathcal{O}(\log A)$.

31.2 Build a RAM Machine

We can build a RAM machine either in Racket as a program, or as a group of circuits.

31.2.1 Build RAM in Racket

- In order to implement the ADT efficiently, we need a new kind of tree: *trie*.
- A trie also lets you implement a set. Fetching a value stored in a trie takes $\mathcal{O}(\log A)$ time, where A is the address.
- In contrast, an AVL tree would have $\mathcal{O}(\log^2 A)$ times, because the total order is also not $\mathcal{O}(1)$ (most likely also $\mathcal{O}(\log A)$).
- In addition, all computation can be done with `Gen` provided in `Gen.rkt`, where the `state` is a RAM.

31.2.2 Build RAM Physically

Von Neumann Architecture

- Unlike the lambda calculus model, where the theory was created before computers were born. RAM machines were being built before the theory was formalized.
- John von Neumann was often credited as the creator of stored program computers.
- A stored program computer has
 - a RAM.
 - a Central Processing Unit (CPU), which the whole job is to run something equivalent to `Gen`
 - two devices for input and output, which sends and receives streams to CPU.
- Engineers cannot build a machine that meets the above requirement exactly. Namely, the RAM has to have a constant number of bits for the number and the address. For example, a RAM that uses 32 bits for address could store a value in between 0 and $2^{32} - 1$.

31.3 Assignment 12

- Assignment 12 provides `RAM.rkt`, which implements RAM. It is implemented as an unbounded RAM that stores non-negative integers.
- The function `core-dump` dumps out all values currently stored inside ram. For example, consider the following definition:

```
(define a (ram-store ram 10 20))
(define b (ram-store a 30 55))
(define c (ram-store b 33 88))
```

Then, try the following code:

```
> (core-dump b)
> (core-dump c)
> (ram-fetch c 10)
> (ram-fetch c 20)
> (ram-fetch c 30)
> (ram-fetch c 33)
```

- If you attempt to fetch an address that is not previously stored, it will return `'undefined`. You cannot rely on these behaviour to write your code, the RAM implementation Mar-moset uses might just crush the program when fetching an undefined value.

- A function that modifies RAM and directly uses recursion would likely follow the form:

```
(define (foo ram)
  :
  (foo ram1))
```

- We still work with **Gen** in Assignment 12, except that it is a bit different from Assignment 11.
- The rule of thumb for **Gen**: **inp** must be an input list, **state** must be a RAM, and **step** takes $O(1)$ running time without the **cont** application.

31.4 Representing Data in RAM

- To represent a non-negative integer, we can store it in one address of RAM. To represent two non-negative integers, we can simply store them in two places.
- To update a value, we can fetch, modify, then store back.
- To be able to represent all integers (including negative ones, there are some possible ways to store it. Suppose that x is an arbitrary integer.
 1. Two RAM locations, say address n and $n + 1$. We store $|x|$ at n , and $[n + 1]$ is 0 if $x \geq 0$ and 1 if $x < 0$.
 2. Two RAM locations, say address n and $n + 1$. We store a non-negative integer a at n , and b at $n + 1$ such that $a - b = x$.
 3. One RAM location. We store x as $2x$ if $x \geq 0$ and $-2x + 1$ if $x < 0$.

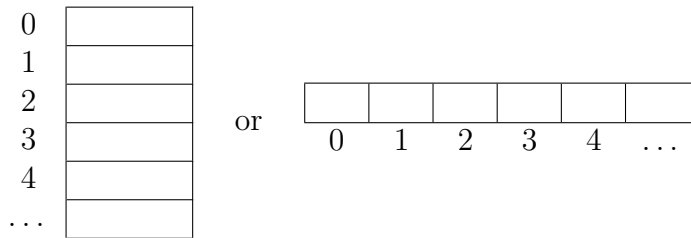
Lecture 32 (November 25, 2020)

The lecture talks about how we can implement RAM. It also covers CPU and how it works. Recall that:

- Although RAM is an ADT, we will only use one instance of RAM. Everything is stored in a single RAM.
- We use two notations for `ram-fetch` and `ram-store`:
 - $[n] := (\text{ram-fetch } \text{ram } n)$
 - $[n] = e := (\text{ram-store } \text{ram } n \ e)$ (= does not mean asserting $[n] = e$, but rather *making* $[n] = e$).
- In abstract RAM n and e can be any natural number, and fetching and storing are constant time operations. In real world there is usually a maximum value in the form of 2^b for each. b is usually 32 or 64, and fetching and storing are $\mathcal{O}(\log n)$ operations.

32.1 Ram Operations

- We draw a RAM as a vertical/horizontal table of a single column, indexed from the top starting at 0.



- After three operations

$[2] = 42$
 $[3] = 4$
 $[3] = 5$

Our ram becomes

		42	5		
0	1	2	3	4	...

Note that the effect of the second operation is overwritten by the third operation. Now 42 is store at 2, we write it as $[2] \equiv 42$ (to distinguish from store operation).

- We could also say $2 \equiv \text{foo}$, to mean the value of the variable `foo` is stored at index 2. This is essentially how one defines variables in RAM.
- Nested fetch and store is also possible: $[[4]] \equiv 42$. This is called indirection/pointer chasing.

32.2 Lists in RAM

There are several ways to build lists in RAM.

- One list:
 - If we only need one list, we use the “vector” representation (also called array).
 1. Pick an address in ram as the variable `lst`. (say `[42] ≡ lst`)
 2. Elements in the list is stored in adjacent indices starting from `lst+1` in reverse order. (43 stores the last element, 44 stores the second last element, etc)
 3. `lst` stores the number of elements in the list.
 - Thus, `[lst]` is `(length lst)` and `[lst+[lst]]` is `(car lst)`.
 - This implementation allows us to retrieve any element in the list in $\mathcal{O}(1)$.
 - `(cdr lst) ≡ [lst] = [lst] + 1`, and `(cons e lst)` can be achieved with `[lst+[lst]+1] = e then [lst]=[lst]+1`.
- N lists
 - If we need two lists `lst1` and `lst2`, we cannot just replicate the method for one list at two different ram locations. The list with smaller address can grow too big and cause an overrun. In fact, for N lists, the first $N - 1$ lists are always constrained in size.
 - To work around this, we need *variable sized vector/array*.
 - Whenever an overrun is going to happen (we can check by looking at the size of the lists), move everything in the lower list to some location after the end of the other list(s).
 - This approach is $\mathcal{O}(n)$ where n is the length of the list. In the worst case every use of `cons` involves a copy.
 - We can do better by reserving space double the length of the list when we make a copy. This way, the total amount of copy operation is linear in the size of the list.
 - `cons` will not be $\mathcal{O}(1)$ every time, but it will be $\mathcal{O}(1)$ on average. This is known as *amortized analysis*
 - Another problem with this approach is that data tends to move towards higher indices, leaving empty space in the low end. In a real world computer we need to do *garbage collection* to reclaim those wasted space.

32.3 Linked-lists

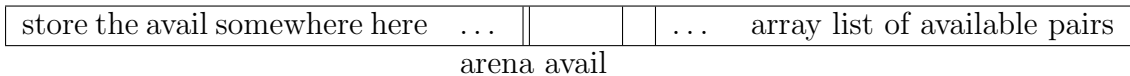
- Linked list more closely resembles lists in Racket.
- To create a linked list

1. Choose a location for `lst` that is not 0.
2. `[lst] = 30`
3. `lst+1` will be a *pointer* to the second element of the list.
4. If there is no next element, `[lst+1] = 0`.

- Thus

```
[lst] ≡ (car lst)
[lst+1] ≡ (cdr lst)
(zero? lst) ≡ (empty? lst)
```

- Note that this way we can take an arbitrary element in the list and treated it as the head of the list, comparing to what has to be done in the vector implementation (taking the `cdr` destroys the current element).
- With this implementation we can also create as many lists as we want, all we have to do is find some unused space and start to put values in. `cons` has to
 1. Find some unused location.
 2. Put the value in that location.
 3. Put the index of the next element beside that location.
- The only problem is that we don't know which locations are used or not.
- The solution is to use another pointer, an “arena avail”, which points to the next available pair space.



- Every time `cons` asks for the space of another pair, put the values at the avail's location, and increases the pointer's value by 2. Of course, you have to store the avail at a separate place, somewhere below the address it is pointing to.

Lecture 33 (November 25, 2020)

This lecture talks about Assignment 12, especially part d-g

- For A12 part d-g, ram is also the only data structure that can be used, but we don't use the operations directly.
- In addition, similar to a stored-program computer, the program itself is stored in the ram.
- The CPU we use would be the generating function, which uses `cpu.rkt`, which implements the programming language we call the machine language.
- This minimalist CPU provides 10 instructions, 7 elementary operations, and 3 for IO and core dump. (See `cpu.rkt` for documentation).
- The CPU works as follows: it first uses a loader (in real world computers, this is often called a bootstrap) to load a list as the consecutive locations in the ram starting from `[0]`.
- The location `0` is a special place that stores the IPA (Instruction Pointer Address). The CPU executes the instruction stores at `[0]` repeatedly and increment `[0]` by 1, until it gets the instruction `core-dump`. Every such execution is called a *cycle*.
- Since `0` is part of the RAM, we can also modify the IPA to skip or repeat execution of certain addresses.
- An obvious limitation of computers with RAM is that recursion is not supported. Tail-recursion is easy to do, but general recursion is harder.

Lecture 34 (November 26, 2020)

This lecture talks about how we can use a trie to implement RAM.

- In general, a trie is a kind of tree, which can act as a set. The set's elements are finite sequences of a certain finite set. In this course we are mainly concerned with binary tries, which contains finite sequences of 0 and 1's.
- The difference between binary trie and binary tree is that a BST requires a total order, and inserting an element takes $\mathcal{O}(\log n)\mathcal{O}(f(n))$ where n is the size of the tree and f is the total order comparison function. Inserting an element into a binary trie is $\mathcal{O}(m)$, where m is the length of the sequence.
- A binary trie is either empty or a node, where its left and right are subtries. The node has a field called happiness. The happiness is set to **true** if the corresponding non-negative integer is there, and **false** otherwise.
- We assign a node with a non-negative integer depending on its binary representation. We start with the root node.
- For example, consider the integer 10, which is 0101 in reversed binary representation. The path from the root to the node corresponding to 10 is left, right, left, right.
- In general, suppose that n is a non-negative integer. Then, the left and right direction of the path corresponds to digit 0 and 1, respectively, in the reversed binary representation of n .
- More examples:

n (decimal)	n (reverse binary)	Path to corresponding node
2	01	left, right
5	101	right, left, right
8	0001	left, left, left, right

- To insert a non-negative integer into the trie, we recurse by dividing the integer by 2 repeatedly until it reaches 0.

- Code for trie insertion:

```
(define-struct ndoe (l r happy))
(define (trie-insert e t)
  (cond
    [(empty? t) (trie-insert e (node empty empty false))]
    [(zero? e) (make-node (node-l t) (node-r t) true)]
    [(even? e)
     (make-node (trie-insert (quotient e 2) (node-l t))
                 (node-r t))
     (node-happy t)]
    [true
     (make-node (node-l t)
                 (node-happy t)
                 (trie-insert (quotient e 2) (node-r t))))])
```

- The implementation for deletion is similar to insertion. However, we might need to cut a branch that contains no happy node anymore. This is called **pruning**.
- We can also further decorate binary trie to make a (key, value) pair, similar to what was done with BSTs in assignment 9. The key is the sequence.
- For RAM implementation, we can replace happiness by the number itself. The key is the address, and the value is the number stored in that address. For detail, see `RAM.rkt`.

Lecture 35 (December 3, 2020)

The final recorded lecture covers history relevant to the course. Topics include how to build a computer, the historical development of the computer, and physical implementations of RAM.

- *Digital* means expressed as a series of digits.
- Problem: Most things in our world are continuous. They are analog quantities that cannot be recorded digitally.
- Digital information can be represented with analog systems as below.
- Example: Glasses of Cola can be used as bits.
- Example: Electronic switches are designed to be either on or off.
- Example: Flashlights can be switched on and off.
- Different methods of encoding and processing information are discussed.
- Example: Information could be encoded using the punch cards of Jacquard and Hollerith of IBM.
- Example: Telephones and their switchboards present a ripe opportunity for information processing.
- Different physical implementations of RAM are discussed including:
 1. Delay Lines
 2. Cathode Ray Tubes (CRT)
 3. Magnetic Core Memory (Core)
 4. Electronic Memory
- Observation: All practical RAMs limit the address and the value. All implementations require more time and space for larger values.

Here is a list of people discussed or mentioned:

- Joseph Marie Jacquard
- Herman Hollerith
- Charles Babbage
- Ada Lovelace
- Claude Shannon

The following figures, who have been mentioned in past lectures, may be relevant to the historical development of computers and the computing technology used in this course:

- Alonzo Church
- Haskell Curry
- John Von Neumann
- Alan Turing

The following concepts are relevant to the historical development of the computer:

- Jacquard Loom
- Repeater
- Hollerith Cards
- Difference Engine
- Analytic Engine
- Stored-program computer
- Telephones and switchboards
- Boolean Algebra
- AT&T Bell Labs
- Switching
- Delay Line
- Cathode Ray Tube (CRT)
- Magnetic Core (Core)
- Register Machine
- Electronic Memory
- Dynamic and Static RAM
- Flip-Flop

Lecture 36 (November 28, 2020)

The lecture talks about history of computers.

Below is a list of peoples and things to read (taken from Piazza).

People:

- Alonzo Church
- Haskell Curry
- John Von Neumann
- Claude Shannon
- Alan Turing
- Herman Hollerith
- Joseph Marie Jacquard
- Charles Babbage
- Ada Lovelace

Things:

- Analytical Engine
- Flip flop
- Dynamic RAM and static RAM
- Core
- CRT memory
- Delay line memory
- Telephone switching
- Register machine
- Turing machine
- Combinator theory
- Boolean algebra
- Stored program computer
- CPU