1.1

Simple ideas → Complex ideas

- primitive expressions
- combinations : compound elems built from simpler ones.
- abstraction : compound elems named & manipulated
- data : stuff to be manipulated
- function : rules for manipulating data.

Notes: round off & truncation errors.

eg. 11/3 ≃ 3.6666666 ≦ etc.

Compound expressions - eg. 10 * 10. (combination)

↳ infix notation

interpreter → reads → evaluates → prints. (loop)

⇓

not necessarily to explicitly instruct print.

Javascript → every statement has a value.

Const size = 2 → constant value is object.

↓

name identifies a constant

Constant declaration → abstraction.

Program environment → memory to store name-obj. pairs.

↳ To evaluate operator combination

recursive

1. evaluate operand expressions of combination
2. Apply function to arguments that are values of operands.

- Terminal nodes (leaf) → either operators or numbers.
- percolate values upwards ∈ tree accumulation.
- const (book) → keywords (cannot be used as name)
  ↳ own evaluation rule.

Necessary elems:

1. Numbers & arithmetic operations are primitive data & func.
2. Nesting of combinations → means of combining operations
3. Constant declarations that associate names w values provide a limited means of abstraction.

Function declaration

- Compound operation → given a name & referred as unit.
- eg. function square (x) {
  return x * x; (body)
  }
  name, parameter, expression.

- General form:

  function-expression (argument-expressions).

  - To evaluate a function:
    1. evaluate the subexpressions of the application, function expression and argument expressions.
    2. Apply the function that is the value of the function expression to values of the argument expressions.

  - Can be used as a building block in def other fns

- Compound functions
  - To apply to arguments, evaluate the body expression of the function w each parameter replaced by corresponding argument.
  - Substitution model - determines "meaning" of function application
    - Help us think about fns application
    - Use local env for parameters
    - Values substituted for parameters in argument fns.

Applicative order vs Normal order

- Interpreter first evaluates function & arg. exp. then applies resulting fns to resulting args. } Applicative order (Calcnal)
- Do not evaluate the args until values are needed — Put sub arg exp for parameters until it obtained an exp involving only operators and primitive fns. } Normal order

more efficient, avoid multiple evaluations
less complicated

Conditional expressions.

- eg. function abs(x) {
  return x >= 0 ? x : -x;
  }

- predicate ? consequent-expression : alt-expression // ternary operator
  ↓
  true / false

- eg2. function abs(x) {
  return x > 0
  ? x x if x > 0
  : x === 0
  ? 0 0 if x = 0
  : -x; -x otherwise.

- Synthetic form is right-associative

- P_1 evaluate P_1
  ? e_1 if true e_1
  : P_2 else evaluate P_2
  ? e_2 if true e_2
  : else ...
  : P_n else evaluate P_n
  ? e_n if true - e_n
  : final-alt-exp. else

- Logical composition operations
  - e_1 && e_2 → and / conjunction
    synthetic sugar → e_1 ? e_2 : false.
  - e_1 || e_2 → or / disjunction
    synthetic sugar → e_1 ? true : e_2.
  - !e → not / negation
    - unary operator (one arg)
      prefix operator

synthetic forms not operator ↓ right hand not always evaluated.

* synthetic form / sugars are
Convenient alt structures for things that can be written in more uniform ways.

- precedence
  lowest
  ...?...:..., && , >< , binary operators , unary operator.

- note: a===b / a!==b are arguments.
  returns true or false

Exercise 1.1

10

12

8

3

6

a=3 (undef)

b=4 (undef)

19

false

true && true ? 4 : 3

16

6

16

//

Ex 1.2

```
(5+4+(2-(3-(6+(4/5))))) / (3*(6-2)*(2-7));
```

Ex 1.3

```
function f(a,b,c){
    return (a>c && b>c)?
        : a*a+b*b
        ? (b>a && c>a)
        : b*b+c*c
        ? (a>b && c>b)
        : a*a+c*c
```

Ex 1.4

If b >= 0, then func return a+b

else b<0, then func return a-b.

```
a_plus_abs_b(a,b){
    return (b>=0 ? plus : minus)(a,b);
}.
```

Ex 1.5.

```
test(0,p()){
    return 0 === 0 ? 0 : p();
}
```

If use applicative order, p() will evaluate first ⇒ stack overflow error

else if normal order, p() will not be evaluated ⇒ test return 0.

- Computer functions vs Mathematical functions
  - Computer functions must be effective.

- Eg. Newton method for sqr roots.

  $\sqrt{x}$ = the y, $y \geq 0$ and $y^2 = x$

  - not a computer function

Pseudo / Javascript

```
function sqrt(x) {
    return the y with y>=0 && square(y) === x;
}
```
(nothing else how to find the number).

- Contrast:
  - Reflection of general distinction btw describing properties of things & describing how to do things.
  - also btw declarative & imperative knowledge.
    declarative: math. imperative: CS

- Newton method of successive approx.
  - guess y for value of sqrt x,
    perform simple manipulation for a better guess
    by averaging y with $\frac{x}{y}$.
  - eg. $\sqrt{2}$, initial guess = 1.

Guess	Quotient	Average
1	$\frac{2}{1} = 2$	$\frac{(2+1)}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{(1.3333+1.5)}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{(1.4167+1.4118)}{2} = 1.4142$
1.4142	...	...

  - In terms of function, let x be constant,

```
function sqrt_iter(guess, x) {
    return is_good_enough(guess, x)
        ? guess
        : sqrt_iter(improve(guess, x), x);
}
```

usually gives preference regarding starting with if.

tail recursion (efficiency)

```
function is_good_enough(guess, x){
    return abs(square(guess) - x) < 0.001;
}
```

```
function improve(guess, x) {
    return average(guess, x/guess);
}
```

```
return (x+y)/2;
```

Ex 1.6

```
function conditional (predicate, then_clause, else_clause) {
    return predicate ? then_clause : else_clause;
}
```

is_good_enough returns true/false depending on the approximation of the guess and $\sqrt{x}$.
If true, it will return the guess value, else
if false, it will call itself using a improved guess value.
Any call of sqrt_iter will lead to infinite loop.
Since applicative_order, it will evaluate return expression of sqrt_iter args. Some will happen to recursive calls.

Ex 1.7.

is_good_enough test = abs(sqr(guess) − x) < 0.001 absolute tolerance.

For very small numbers, eg. sqrt(0.0001) ≈ 0.032130...
interval if expected 0.01 → error of over 200%

For very large numbers, rounding error might make algorithm fail to get close to square root, then program will not terminate.

```
const error_threshold = 0.01;
function is_good_enough(guess, x) {
    return relative_error(guess, improve(guess, x))
        < error_threshold;
}
function relative_error(estimate, reference) {
    return abs(estimate - reference)/reference;
}
```

Ex 1.8

Newton method for cube roots, $y \approx \sqrt[3]{x}$,
then better approximation, $\frac{x/y^2 + 2y}{3} \approx \sqrt[3]{x}$.

```
function cube_iter(guess, x) {
    return is_good_enough(guess, x)
        ? guess
        : cube_iter(improve(guess, x), x);
}
```

const error_threshold = 0.01.

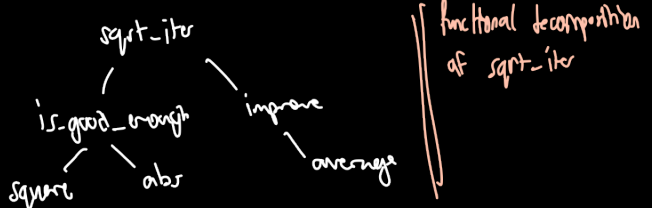
```
function is_good_enough (guess, x) {
    return relative_error ( cube(guess), x)
            < error_threshold
}

function relative_error (est, ref) {
    return abs( est - ref) / ref
}

function improve ( y , x) {
    return (x / (y*y) + 2y) / 3
}
```

Recursive

- sqrt_iter function
  - ↳ breaks up into sub-problems.
    - 1. how to tell if guess is good enough? } each task accomplished by a separate func.
    - 2. how to improve on guess
    - etc.



functional decomposition of sqrt_iter

- each function accomplishes an identifiable task.
- can be used as module in def other funcs.
- eg. function is_good_enough (guess, x) {
  return abs(square(guess) − x) < 0.001;
  }

  ↑ abstraction of function;
  functional abstraction:
  details can be suppressed, considered at a later date.

- function → suppresses details (do not need to know it to apply it).

Local names

- choice of name of parameter should not matter for implementation.
- names of a function must be local to the body of func.
- else would be confused.
- The func declaration binds its parameter ⇒ bound. (body of func or scope)
- Otherwise name is free

Block structure:

function sqrt(x) {
  function is_good_enough
  function improve
  function sqrt_iter
  return sqrt_iter(1, x);
}

localizes the subfunctions
allows a func to have internal declarations to prevent clashes with other auxiliary functions in the programme.

Better block structure (lexical scoping):

Since x is bound in declaration of sqrt,
not necessary to pass x explicitly to each of the functions.
Instead allow x to be a free name.

```
function sqrt(x){
    function is_good_enough(guess)
    function improve(guess)
    function sqrt_iter(guess)
    return sqrt_iter(1);
}
```

- Block structure helps break up large programs into tractable pieces. (Decomposition)

- Lexical scoping dictates that free names in a function are taken to refer to bindings made by enclosing function declarations.
  - i.e. they are looked up in the env in which function was declared.

1.2

- The ability to visualise the consequences of actions under consideration is crucial
- We are planning the course of action to be taken by a process
- and we control the process by means of a program

- A Function is a pattern for local evaluation of a computational process
  - specifies how each stage is built upon prev. stages
  - make statements about overall, global behaviour of a process

Linear Recursion & Iteration

$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
$= n \cdot (n-1)!$
$= n \cdot (n-1) \cdot (n-2)! \cdots$

```
function factorial (n) {
    return n === 1
        ? 1
        : n * factorial (n-1);
}
```

recursive function

function declaration refers to itself (directly / indirectly)

In substitution model:

expansion – built up of deferred operations
- factorial (3)
- 3 * factorial (2)
- 3 * (2 * factorial (1))
- 3 * (2 * 1)
- 3 * (2)
- 6

contraction – operations are actually performed

chain of deferred operation

linear recursive process.

recursive process

- to carry out requires interpreter to keep track of the operations to be performed later on.
- Example – length of chain of deferred multiplication, which is the amt. of info to keep track on, grows linearly with n (is proportional to), just like number of steps.

Using iteration:

rule for computing n!:
1. Multiply 1 by 2,
2. Then, multiply the result by 3
3. then, by 4
⋮
n−1. then, by n.

maintained a running product and a counter that counts from 1 to n.

Counter & Product simultaneously change from one step to the next according to the rule:

new vals: product ← counter · product
counter ← counter + 1
} old vals.

and n! is the value of the product when counter exceeds n.

Contrast:

Iterative
- The state variables provides a complete description of state of process at any point.
- ∴ If suddenly terminated we can easily resume with the values of the state variables

Recursive
- additional hidden info not in state variables.
- indicates where the process is in the chain of deferred operations.
- the longer the chain, the more info it maintains

process does not grow and shrink

```
function factorial (n) {
    return fact_iter (1, 1, n);
}
function fact_iter (product, counter, max_count) {
    return counter > max_count
        ? product
        : fact_iter (counter * product,
                     counter + 1,
                     max_count);
}
```

also recursive function

each itr the pdts & count changes.

fact_iter (3)
fact_iter (1, 1, 3)
fact_iter (1, 2, 3)
fact_iter (2, 3, 3)
fact_iter (6, 4, 3)
6

keeps track of pdts, counter and max_count.

linear iterative process for 3!

num of steps is also proportional to 1.

but process is iterative – state is captured by 3 state variables.
(keep track of only 3 numbers in order to proceed)

In general, an iterative process is one where state can be summarized by a fixed number of state variables, together with a fixed rule that describes how the state variables should be updated as the process moves from to state and an optional end test that specifies conditions under which process should terminate.

* most implementation languages (C, C, java, python)
  are designed to consume memory that grows with number of function calls.
  - does not matter if the process is iterative.
  - these languages can describe iterative process by resorting to special purpose looping constructs (C do, repeat, until, while, for).
  - Javascript does not show this defect, will execute an iterative process in constant space, even if the iterative process is describe by a recursive function.
    O(1)

tail recursive implementation
- iteration can be expressed using ordinary function call mechanism.
- special iteration constructs are useful only as syntactic sugar.
  (care of reserving syntax.)

Ex 1.9

1st model:

```
plus(4,5)
inc(plus(3,5))
inc(inc(plus(2,5)))
inc(inc(inc(plus(1,5))))
inc(inc(inc(inc(plus(0,5)))))
inc(inc(inc(inc(5))))
inc(inc(inc(6)))
inc(inc(7))
inc(8)
9
```

∴ recursive process.

2nd model:

```
plus(4,5)
plus(3,6)
plus(2,7)
plus(1,8)
plus(0,9)
9.
```

∴ iterative process.
since number of variables are fixed.

Ex 1.10.

```
function A(x,y){
    return y===0
        ? 0
        : x===0
        ? 2*y
        : y===1
        ? 2
        : A(x-1, A(x,y-1));
}
```

A(1,10) = 1024
A(2,4) = 65536
A(3,3) = 65536
f(n) = 2n
g(n) = A(1,n)
= A(0, A(1,n-1))
= 2 * A(1,n-1)
= 2 * A(0, A(1,n-2))
= 2 * 2 * A(1,n-2)
⋮
2^n

h(n) = A(2,n)
= A(1, A(2,n-1))
= A(1, A(1, A(2,n-2)))
= A(1, A(1, ... A(1, A(2,1))...
= A(1, A(1, ... A(1,2))...
(n)
= A(1, A(1, ... A(0, A(1,1)))...
$2^{2^{2^{\cdots}}}$ (n)
2^4 A(1,1)
2 * 2.
4?
$= 2^{2^{2^{\cdots}}}$

Tree recursion

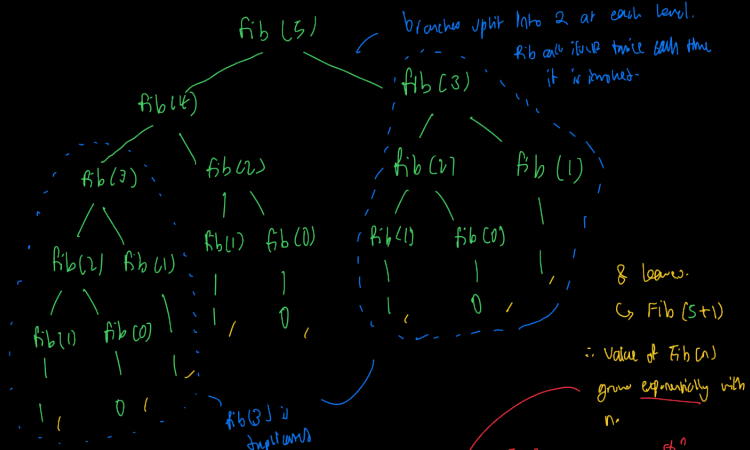
Eg. Fibonacci No. (Sum of nos of preceding 2)

1, 1, 2, 3, 5, 8, 13, 21, ...

Maths definition:

$$Fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

```
function fib(n) {
  return n === 0
    ? 0
    : n === 1
    ? 1
    : fib(n-1) + fib(n-2);
}
```



branches split into 2 at each level.
fib calls itself twice each time it is invoked.

8 leaves.
↳ Fib(5+1)

∴ Value of Fib(n) grows exponentially with n.

fib(3) is duplicated
∴ redundant calc.

From Maths: $Fib(n) \approx \frac{\phi^n}{\sqrt{5}}$
where $\phi = \frac{1+\sqrt{5}}{2}$ (golden ratio)
and $\phi^2 = \phi + 1$.

General

Num of steps req by tree recursive process ∝ num of nodes in tree

Space req. ∝ max depth of tree

1. Process / number of steps grows exponentially with input.
2. Space required grows linearly, only kept track of nodes above us in tree at any time.

Iterative process for fib nums.

a = 1, b = 0

Itr
a ← a+b
b ← a

```
function fib(n) {
    return fib_iter(1, 0, n);
}
function fib_iter(a, b, count) {
    return count === 0
        ? b
        : fib_iter(a+b, a, count-1);
}
```

* To formulate this iterative algo, one need to notice the computation can be recast as an iteration w/ 3 state vars.

|| linear iteration
Chinese steps wrt n)
∴ less steps than tree.

- tree recursion
  - useful in helping us to understand and design programs.
  - more straightforward even though less efficient

Eg. Counting Change.

To compute the num of ways to change any given amt of money

- Simple soln: recursive tree.

Num of ways to change amount = a.

Num of kinds of coin = n.

- no. of ways to change amt. a using all but first kind of coin. (divide into 2 groups.)
- no. of ways to change amt. a−d using all n kinds of coins,

d = denomination of first kind of coin.

∴ Total no. of ways to make change for same amt. = no. of ways to make change for the amt. w/o using any of first coin (using fewer kind of coins) + no. of ways to make change assuming that we do use the first kind of coin.

||

no. of ways to make change for the amt that remains after using a coin of that kind

(changing smaller amts)

Specifying the following degenerate cases:

- If a == 0, we should count that as 1 way to make change.
- If a < 0, we should count that as 0 ways to make change.
- If n is 0, we should count that as 0 ways to make change.

```
function count_change (amount) {
    return cc (amount, 5);
}

function cc (amount, kind_of_coin) {
    return amount === 0
        ? 1
        : amount < 0 || kinds_of_coin === 0
        ? 0
        : cc (amount, kinds_of_coin - 1)
        +
        cc (amount - first_denomination (kinds_of_coin),
            kind_of_coin);
}

function first_denomination (kinds_of_coin) {
    return kinds_of_coin === 1 ? 1
        : kinds_of coin === 2 ? 5
        : kinds_of coin === 3 ? 10
        : kinds_of coin === 4 ? 25
        : kinds_of coin === 5 ? 50
        : 0;
}
```

generates a tree-recursive process with redundancies.

Although inefficient, it is easy to specify and understand.

One way to cope with redundant computations is to arrange matters so we automatically construct a table of values as they are computed.

∴ each time when we to apply function
↳ check if value is stored in table
(tabulation / memoization)

takes as input
no. of coins available.
returns denomination of first kind

Ex 1.11

$f(n) = n, \quad n < 3$

$f(n) = f(n-1) + 2f(n-2) + 3f(n-3), \quad n \geq 3$

Write a recursive & iterative process for f.

Recursive:

```
function f(n) {
    return n >= 3
        ? f(n-1) + 2f(n-2) + 3f(n-3)
        : n;
}
```

Iterative; (hard to come up with).

```
function f(n) {
    function iter_f (a, b, c, count) {
        return count === 0
            ? a
            : iter_f (a + 2*b + 3*c, a, b, count - 1);
    }
    return n < 3
        ? n
        : iter_f (2, 1, 0, n-2);
}
```

Ex 1.12:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

— elements of pascal triangle is the binomial coefficient. because nth row consists of coefficient of the terms in the expansion of $(x+y)^n$.

```
function pascal-element (row, index) {
    return index > row
        ? false
        : index === 1 || index === row
            ? pascal-element(row-1, index-1) + pascal-element(row-1, index)
            : 1;
}
```

check condition

Ex 1.13

prove $Fib(n)$ closest int to $\frac{\phi^n}{\sqrt{5}}$, $\phi = \frac{1+\sqrt{5}}{2}$

* use induction & defn of fibonacci numbers.

First, $Fib(n) = \frac{\phi^n - \psi^n}{\sqrt{5}}$ where $\psi = \frac{1-\sqrt{5}}{2}$ using strong induction.

$Fib(0) = 0, \frac{\phi^0 - \psi^0}{\sqrt{5}} = 0.$

$Fib(1) = 1, \frac{\phi^1 - \psi^1}{\sqrt{5}} = 1$

So statement is true for $n = 0, 1$. Given $n \geq 1$, assume proposition to be true for $0, 1, 2, \ldots, n \ldots$

$$Fib(n+1) = Fib(n) + Fib(n-1) = \frac{\phi^n - \psi^n + \phi^{n-1} - \psi^{n-1}}{\sqrt{5}}$$

$$= \frac{\phi^{n-1}(\phi+1) - \psi^{n-1}(\psi+1)}{\sqrt{5}}$$

$$= \frac{\phi^{n-1}(\phi^2) - \psi^{n-1}(\psi^2)}{\sqrt{5}} = \frac{\phi^{n+1} - \psi^{n+1}}{\sqrt{5}}$$, so statement is true.

Notice that since $|\psi| < 1$ and $\sqrt{5} > 2$, one has $\left|\frac{\psi^n}{\sqrt{5}}\right) < \frac{1}{2}$

Thus integer closest to $Fib(n) + \frac{\psi^n}{\sqrt{5}} = \frac{\phi^n}{\sqrt{5}}$ is $Fib(n)$.

Order of growth.

- rates at which a process consumes computational resources.
- a gross measure of the resources required by a process as the inputs become larger.
- In general, there are a number of properties of the problem with respect to which it will be desirable to analyse a given process.
- In computers that do only a fixed number of operations at a time, the time required will be proportional to the number of elementary machine operations performed.

- $R(n)$ has an order of growth $\Theta(f(n))$
  - ↳ $R(n) = \Theta(f(n))$ where $R(n)$ is amt. of resources process requires for a problem of size n, n is a parameter that measures the size of the problem.

$k_1 f(n) \leq R(n) \leq k_2 f(n)$ for any sufficiently large value of n, k_1 & k_2 are positive constants.

- For linear recursive process,
  number of steps ∝ input
  ∴ steps required for process : $\Theta(n)$.
  space required for process : $\Theta(n)$.

  For iterative factorial,
  steps - $\Theta(n)$,
  space - $\Theta(1)$, constant.

Eg. a process req. n^2 steps, a proce req. $1000n^2$ steps,
a process req. $3n^2 + 10n + 17$ steps → $\Theta(n^2)$
∴ crude description.

Order of growth is a useful indication for how we expect the behaviour of process to change as we change the size of the problem.

For $\Theta(n)$ process, double size = double resource used

For exponential proces, each increment in size = multiply resource utilization by constant factor.

Ex 1.14. $\sin x \approx x$ if x is sufficiently small.
$\sin x = 3\sin\frac{x}{3} - 4\sin^3\frac{x}{3}$

```
function cube (x) {
    return x * x * x;
}
function p (x) {
    return 3 * x - 4 * cube (x);
}
function sine (angle) {
    return ! (abs (angle) > 0.01)
        ? angle
        : p (sine (angle / 3));
}
```

1. When $x = 12.15$,
   p is applied 5 times.

2. time $\Theta(\log(x))$ base of log is immaterial
   space $\Theta(\log(x))$ For order of growth as
   since in each recursive call, log of diff bases
   angle is divided by 3. differ by a constant
   factor.

Exponentiation

- problem of computing the exponential of a number.
- $b^n = b \cdot b^{n-1}$, b as arguments, n as positive int exponent.
  $b^0 = 1$
- function expt(b, n) {
  return n === 0
  ? 1
  : b * expt(b, n-1);
  }. linear recursive process, O(n) time and space.
- function expt(b, n) {
  return expt_iter(b, n, 1);
  }
  function expt_iter(b, counter, product) {
  return counter === 0
  ? product
  : expt_iter(b, counter-1, b * product);
  }. linear iterative process, O(n) time and O(1) space.
- rather than doing $b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$,
  we can do
  $b^2 = b \cdot b$
  $b^4 = b^2 \cdot b^2$
  $b^8 = b^4 \cdot b^4$ (using 3 multiplications).
  we can combine them:
  $b^n = (b^{n/2})^2$ if n is even
  $b^n = b \cdot b^{n-1}$ if n is odd.
  } to test: function is_even(n) { return n % 2 === 0; }
  function fast_expt(b, n) {
  return n === 0
  ? 1
  : is_even(n)
  ? square(fast_expt(b, n/2))
  : b * fast_expt(b, n-1);
  }
  space: O(log n), time: O(log n).

└ Observe b^{2n} using fast_expt req. one more multiplication than b^n.
∴ size of exponent can double (approx.) with every new multiplication.
∴ no. of multiplication req. for an exponent of n grows as fast as log n to base 2.

- $\Theta(n)$ and $\Theta(\log n)$ growth diff becomes large when $n \to \infty$.
  eg. for fast_expt $n = 1000$ requires 14 multiplications

- successive squaring → device itr algorithm that computes exponentials with a logarithmic number of steps

Ex 1.15.
  iterative exponential process that uses successive squaring and a logarithmic number of steps.
  * $(b^{n/2})^2 = (b^2)^{n/2}$
  - additional state variable a.
  - state transformation that product ab^n is unchanged from state to state.
  - $a = 1$ at the beginning
  - ans = a at the end.
  - Technique of stating an invariant quantity that remains unchanged from state to state → powerful way to design iterative algorithm.
  - function fast_expt_iter (a, b, n) {
    return n === 0
    ? a
    : is_even(n)
    ? fast_expt_iter (a, b * b, n/2)
    : fast_expt_iter (a * b, b, n-1);
    }
  - function fast_expt (b, n) {
    return fast_expt_iter (1, b, n);
    }

Ex 1.16. repeated multiplication.
  ⇒ integer multiplication by means of repeated addition.

```
function times (a, b) {
    return b === 0
        ? 0
        : a + times (a, b - 1);
}
```
Steps: $\Theta(n)$

```
function double (a) {
    return a * 2;
}
function halve (a) {
    return a / 2;
}
```

Suppose function double doubles an integer,
halve divides integer by 2
∴ design a multiplication function that is analogue to fast_expt and uses $\Theta(\log n)$ steps.

```
function fast_times (a, b) {
    return b === 1
        ? a
        : a === 0 || b === 0
        ? 0
        : is_even (b)
        ? double ( fast_times (a, halve (b)))
        : a + fast_times (a, b - 1);
}
```

Iterative process for multiplying 2 integers (in term of adding, doubling & halving and $\Theta(\log n)$ steps).

```
function fast_times_iter (total, a, b) {
    return b === 1
        ? total + a
        : a === 0 || b === 0
        ? 0
        : is_even (b)
        ? fast_times_iter (total, double (a), halve (b))
        : fast_times_iter (total + a, a, b - 1);
}
```

Ex 1.18.

Compute Fibonacci No. in $\theta(\log n)$ steps

```
function fib_iter (a, b, p, q, count) {
    return count === 0
        ? b
        : is_even(count)
        ? fib_iter(a,
                   b,
                   p * p + q * q,
                   2 * p * q + q * q,
                   count / 2)
        : fib_iter(b * q + a * q + a * p,
                   b * p + a * q,
                   p,
                   q,
                   count - 1);
}
```

Greatest Common Divisor

- the largest integer that divides both a and b with no remainder.
- used to reduce rational numbers to lowest terms.
  (divide both numerator & denominator by their GCD.)
- Factor 2 integers to find GCD and search for common factors.
- more efficient algo?
- if r is the remainder when a is divided by b, then the common divisors of a and b are same as b and r.

  ∴ GCD(a, b) = GCD(b, r).
- to reduce computing of GCD to smaller pair of integers.

Euclid's algo:

eg. GCD(206, 40) = GCD(40, 6)
= GCD(6, 4)
= GCD(4, 2)
= GCD(2, 0)
= 2.

possible to show that starting w/ any 2 +ve integers and performing repeated reduction will always produce a pair with second number is 0.

```
function gcd(a, b) {
  return b === 0
    ? a
    : gcd(b, a % b);
}
```

∴ iterative, steps = $\Theta(\log n)$.

In relation to fibonacci numbers:

Lamé's theorem:
If euclid's algo req. k steps to compute the gcd of some pair, then smaller number in the pair must be greater than or equal to the kth Fibonacci number.

Let n be smaller input, if process takes k steps, then $n \geq Fib(k) \approx \phi^k/\sqrt{5}$

∴ k grows as log (to base ϕ) of n.

Ex 1.19 when gcd uses normal-order evaluation, using substitution method,
fewer remainder operations.
in applicative order,
4 remainder operations.

> read 1.19 for illustration.

Testing for primality

- checking primality of integer n
- Searching for divisors:
  - One way to test if n is prime is to find n's divisors

```
function find_divisor(n, test_divisor){
    return square(test_divisor) > n
        ? n
        : divides(test_divisor, n)
        ? test_divisor
        : find_divisor(n, test_divisor+1);
}
```

```
function divides(a, b){
    return b % a === 0;
}
```

1. finds smallest integral divisor (greater than 1) of a given number n.
   - testing n for divisibility by successive integers starting at 2.

To test if prime:

```
function is_prime(n){
    return n === smallest_divisor(n);
}
```

→ end test: if n is not prime, it must have a divisor ≤ $\sqrt{n}$.

∴ algo only need to test divisor between 1 and $\sqrt{n}$.

∴ steps: $\Theta(\sqrt{n})$.

- fermat test:

- If n is a prime number and a is any positive integer less than n, then a raised to the nth power is congruent to a modulo n.

```
function expmod(base, exp, m){
    return exp === 0
        ? 1
        : is_even(exp)
        ? square(expmod(base, exp/2, m)) % m
        : (base * expmod(base, exp-1, m)) % m;
}
```

- uses successive squaring.
  ∴ steps: $\Theta(\log exp)$

- computes exponential of a number % another number.

```
function fast_is_prime(n, times){
    return times === 0
        ? true
        : fermat_test(n)
        ? fast_is_prime(n, times-1)
        : false;
}
```

2. true if succeeds everytime.

```
function fermat_test(n){
    function try_it(a){
        return expmod(a, n, n) === a;
    }
    return try_it(1 + math_floor(math_random() * (n-1)));
}
```

- choose random a btw [1, n-1].
  - check if $a^n \equiv a$.

— Probabilistic Methods.

- Fermat's test does not guarantee if the answer is correct.
- It is only reducing the probability of error in the test.
- Exceptions are not correct.
  Numbers exists that Fermat's test.
  They are not prime yet a^n is congruent to $a \bmod n$ for all n.
- Variations of Fermat tests that cannot be fooled
  - One can prove that for any n, the condition does not hold for most of the integers a even when n is prime.
  - $\therefore$ If n passes the test for 2 random choices of a, chances are better than 3/4 that n is prime.
  - are known as probabilistic algos.

1.3

When we declare

```
function cube(a) {
    return x * x * x;
}
```

} we are talking about a method to obtain the cube of any number.

↳ Build abstractions by assigning names to common patterns and then to work in term of abstraction directly.

→ higher-order functions:
functions that accept other functions as arguments / return functions as values.

Functions as arguments?

```
function sum_integers(a, b) {
    return a>b
        ? 0
        : a + sum_integers(a+1, b);
}
```
sum of ints from a to b.

```
function sum_cubes(a, b) {
    return a>b
        ? 0
        : cube(a) + sum_cubes(a+1, b);
}
```
sum up the cube of ints from a to b.

```
function pi_sum(a, b) {
    return a>b
        ? 0
        : 1 / (a * (a+2)) + pi_sum(a+4, b));
}
```
sum of a sequence $\frac{1}{1\cdot3} + \frac{1}{5\cdot7} + \frac{1}{9\cdot11} + \dots$ converges to $\frac{\pi}{8}$.

Observe a common underlying pattern.
- differs only in the name of the function
- template:

```
Function name(a,b) {
    return a>b
        ? 0
        : term(a) + name(next(a), b);
}
```

↳ similar to sigma notation:

$$\sum_{n=a}^{b} f(n) = f(a) + \dots + f(b).$$

abstraction of summation of a series

↳ allows us to deal with the concept of summation.
eg. formulate general result about sums that are independent of the particular series.

Transforming them into common template:

```
function sum(term, a, next, b) {
    return a>b
        ? 0
        : term(a) + sum(term, next(a), next, b);
}
```

↓

```
function pi_sum(a,b) {
    function pi_term(x) {
        return 1 / (x*(x+2));
    }
    function pi_next(x) {
        return x+4;
    }
    return sum(pi_term, a, pi_next, b);
}
```

↳ building blocks in formulating further concepts:

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx.$$

eg. definite integral of a function f between limits a and b.

```
function integral(f, a, b, dx) {
    function add_dx(x)
        return x + dx;
    }
    return sum(f, a + dx/2, add_dx, b) * dx;
}
```

Lambda Expressions.

- Rather than declaring trivial functions such as pi_term & pi_next,

  more convinient to directly specify "the function that returns its input incremented by 4"

  or "the function that returns the reciprocal of its input times its input plus 2".

- lambda expressions: syntactic sugar for creating functions.

  ```
  x => x+4
  x => 1 / (x * (x+2))
  ```

- ```
  function pi_sum (a, b) {
      return sum ( x => 1 / (x * (x+2)),
                   a,
                   x => x + 4,
                   b );
  }
  ```

- In general, => exp. are used to create functions in the same way as function declarations, except that no name is specified. For function and return keywords and braces are omitted.

  (parameters) => expression

  ```
  function plus4(x) {
      return x+4;
  }
  ```
  =
  ```
  const plus4 = x => x+4;
  ```

  read as: the function of an argument x that results in the value plus 4.

- Like any expression that has a function as its value, a lambda expression can be used as the function expression in an application:

  ```
  ((x,y,z) => x+y+square(z)) (1,2,3);
  ```

  lower precedence than function application: parentheses around the lambda expression.

  in any context where we would normally use a function name.

Using const to create local names:

- need local names in our functions other than those bound by parameters.

  eg. $f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$

  $a = 1+xy$

  $b = 1-y$

  $f(x,y) = xa^2 + yb + ab$

  - we would like to include local names x and y and the names of intermediate quantities like a, b.
  - auxiliary function to bind local names:

    ```
    function (x,y) {
        function f_helper(a,b) {
            return x * square(a) + y * b + a * b;
        }
        return f_helper ( 1 + x*y, 1 - y );
    }
    ```

  - lambda exp. to specify an anonymous function for binding local names:

    ```
    function f_2(x,y) {
        return ((a,b) => x * square(a) + y * b + a * b) ( 1 + x * y , 1 - y );
    }
    ```
    function body => single call to that function.

  - more conventional way of using const declaration):

    ```
    function f_3 (x,y) {
        const a = 1 + x * y;
        const b = 1 - y;
        return x * square(a) + y * b + a * b;
    }
    ```
    names are declared with const inside a block => have body of function surrounding block as their scope.

Conditional Statements.

function is unnecessarily inefficient.

=> 2 identical calls.

=> introduce separate function, or

```
function expmod (base, exp, m) {
    return exp === 0
        ? 1
        : is_even(exp)
        ? ( expmod(base, exp/2, m)
            * expmod(base, exp/2, m)) % m
        : (base * expmod(base, exp-1, m)) % m;
}
```

```
function expmod (base, exp, m) {
    const half_exp = expmod (base, exp/2, m);
    return exp ===0
        ? 1
        : is_even(exp)
        ? (half_exp * half_exp) %m
        : (base * expmod(base, exp-1, m)) % m;
}
```

NON-TERMINATING!!!

const declaration is outside of conditional exp.

∴ execute even when base case is met

To resolve this:

Conditional Statements!

- allow return statements to appear in the branches of the statement.
- general form

  if (predicate) { consequent statement } else { alternative statement }.

```
function expmod(base, exp, m) {
    if (exp === 0) {
        return 1;
    } else {
        if (is_even(exp)) {
            const half_exp = expmod( base, exp/2, m);
            return (half_exp * half_exp ) % m;
        } else {
            return ( base * expmod (base, exp - 1, m)) % m;
        }
    }
}
```

- Interpreter first evaluates the predicate.

  If true: evaluates the consequent-statements in sequence.

  If false: evaluates alt-statements.

  Forms its own block.
```

Functions as general Methods.

eg:

Finding root of eqns by the half-interval method.

- The idea is that if points a, b are such that $f(a) < 0 < f(b)$, then f must have at least one zero between a and b.
- To locate a zero, let x be average of a and b, then find $f(x)$.
  If $f(x) > 0$, then f must have a zero between a and x.
  If $f(x) < 0$, then f must have a zero between x and b.
- Continuing this way, we can identify smaller and smaller intervals on which f must have a zero.
- Process stops when interval is small enough.
- Since the interval of uncertainty is reduced by half at each step of the process, the number of steps required grows as $\Theta(\log(L/T))$ where L is the length of the original interval and T is the error tolerance.

```
function search(f, neg_point, pos_point) {
  const midpoint = average(neg_point, pos_point);
  if(close_enough(neg_point, pos_point)) {
    return midpoint;
  } else {
    const test_value = f(midpoint)
    return positive(test_value)
      ? search(f, neg_point, midpoint)
      : negative(test_value)
      ? search(f, midpoint, pos_point)
      : midpoint;
  }
}
```

First compute the midpt of the 2 points.

assume we are given function f together w/ points at which values are neg. and pos.

check if the given interval is small enough.
↳ encountered for test_value = 0.

if so, just return midpoint.

Compute the test value of f at midpoint

⇒ if test_value > 0, continue the process with a new interval running from original negative point to midpoint.

if test_value < 0, continue interval from midpoint to pos_point.

```
function close_enough(x, y) {
  return abs(x - y) < 0.001;
}
```

Finding fixed points of functions

- x is a fixed point of f if $f(x) = x$.
- Begin w/ an initial guess and apply f repeatedly:

  $f(x),\ f(f(x)),\ f(f(f(x)))\ \ldots$

- ∴ Values of function are input and an initial guess

```
const tolerance = 0.00001;
function fixed_point(f, first_guess) {
  function close_enough(x, y) {
    return abs(x - y) < tolerance;
  }
  function try_with(guess) {
    const next = f(guess);
    return close_enough(guess, next)
      ? next
      : try_with(next);
  }
  return try_with(first_guess);
}
```

f must converge!

A way to prevent guess from oscillation is to prevent it from changing too much.

based on the idea of repeatedly improving guess until the result satisfy some criterion.

⇒ average damping.

Functions as return values.

- we can express the idea of average damping:

```
function average_damp(f) {
    return x => average(x, f(x));
};
```

takes a function f

return a function as value when applied to x; produces an average of x and f(x).

Using average_damp,

```
function sqrt(x) {
    return fixed_point(average_damp(y => x / y), 1);
};
```

With 3 ideas

⇒ much clearer

∴ need know how to choose process formulations that are clear and where useful elements are exposed as separate entities that can be reused in other applications.

Newton's method

- If $x \mapsto g(x)$ is a differentiable function, then a solution of the equation $g(x) = 0$ is a fixed point of the function $x \mapsto f(x)$, where

$$f(x) = x - \frac{g(x)}{Dg(x)}$$ ⇒ derivative of g evaluated on x.

- Newton's method is using fixed point method to approx. a root.
  - for many functions g and for sufficiently good initial guess for x, Newton's method converges very rapidly to a root of $g(x) = 0$.
  - need the idea of derivative:

"derivative" is another that transforms a function into another function.

In general,

$$Dg(x) \approx \frac{g(x + dx) - g(x)}{dx}$$ (dx: small number), in the limit of small dx.

derivative of g.

```
function deriv(g) {
    const dx = 0.00001;
    return x => (g(x + dx) - g(x)) / dx;
};
```

```
function newton_method(g, guess) {
    return fixed_point(newton_transform(g), guess);
}
```

takes a function as arg.

computes the function for which we want to find a zero.

```
function newton_transform(g) {
    return x => x - g(x) / deriv(g)(x);
};
```

Abstractions & First-class functions.

Newton's method as fixed point method:

- earlier method begins by a function and finds a fixed point of some transformation of a function: (square root)

```
function fixed_point_of_transform(g, transform, guess) {
    return fixed_point(transform(g), guess);
};
```

returns a fixed point of the transformed function.

```
function sqrt(x) {
    return fixed_point_of_transform(
        y => x / y,
        average_damp,
        1);
}
```

```
function sqrt(x) {
    return fixed_point_of_transform(
        y => square(y) - x,
        newton_transform,
        1);
};
```

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated.

Elements w. the fewest restrictions ⇒ first-class status.

"rights and privileges" of first-class elems:

- may be referred to using names.
- may be passed as arguments to functions
- may be returned as the results of functions
- may be included in data structures.

- gain in Expressive power.

2.1

Data abstraction.

- A method that helps us isolate the usage of compound data objects from how it is constructed from more primitive data objects.
- Basic idea is to structure the programs that use such compound data objects so that they operate on 'abstract data'.
- Programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand.
- "Concrete" data representation is defined independent of the programs that use the data.
- Interface btw 2 parts of our system will be a set of functions ⇒ selectors & constructors.
  ↳ implements the abstract data in terms of concrete representation

Eg: Arithmetic Operations For Rational Numbers

→ able to subtract, add, multiply, divide

→ assume a way of constructing a rational number from a numerator and a denominator.

→ assume, given a rational number, we have a way of extracting its numerator and denominator.

→ assume, constructors and selectors are available as functions:

- make_rat(n,d)
  returns rational no. whose numerator is the integer n and denominator is d.
- numer(x) returns the numerator of the rational number x.
- denom(x) returns the denominator of the rational number x.

→ 'wishful thinking'

→ relations:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \Rightarrow$$

```
function add_rat(x,y){
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                    denom(x) * denom(y))
}
```

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \Rightarrow$$

```
function sub_rat(x,y){
    return make_rat(numer(x) * denom(y) - numer(y) * denom(x),
                    denom(x) * denom(y));
}
```

```
function mult_rat(x,y){
    return make_rat(
    numer(x) * numer(y),
    denom(x) * denom(y));
```

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2} \Leftarrow$$

```
}
```

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1 d_2}{d_1 n_2} \Rightarrow$$

```
function div_rat(x,y){
    return make_rat(numer(x) * denom(y),
                    denom(x) * numer(y));
}
```

```
function equal_rat(x,y){
    return numer(x) * denom(y)
        === numer(y) * denom(x)
}
```

$\Leftarrow \frac{n_1}{d_1} = \frac{n_2}{d_2}$ iff $n_1 d_2 = n_2 d_1$

Need some way to glue together a numerator and denominator to form a rational number.

Pairs

- To implement the concrete level of our data abstraction
- Pair takes 2 args and returns a compound data object that contains the arguments as parts.
- extract parts of pairs. Primitive using heads and tail.
- const x = pair(1,2); ⇒ data object that can be given a name and manipulated.
  ↳ can be used to form pairs whose elements are pairs...
- general purpose building blocks ⇒ acts as the glue
- data objects constructed from pairs are called list-structured data.

```
function make_rat(n,d) { return pair(n,d); }.
function numer(x) { return head(x); }.
function denom(x) { return tail(x); }
function print_rat(x)
    return display(stringify(numer(x)) + "/" + stringify(denom(x)));
```

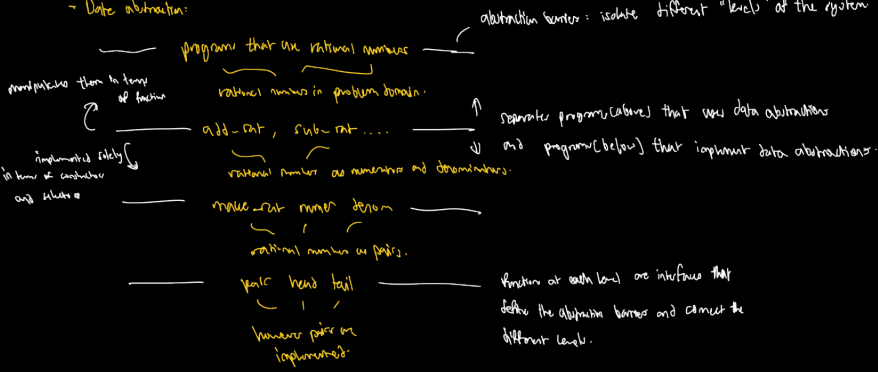
⇒ note that the rational number implementation does not reduce rational numbers to lowest terms.
↳ can use gcd to reduce to lowest term.

Abstraction Barriers

→ issues raised by rational-number example:

- defined the rational-number operations in terms of a constructor make-rat and selectors numer and denom.
- The underlying data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed; then to use only those operations on data.

- Data abstraction:



→ this simple idea has many advantages:

1. makes programs much easier to maintain and modify.
2. complex data structures can be represented in a variety of ways with the primitive data structures provided by a programming language
3. Choice of representation influences the programs that operate on it.
   ∴ all such programs might have to be modified accordingly.
4. Task would be time-consuming and expensive for large programs unless the dependence on representation were to be confined by design to a few ⇒ all. ways of reducing rational numbers is to perform reduction whenever we access parts of a rational number, rather than we construct it.

→ Controlling the dependence on the representation to a few interface functions ⇒ help us design programs and modifying them. ⇒ allows us to maintain the flexibility to consider alt. implementations.

⇒ decision to use gcd at construction time or selection time ⇒ data abstraction model gives us a way to defer the decision and make progress to work on the programs.

If for typical use, we access numerator & denominator of some rational number many times, prefer to compute gcd when rational numbers are constructed

```
function make-rat(n,d) {return pair(n,d);}
function numer(x) {
    const g = gcd(head(x), tail(x));
    return head(x)/g;
}
function denom(x) {
    const g = gcd(head(x), tail(x));
    return tail(x)/g;
}
```

When we change from one rep. to another, add-rat or sub-rat etc. don't need to be modified

What is meant by data?

⇒ implementation of add_rat, sub_rat

in terms of 3 unspecified functions : make_rat, numer, denom.

⇒ think of operations as different data objects.

Whose behaviour are specified by latter 3 functions.

⇒ In general, data is defined by some collection of selectors and constructors, together with specified conditions that these functions must fulfill in order to be a valid representation.

⇒ can serve to define not only "high-level" data objects (rational no.) but also lower-level objects as well. (eg. pair)

⇒ we could implement pair, head and tail without using any data structures at all but only using functions.

valid ways to def. pair? Verify that these functions satisfy the conditions.

```
function pair (x, y) {
    function dispatch (m) {                  // informally defined function
                                             // takes one arg., returns either x or y. (Returns if 0 or 1)
        return m === 0
            ? x
            : m === 1
            ? y
            : error (m, "argument not 0 or 1 — pair");
    }
    return dispatch;
}

function head (z) { return z(0); }.   ⇒ apply z to 0, z applied to 0 will yield x.
function tail (z) { return z(1); }.
```

demonstrates the ability to manipulate functions as objects automatically provides the ability to represent compound data.

Style of programming ⇒ message passing

Interval Arithmetic

- ability to manipulate inexact quantities with known precision
- returns number of known precision
- Compute $R_p = \frac{1}{1/R_1 + 1/R_2}$
- Interval arithmetic ⇒ a set of operation for combining intervals / objects that rep. range of possible values of an inexact quantity.
- result of +, −, ×, ÷ ⇒ interval = range of result
- Abstract object: interval ⇒ 2 endpoints (lower bound and upper bound).
- Interval can be constructed using make_interval

- min val. of the sum = sum of 2 lower bounds.
- max val. of sum = sum of 2 upper bound.

```
function add_interval (x, y) {
    return make_interval ( lower_bound (x) + lower_bound (y), upper_bound (x) + upper_bound (y));
}.
```

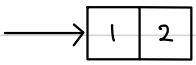
```
function mul_interval (x, y) {
    const p1 = lower_bound (x) * lower_bound (y);
    const p2 = lower_bound (x) * upper_bound (y);
    const p3 = upper_bound (x) * lower_bound (y);
    const p4 = upper_bound (x) * upper_bound (y);
    return make_interval ( math_min (p1, p2, p3, p4)),
                           math_max (p1, p2, p3, p4));
}.
```

find min & max of product of bounds.

```
function div_interval (x, y) {  → multiplies first by reciprocal of bound.
    return mul_interval (x, make_interval ( 1 / upper_bound (y),
                                            1 / lower_bound (y))) };
}
```

bounds are reciprocal of upper bound & reciprocal of lower bound.

2.2



= pair (1, 2)

Pair is a primitive 'glue' to construct compound data objects.

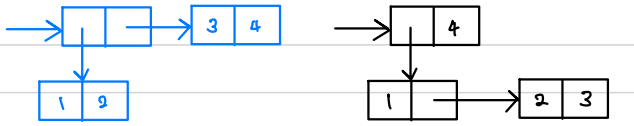
Box-and-Pointer notation: each compound object is shown as a pointer to a box.

Box: left part – head of pair

right part – tail of pair

Pair – can combine numbers and pair ← essence of list's structure as a representation tool.
⇒ Closure property of pair

– is a universal building block to construct all sorts of data structure



pair (pair (1,2), pair (3,4)) pair (pair (1, pair (2,3)) , 4)

∴ An operation for combining data satisfies the closure property if the result of combining things with that operation can themselves be combined using the same operation.

⇒ Closure permits us to create hierachical structures (structures made up of parts which themselves are made up of parts and so on.)

Representing Sequences.

Sequence: ordered collection of objects.

constructed with pairs.

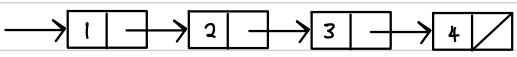
eg.

```
pair(1,
     pair(2,
          pair(3,
               pair(4, null))));
```

head of each pair is the corresponding item in the chain.

tail of the pair is the next pair in the chain

tail of final pair signals end of sequence.



Sequence formed by nested pair applications — list

list ($a_1, a_2, \dots, a_n$)

≡ pair (a_1 , pair (a_2 , pair (a_3 , ... , pair (a_n , null) ...)))

terminate chain of pair ⇒ empty list

Box notation — textual representation of box-and-pointer diagrams.

eg. pair (1,2) = [1, 2]

list (1,2,3,4) = [1 [2 [3 [4, null]]]]

Selecting items in list

head ( ); → selecting first item in list

tail ( ) ; → selecting sublist (consist of all but the first item)

Nested application of head and tail can extract second, third and subsequent items in list.

list notation: eg. [1 [[2,3], [[4, [5, null]], [6, null]]]].

list (1, [2,3], list(4,5), 6).

Manipulate a list

- successively using tail to walk down a list.
- eg.

```
function list_ref (items, n) {
    return n===0
        ? head (items)
        : list_ref ( tail(items) , n-1);
}
```

list: items; number: n

return nth item of list

if n=0, return head of list.

otherwise, return the (n-1)st item of the tail of the list.

is_null ( ); (test whether its argument is the empty list null).

Simple recursion plan

```
function length (items) {
    return is_null (items)
        ? 0
        : 1 + length (tail (items));
}
```

return number of items in a list

applied successfully until reaching base case:

length of empty set is 0.

reduction step: the length of any list is 1 + length of tail of the list.

iterative:

```
function length (items) {
    function length_iter (a, count) {
        return is_null (a)
            ? count
            : length_iter ( tail(a), count + 1);
    }
    return length_iter (items , 0);
}
```

append ( );

↳ constructs an answer list by adjoining elements to the front of the list with pair while walking down a list using tail.

Takes 2 lists as argument → combines their elements to make a new list.

recursive

```
function append (list1 , list2) {
    return is_null (list1)
        ? list 2
        : pair ( head (list 1), append (tail (list1), list2));
}
```

if list1 is empty, result is list2.

otherwise,

adjoin the head of list1 to the result.

append the tail of list1 and list2.

Mapping over lists

- Apply transformation to each element in a list.

- eg.
```
function scale_list (items, factor) {
    return is_null (items)
        ? null
        : pair (head (items) * factor,
                scale_list (tail (items), factor));
}.
```

Scales each number in a list by a given factor.

Original definition of scale_list recursive structure draws attention to the element-by-element processing of the list.

Difference: think about the processes differently.

abstract this general idea
⇒ as a common pattern expressed by a higher order function
⇒ map.
⇒ takes a function as a argument and a list, and returns a list of the result produced by applying the function to each elements in a list.

defining scale_list in terms of map suppresses that level of detail and emphasizes that scaling transforms a list of elements to a list of results.

```
function map (fun, items) {
    return is_null (items)
        ? null
        : pair (fun (head (items)),
                map (fun, tail (items)));
}.
```

```
∴ const scale_list = (items, factor) => map( x => x * factor, items);
```

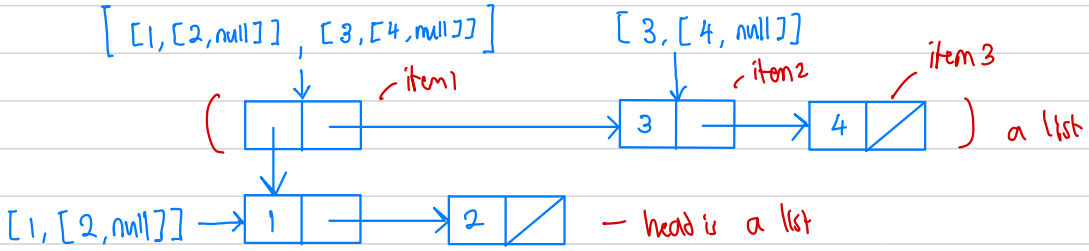
important construct. → captures a common pattern
→ establishes a higher level of abstraction in dealing with lists.

map helps establish an abstraction barrier that isolates the implementation of functions that transform lists from details of how the elements of the list are extracted and combined.

abstraction give us flexibility to change low-level details of how sequences are implemented, while preserving the conceptual framework of operation that transforms sequences to sequences.

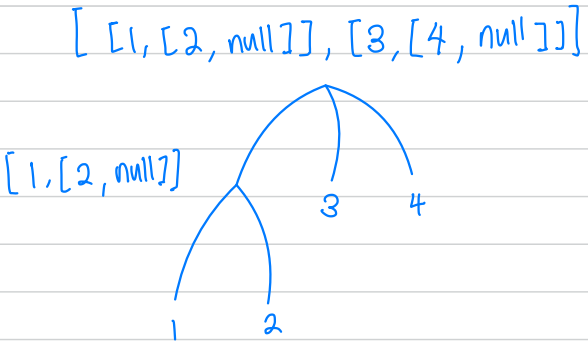
Hierarchical Structures.

- Representation of sequences in terms of lists generalizes naturally to represent sequences whose elements themselves may be sequences.
- eg. pair(list(1,2), list(3,4));
  = list of 3 items.



- Trees - sequences whose elements are sequences.

  elements of the sequence are the branches of the tree, elements themselves are subtrees.



Recursion - deals with tree structures.
- reduce operations on trees to operations on their branches until reaches the leaves of the trees.

eg.
```
function count_leaves (x) {
return is_null(x)
    ? 0                          → value of empty list = 0.
    : ! is_pair(x)               → if reach actual leaves,
                                   count_leaves of a leaf is 1.
      ? 1
      : count_leaves(head(x)) + count_leaves(tail(x));
}
```

primitive predicate to test whether its argument is a pair.

In the reduction step, strip off the head of the list, must take into account that the head may itself be a tree whose leaves we need to count.
⇒ count_leaves of a tree x is count_leaves of the head of x plus count_leaves of the tail of x.

Similar:
```
function length ( x ) {
    return is_null ( x )
        ? 0   → the length of empty list is 0.
        : 1 + length(tail ( x ));
}
```
the length of a list is 1 + tail of the list

Mapping Over trees.

- map is a powerful abstraction for dealing with trees.

- scale_tree function is analogue to scale_list,

- (recursive plan of scale_tree is similar to count_leaves).

```
function scale_tree (tree, factor) {
    return is_null(tree)
        ? null
        : ! is_pair(tree)
        ? pair( scale_tree (head(tree), factor),
                scale_tree ( tail (tree), factor));
}
```

a tree whose leaves are numbers — a numeric factor

returns a tree of the same shape, where each number is multiplied by a factor.

- or we can regard the tree as a sequence of sub-trees and use map.

```
function scale_tree (tree, factor) {
    return map ( sub_tree => is_pair ( sub_tree)
                        ? scale_tree ( sub_tree, factor)
                        : sub_tree * factor,
                tree);
}
```

base case: if tree is a leaf, simply multiply by factor.

maps over each sequence, scaling each sub-tree in turn and return the list of results

- many tree operations can be implemented by similar combinations of sequence operations and recursion.

Sequences as Conventional Interfaces.

- powerful design principle for working with data structures.

- program abstractions (implemented as higher-order functions) can capture common patterns in programs that deal with numerical data.

- our ability to formulate analogous operations for working with compound data depends crucially on the style in which we manipulate our data structures.

- eg. function sum_odd_squares (tree) { ← analogous to count_tree function
  return is_null (tree) ← takes a tree as a argument
  ? 0
  : ! is_pair (tree)
  ? is_odd (tree) ? square (tree) : 0
  : sum_odd_squares (head (tree)) + ← computes sums of the squares of the leaves that are odd.
  sum_odd_squares (tail (tree));
  }.

This program:
- enumerates the leaves of a tree;
- filters them, selecting the odd ones.
- square each of the selected ones
- accumulates the result using +, starting with 0.

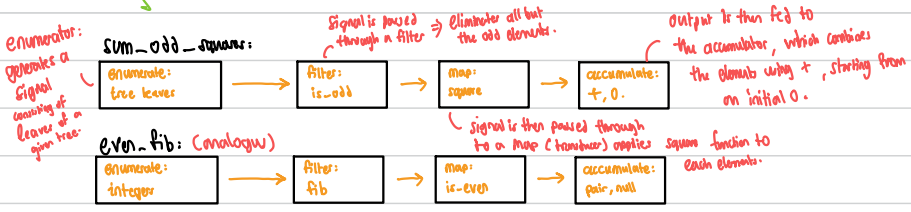
On the surface, these 2 functions are very different

despite the fact that they are structurally very different, a more abstract description of the 2 reveals similarities.

function even_fibs (n) {
function next (k) {
if (k > n) {
return null;
} else {
const f = fib (k); ← k is less than or equal to a given integer n.
return is_even(f)
? pair (f, next(k+1)) ← constructs a list of all even fibonacci numbers.
: next (k+1);
}.
}.
return next (0);
}.

note: these functions fails to exhibit signal-flow structure.

This program:
- enumerates the integers from 0 to n.
- computes the fibonacci numbers for each integer.
- filters them, selecting the even ones.
- accumulates the result using pair, starting with the empty list.



Sequence Operations.

- key to organise programs to clearly reflect the signal-flow structure is to concentrate on the "signals" that flow from one stage of the process to the next.

- If we represent these signals as lists, then we can use list operations to implement the processing at each of the stages.

- eg. map (square, list(1, 2, 3, 4)); implement mapping stages of the signal flow diagrams.

```
function filter (predicate, sequence) {   filtering a sequence to select only those elements that satisfy
    return is_null(sequence)                a given predicate.
        ? null
        : predicate (head(sequence))
        ? pair (head(sequence),
                filter(predicate, tail(sequence)))
        : filter(predicate, tail(sequence));
}.

function accumulate (op, initial, sequence) {   accumulation.
    return is_null(sequence)
        ? initial
        : op (head(sequence),
              accumulate(op, initial, tail(sequence)));
}.
```

Enumerate sequence of elements to be processed:
(for exm-filter, need to generate the sequence of integers in a given range)

```
function enumerate_interval (low, high) {
    return low > high
        ? null
        : pair (low,
                enumerate_interval (low + 1, high));
}.
```

(to enumerate the leaves of a tree)

```
function enumerate_tree (tree) {
    return is_null(tree)
        ? null
        : ! is_pair(tree)
        ? list(tree)
        : append (enumerate_tree (head(tree)),
                  enumerate_tree (tail(tree)));
}.
```

Reformulation of sum_odd_squares and even_fibs:

```
function sum_odd_squares (tree) {
    return accumulate ( plus,
                        0,
                        map ( square,
                              filter (is_odd,
                                      enumerate_tree (tree))));
}.
```

- add up the results.
- square each element
- filter this only to keep the odd numbers in sequence
- enumerate the sequence of leaves of the tree.

```
function even_fibs (n) {
    return accumulate ( pair,
                        null,
                        filter (is_even,
                                map (fib,
                                     enumerate_interval (0,n))));
}.
```

- accumulates the results into a list.
- filter the resulting sequence to keep only the even elements
- generate the Fibonacci number for each integer
- enumerate integers from 0 to n.

- expressing programs as sequence operations helps us make program designs that are modular ⇒ designs that are constructed by combining relatively independent pieces.

- encourage modular design by providing a library of standard components, together with a conventional interface for connecting the components in flexible ways.

- modular construction ⇒ powerful strategy for controlling complexity in engineering design
- sequence operations provide a library of standard program elements that we can mix and match.
- eg. we can reuse pieces from sum_odd_squares and even_fibs functions in a program that constructs a list of the squares of the first n+1 Fibonacci numbers:

```
function list_fib_squares (n) {
    return accumulate ( pair,
                        null,
                        map (square,
                             map (fib,
                                  enumerate_interval (0,n))));
}.
```

- we can rearrange the pieces for computing the product of the squares of odd integers in sequence:

```
function product_of_squares_of_odd_elements (sequence) {
    return accumulate( times,
                       1,
                       map ( square,
                             filter (is_odd, sequence)));
}
```

- sequence implementations also let us define a conventional interface that permits us to combine processing modules.
- localize the data-structure dependencies in programs to a small number of sequence operations.

Nested Mappings

- Sequence paradigm expansion to include many computations that are commonly expressed using nested loop.
- Given a positive integer n,
  find all the ordered pairs of distinct positive integers i and j,
  where $1 \leq j < i \leq n$ such that $i+j$ is prime

eg. $n=6$, then the pairs:

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

- To organise this computation -
  1. generate the sequence of all ordered pairs of positive integers $\leq n$.
  2. filter those pairs whose sum is prime
  3. then, for each pair (i,j), produce the triple $(i,j,i+j)$.

```
accumulate (append,
            null,
            map (i ⇒ map (j ⇒ list(i,j),
                          enumerate_interval (1, i-1)),
                 enumerate_interval (1, n)));
```

- accumulate (append, null, ...): Combine all the sequence for all the i produces the required sequence of pairs.
- map (i ⇒ ...): For each integer $i \leq n$
- map (j ⇒ list(i,j), ...): enumerate the integer $j < i$
- list(i,j): for each such i,j generate the pair (i,j)
- enumerate_interval (1, i-1): for each i in the sequence, map along this.
- enumerate_interval (1, n): map the sequence operation along this.

- Combination of mapping and accumulating with append is so common that it is isolated as a separate function:

```
Function flatmap (f, seq) {
    return accumulate (append, null, map (f, seq));
};
```

```
function prime_sum_pairs(n) {
    return map(make_prime_sum,
               filter(is_prime_sum,
                      flatmap(i => map(j => list(i, j),
                                       enumerate_interval(1, i-1)),
                              enumerate_interval(1, n))));
}
```

⇒ generates the sequence of results by mapping over filtered pairs
⇒ constructs a triple consisting of 2 elements in a pair along with their sum.

```
function make_prime_sum(pair) {
    return list(head(pair), head(tail(pair)),
                head(pair) + head(tail(pair)));
}
```

```
function is_prime_sum {
    return is_prime(head(pair) + head(tail(pair)));
}
```

⇒ filters the sequence of pairs to find those whose sum is prime.
⇒ args is pair, extracts integers from pair
⇒ predicate that applies to each element.

- Nested mappings are useful for sequences other than those that enumerate intervals.

- eg. generate all permutations of a set S. ⇒ all the ways to ordering the items in the set.

{1,2,3} ⇒ {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}

· plan:

1. For each item x in S, recursively generate the sequence of permutation S-x and adjoin x to the front of each one.
2. This yields, for each x in S, the sequence of permutations of S that begin with x.
3. Combining these sequences for all x gives all the permutations of S.

```
function permutation(S) {
    return is_null(S)
        ? list(null)
        : flatmap(x => map(p => pair(x, p),
                           permutation(remove(x, S))),
                  S);
}
```

⟶ in the terminal case, empty list = set of no elements ⇒ list(null) sequence with one item; the set with no elements

retrns all the items in a given sequence except for a given item.

```
function remove(item, sequence) {
    return filter(x => !(x === item),
                  sequence);
}
```

- reduces the problem from generating permutations of S to generating permutations of sets with fewer elements than S.

2.3 Symbolic Data

Strings.

- We can form compound data with strings:
  list("a", "b", "c")
  list(list ("Jakob", 27), list("Lova", 9), list("Lusica", 24)).

- To distinguish strings from names - " ".

- expresion z denotes value of the name z,
  expression "z" denotes a string that consists of a single character.

- predicate:
  === returns true if 2 strings are the same.

takes 2 args : a list of strings and a string
or a list of numbers and a number.

```
function member (item, x) {
    return is_null (x)
        ? null
        : item === head (x)
        ? x
        : member( item, tail (x) );
}
```

if first arg is not contained in the list => returns null.
Otherwise returns sublist of the list in the beginning with first occurance of the string or number.

Example: Symbolic Differentiation.

- An illustration of Symbol Manipulation and a further illustration of data abstraction.

- The design of a function that takes as args an algebraic expression and a variable returns the derivative of the expression w.r.t the variable.

- Follows the same strategy of data abstraction that we followed in developing the rational number system in 2.1.
  - 1. First, define a differentiation algorithm that operates on abstract objects such as 'sums', 'products', 'variables' without worrying about how these are to be represented.

- A very simple symbolic-differentiation program that handles expressions that are built up using only + and × with 2 args.

Differentiation of such expressions can be carried out using these reduction rules:

$$\frac{dc}{dx} = 0 \quad , c \text{ is a constant, } x \text{ is a variable}$$

$$\frac{dx}{dx} = 1$$

recursive in nature

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

to obtain the derivative of a sum, first find derivative of the terms and add them

⇒ each of the terms may be an expression that needs to be decomposed.

⇒ decomposing into smaller & smaller pieces will eventually produce pieces that are either constant or variables

- We need to tell if an expression is a sum, a product, a constant or a variable
- We need to extract the parts of an expression.

- We need to construct expressions from parts.
- Assuming we have these functions:

```
is_variable(e)
is_same_variable(v1, v2)
is_sum(e)
addend(e)
augend(e)
make_sum(a1, a2)
is_product(e)
multiplier(e)
multiplicand(e)
make_product(m1, m2)
```

express the differentiation rules in this function:

```
function deriv(exp, variable) {
    return is_number(exp)
        ? 0
        : is_variable(exp)
        ? is_same_variable(exp, variable) ? 1 : 0
        : is_sum(exp)
        ? make_sum(deriv(addend(exp), variable),
                   deriv(augend(exp), variable))
        : is_product(exp).
        ? make_sum(make_product(multiplier(exp),
                                deriv(multiplicand(exp),
                                      variable)),
                   make_product(deriv(multiplicand(exp),
                                      variable),
                                multiplier(exp)))
        : error(exp, "unknown expression");
}.
```