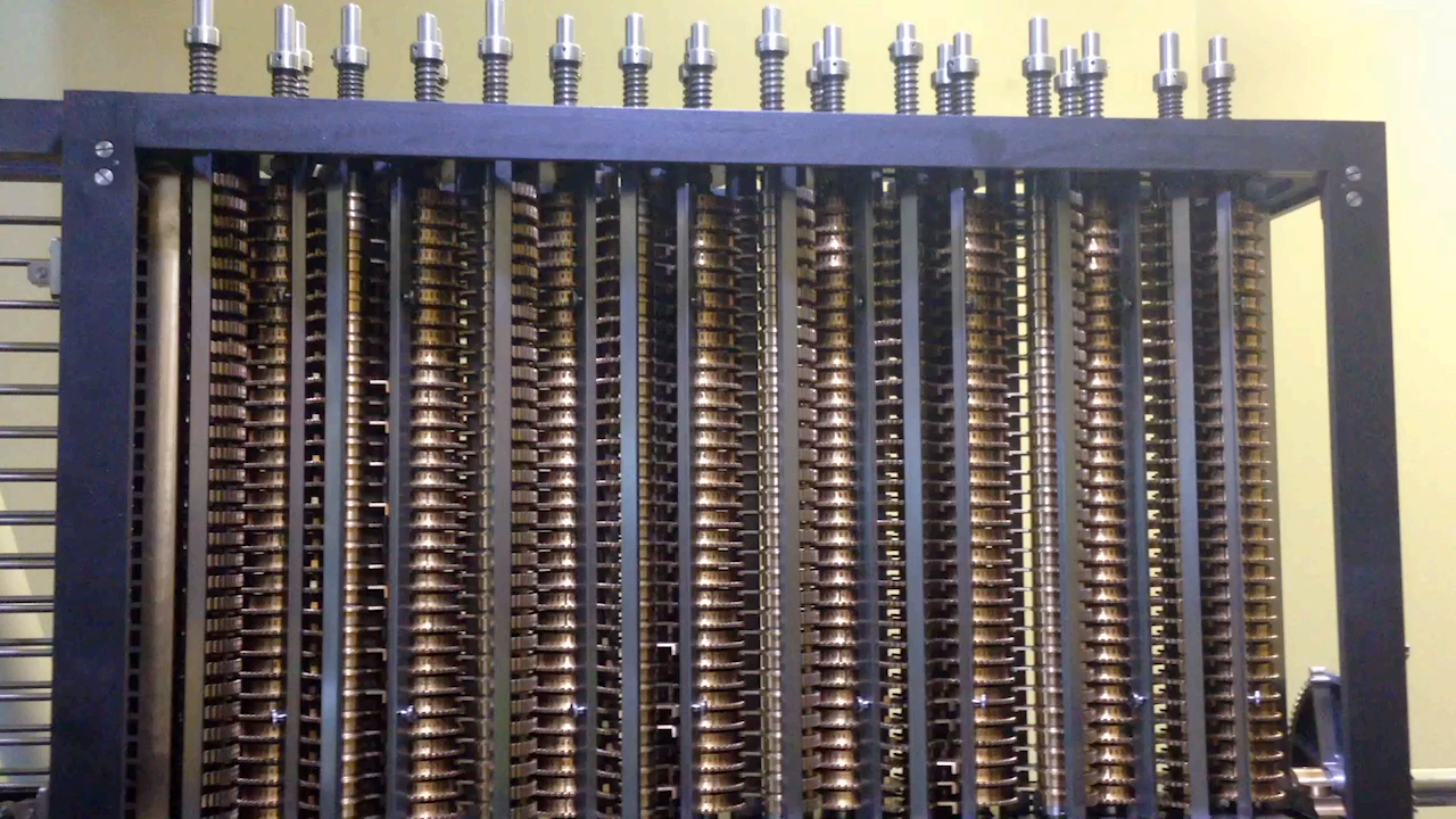
CS4111 - Computer Science

Lecture Set 4: Boolean Algebra and Recursion

Logic

- True or False
  - (IF-THEN-ELSE)
- Charles Babbage (1791 - 1871)
  - Differential Engine (1822)
    - Solve Polynomial Functions
    - Faraday's electric engine (1821)
  - Analytical Engine (1830)
    - Programmable, memory, printer, CPU
    - First built 153 years later!

Vision while on opium
"The Void"
"Existence"



George Boole (1815 - 1864)

- First Professor of Mathematics in UCC
- Formalised logic
- Lets us reason about unseen cases
  - Enables scaling in modern computers — hyperscale
- “The Joy Of Logic”
  - https://vimeo.com/137147126

- Boolean Operators
  - (AND, OR…)
- Relational Operators
  - (<, >, =…)
- Prefix notation?
  - (> 2 1) ... True
  - (< 4 2) ... False
- Racket?

```
>  (> 2 1)
#t
> (= 2 1)
#f
> (< (+ 3 1) (* 4 5))
#t
```

- > (+ 2 (> 3 1))
  - Error

- Boolean Operators
  - (AND, OR…)
- Relational Operators
  - (<, >, =…)
- Prefix notation?
  - (> 2 1) ... True
  - (< 4 2) ... False
- Racket?

```
> (> 2 1)
#t
> (= 2 1)
#f
> (< (+ 3 1) (* 4 5))
#t
```

- > (+ 2 (> 3 1))
  - Error

Question

Is (3 2 2 1) a descending list?

(> 3 2 2 1).. #f

(>= 3 2 2 1)..#t

Conditionals

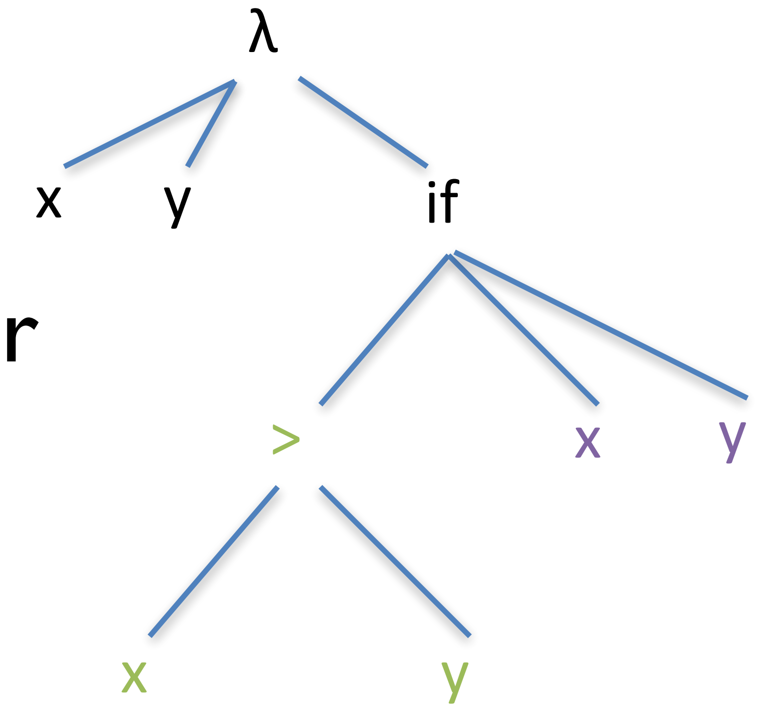
- General view of conditional:
  - if E then C1 else C2
- Meaning:
  - if condition E is true
    - THEN execute command(s) C1
    - ELSE execute command(s) C2
- λ calculus / Racket view:
  - if condition E is true
    - THEN return C1
    - ELSE return C2

Examples

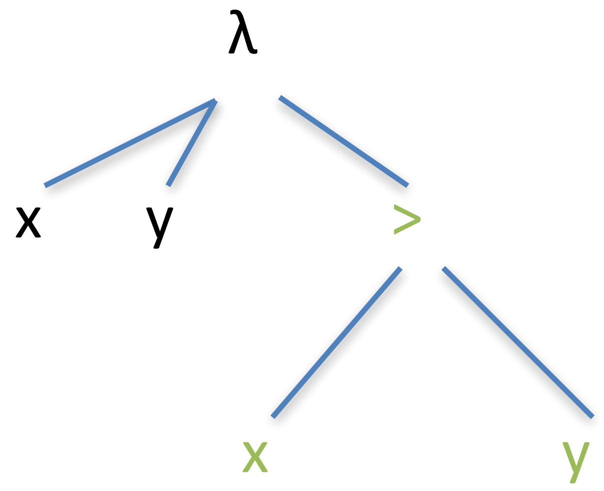
- > (if (> 2 0) “first” “second”)
  - ``first"
- Return the larger of two numbers:
  - (λxy. if (> x y) x y)
- AST:

Similar (but different)

(λxy. (> x y))

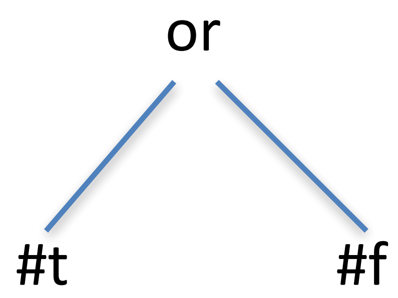


Returns larger number

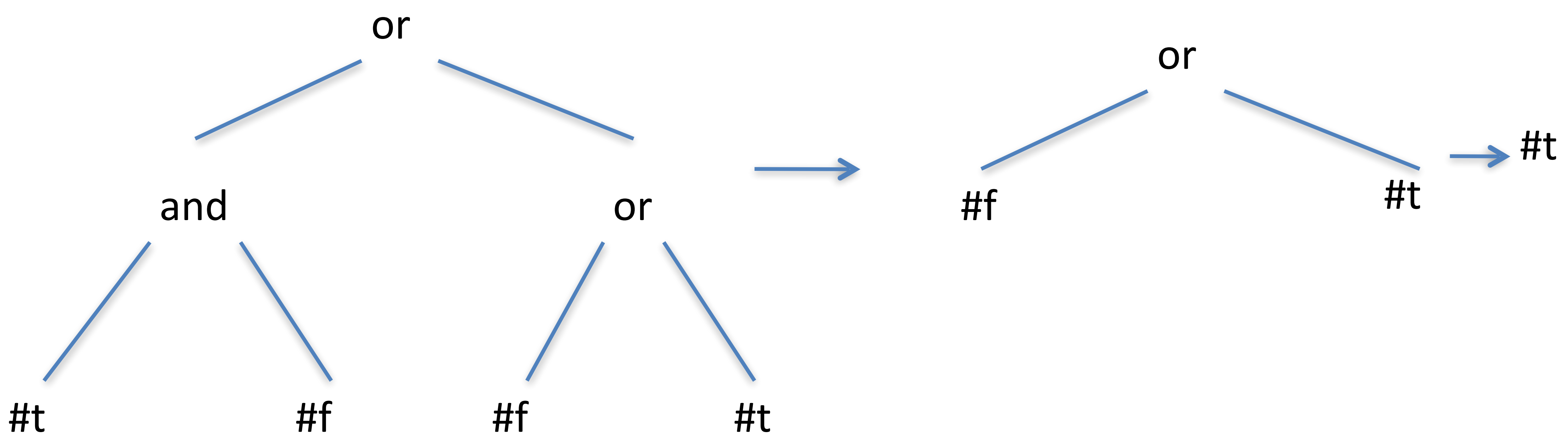


Returns true or false

- IF *can* have only one part as well:
  - (λxy. if (> x y) x)
- **Notice:** λ calculus can use all the classical boolean constructs:
  - and, or, not
    - (or #t #f)
    - #t
    - AST:



- (not #t)
  - #f
- (or #f #t)
  - #t
- (and #f #t)
  - #f
- (or (and #t #f) (or #f #t))
  - #t
- AST:



All numbers are considered #t

Why return the first item?
Efficiency: This can save unnecessary evaluations..

(or (f1 a) (f2 a) (f3 a)…(f1000 a))

Stop evaluating as soon as possible

- 31
- (or #t 31)
  - #t

Note:
This is different to many languages, e.g. zero is often *false*

Note:
'or' returns the first TRUE value it can find; otherwise it returns FALSE.

Note:
Sometimes the first TRUE value is **not** a boolean!

AND is the opposite of OR

- (and 3 -1)
  - -1
- (and -1 3)
  - 3
- (and 1 #f)
  - #f
- (and #f 2)
  - #f
- (or 1 #f)
  - 1
- (not -1)
  - #f

Efficiency of AND vs OR
AND requires everything to be evaluated for true

(and (f1 a) (f2 a) (f3 a)…(f1000 a))

Note:
As with OR, the TRUE item could be non-boolean

- Extra Arguments?
  - (not 1 2)
    - **not**: *arity mismatch…*
      expected: 1
      given: 2
  - (and 1 2 3 4)
    - 4
    - Returns the last item as it looks for a false value
  - (or 1 2 3 4)
    - 1
    - Returns the first true item as it looks for a true value

- Strings are always true
  - (and “hello” “goodbye”)
    - “goodbye”
  - (or “hello” “goodbye”)
    - “hello”

Remember:
and returns the LAST true item, **or** returns the FIRST true item

Use of Conditionals

- Decision making
- Give appearance of intelligence
  - (define pass?
    (lambda (x)
    (if (>= x 40) “pass” “fail”)))
  - (define pass2?
    (lambda (x)
    (if (>= x 40) #t #f)))

(pass? 25)
“fail”
(pass2? 25)
#f

Which is better?
pass2? because it returns a boolean

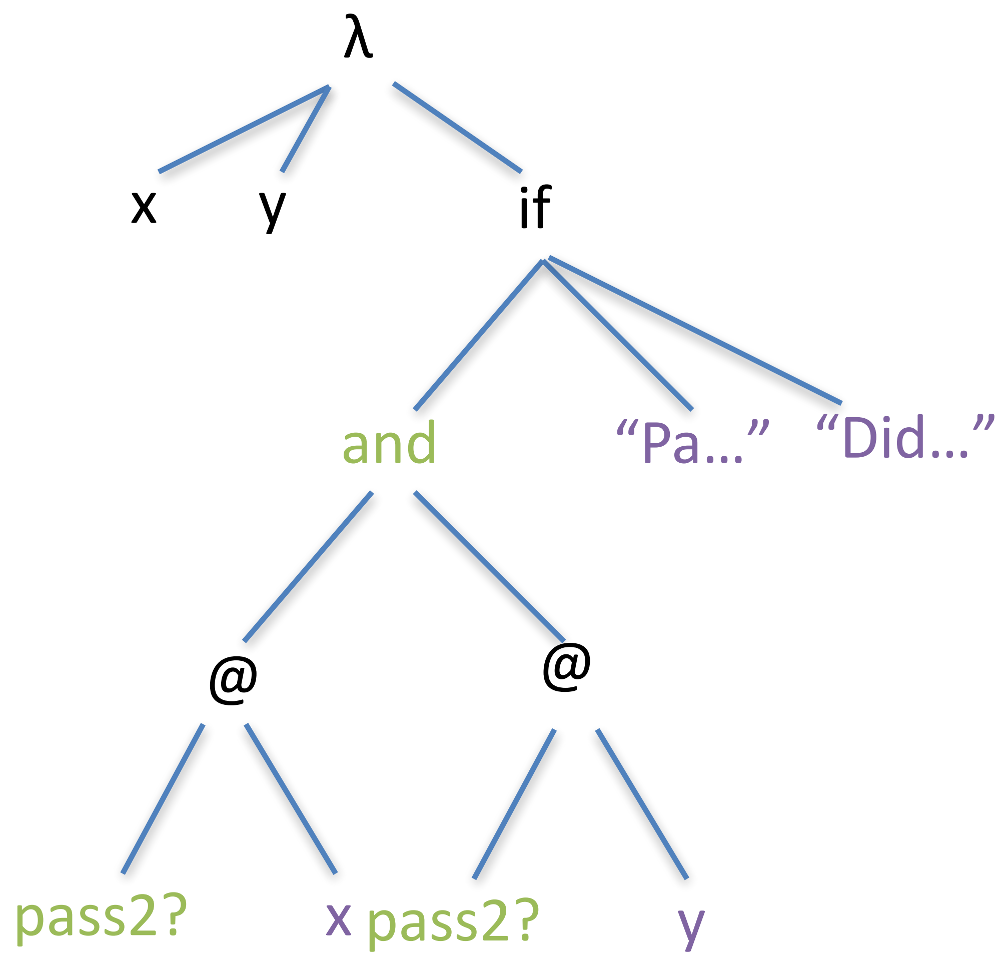
- pass? or pass2?
  - pass2? returns either #t or #f
  - pass? returns a string each time
  - A string *has* a boolean value: **#t**.

```
(if
  (and (pass? 35) (pass? 45))
  "Passed both"
  "Didn't pass both")
```

(and "fail" "pass")

"pass"… incorrect!

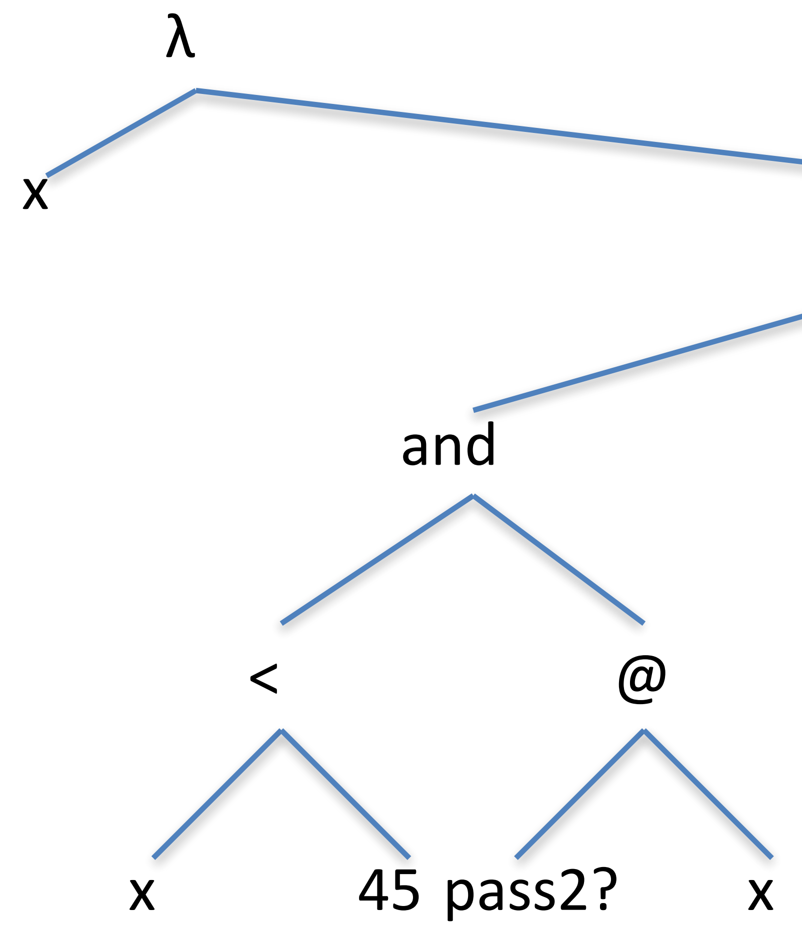
```
(define passBoth (lambda(x y)
  (if
    (and (pass2? x) (pass2? y))
    "Passed Both"
    "Didn't pass both")
    ))
```



– Another example

- (define scrape

```
(define scrape
    (lambda (x)
        (if   (and (< x 45) (pass2? x))
                              #t   #f )))
```



pass2? is **reusable**

- (scrape 42):

  (< *42* 45) → #t

  (pass2? *42*) → #t

  → (and (< x 45) (pass2? 42)) → #t

- (define pass3? (lambda (x) (>= x 40)))
  - Evaluates (>= x 40)
  - Returns the boolean value.

- More examples:
  - Write two functions
    - (1) Check if a number is even.
    - (2) Checks if a number is high-even, that is, if the number is greater than 20 ***and*** even.

high-even

- Built in Racket function:

  > (integer? **x**) ... #t if **x** is an integer,
  #f otherwise

```
> (define even
        (lambda (x)
            (integer? (/ x 2))
        )
  )

> (define high-even
        (lambda (x)
            (and (> x 20) (even x))
        )
  )
```

Note: No IF part

```
> (define high-even2
          (lambda (x)
               (if (> x 20) (
          )
)
```

Function Call Overhead
Housekeeping required for each call
e.g. set up local variables

Which is better? high-even or high-even2?

high-even2 executes a function call first, incurs "overhead"

high-even relies on short-circuiting behaviour of AND.

When **(> x 20)** returns **#f,** execution stops

Remember: AND returns the first FALSE item it finds

Therefore, high-even is better.

Recursion

Recursion

- Solve a problem with a function that calls itself
- For example, how do you calculate Factorial *n*?
- 3! = 3 * 2 * 1
- 4! = 4 * 3 * 2 * 1
- Answer: n * Factorial (n-1)
- …. kind of

Induction

- Prove for simple case
- Prove for case i+1
- Assume true for all

Inductive proof for dominoes:

- Informal
  - The first domino knocks over the second
  - which knocks the third
  - and so on ....

- Classic
  - The first domino falls
  - Whenever the ***i***th domino falls, it knocks the ***i+1***th domino
  - Therefore, all the dominoes fall.
- Idea
  - Can prove something for a simple case
  - Prove it for a general case
  - Assume proven for all cases
- Important because?
  - Numbers go to infinity
  - Impossible to prove for every case

The Joy of Logic
It lets us reason about unseen cases

Recursion is similar to Induction

- Recursion
  - Solve *simple case* of a problem
  - Figure out how complex (*general*) case can be solved
  - ….using the simple case
  - Magically solves all cases
- Example: Compute Factorial
  - Factorial 1 = 1 (Simple Case)
  - Factorial n = n * (n-1) * (n-2) *… * 1
  - Factorial n-1 = (n-1) * (n-2) *… * 1
  - Factorial n = n * Factorial (n-1) (General Case)

- Factorial 1 = 1 [SIMPLE CASE]
- Factorial n = n * Factorial (n-1) [GENERAL CASE]
- Fact 3: *(shorthand for Factorial 3)*
  - Fact 3 = 3 * Fact 2
  - Fact 2 = 2 * Fact 1
  - Fact 1 = 1
- Go back up:
  - Fact 2 = 2 * 1
  - Fact 3 = 3 * 2 * 1
- Answer = 6.
- Each line:
  - Does ONE thing
  - Passes on the rest of the problem (to itself)

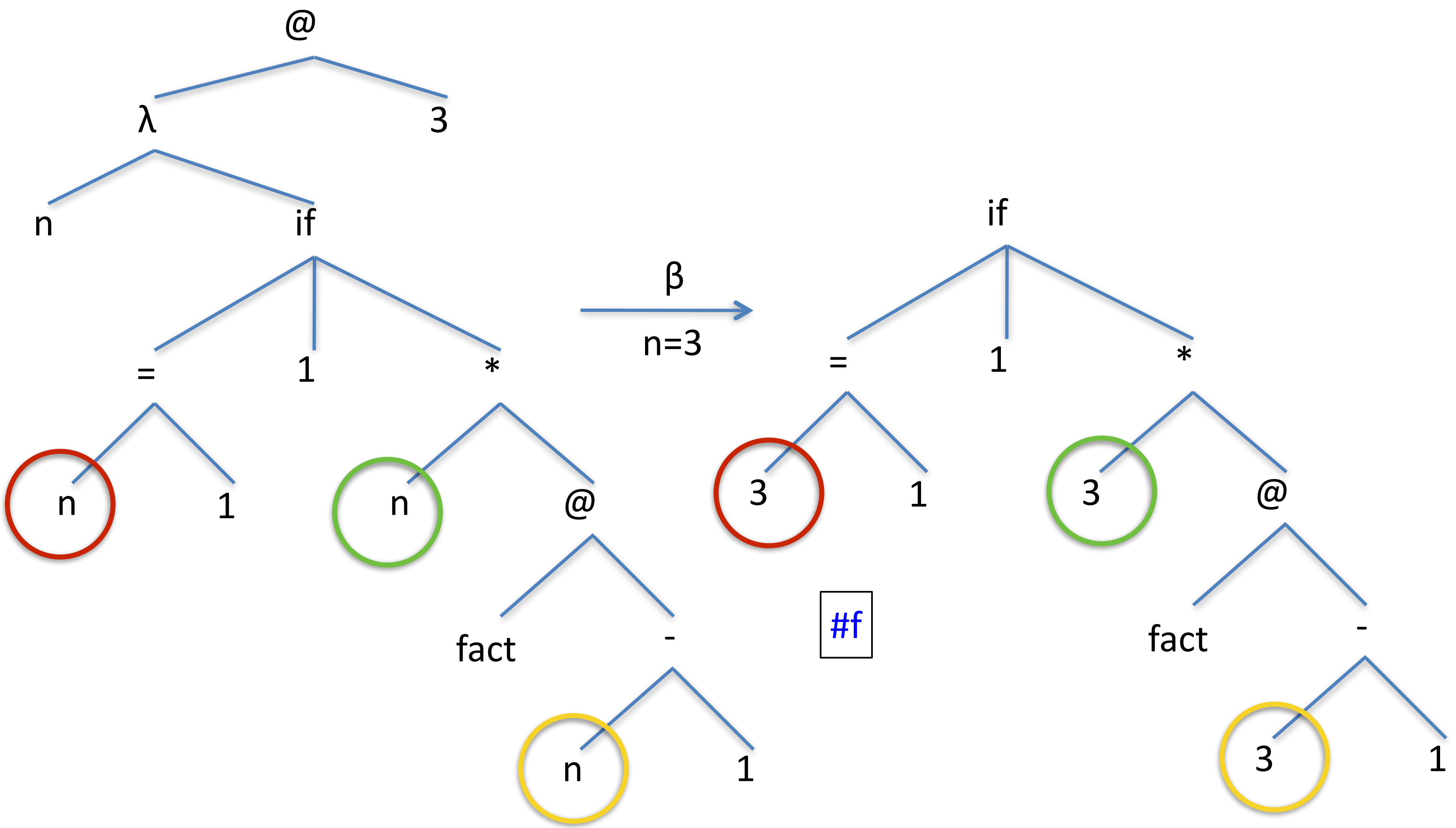
Implementation and Execution

- fact : (λn. if (= n 1)
  1
  (* n (fact (- n 1))))
- Execute (fact 3):
  - if (= 3 1) 1 (* 3 (fact (- 3 1)))
  - (* 3 (fact (- 3 1))) = (* 3 (fact 2))
- Execute (fact 2):
  - if (= 2 1) 1 (* 2 (fact (- 2 1)))
  - (* 2 (fact (- 2 1))) = (* 2 (fact 1))
- Execute (fact 1):
  - if (= 1 1) 1 (* 1 (fact (- 1 1)))
  - 1

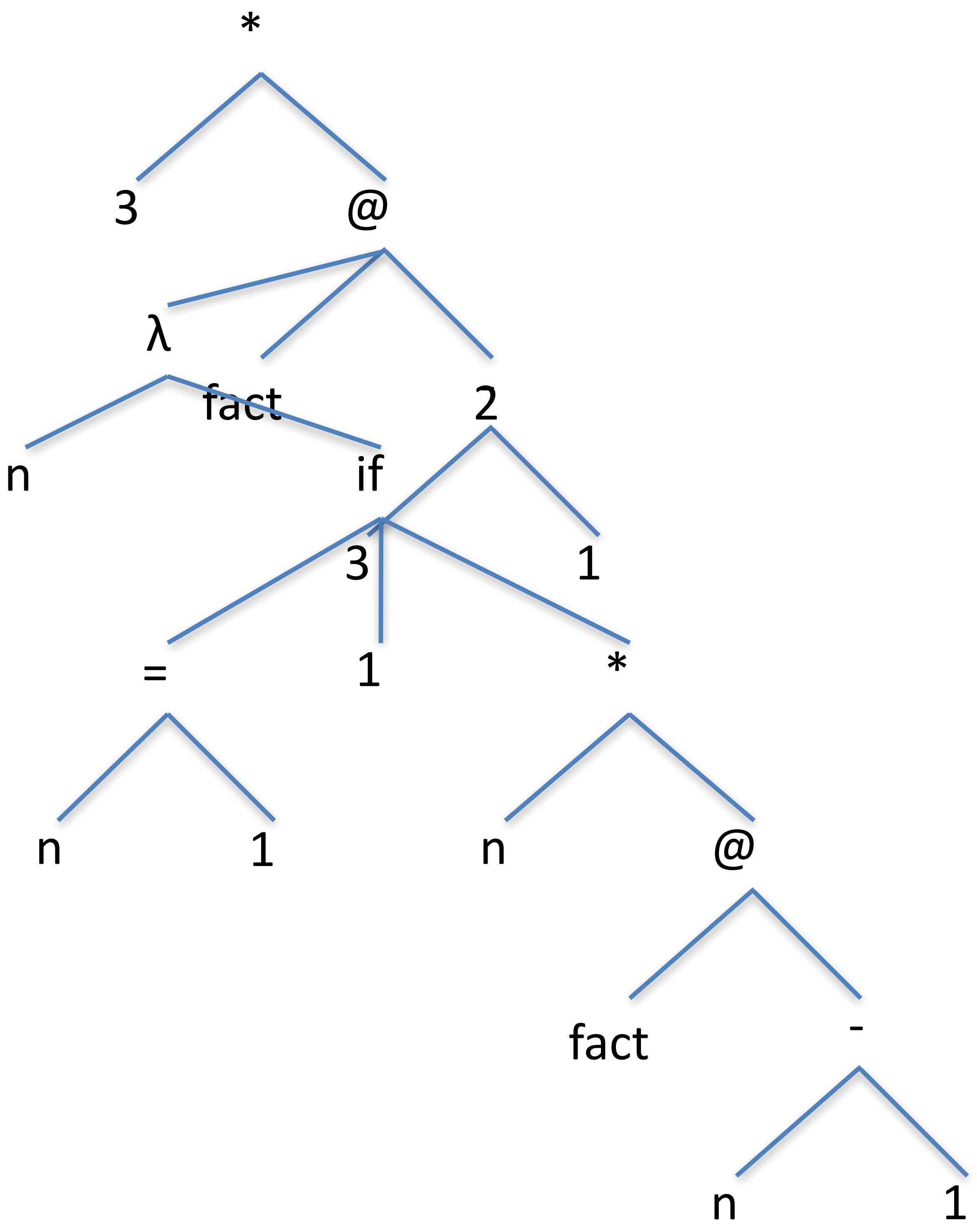
Recursive Call

- Go back:
  - (* 3 (fact 2)) = (* 3 2)
  - $\boxed{\text{6 Final Answer}}$
- Go back:
  - (* 2 (fact 1)) = (* 2 1)
  - 2

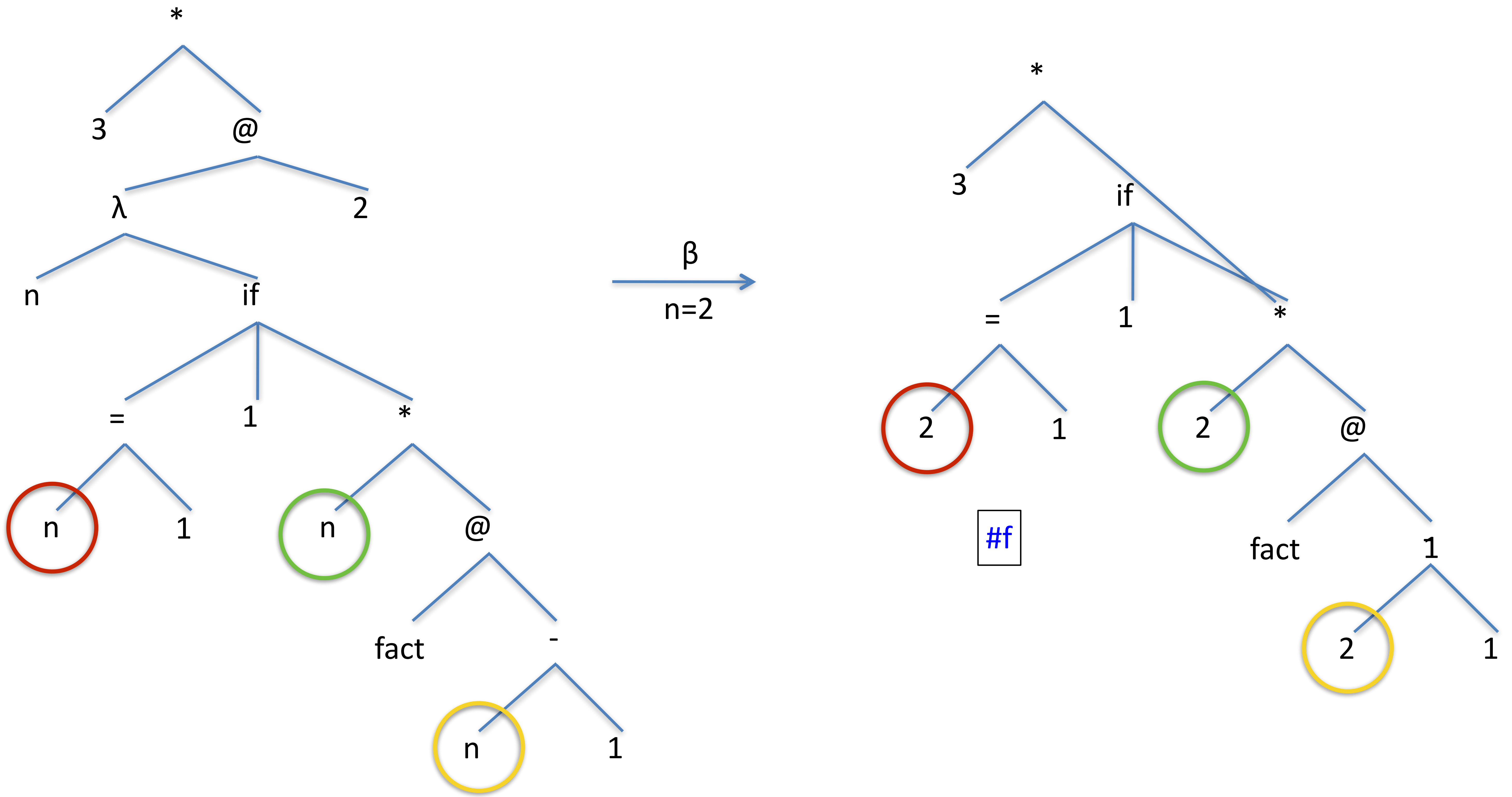
AST (fact 3)



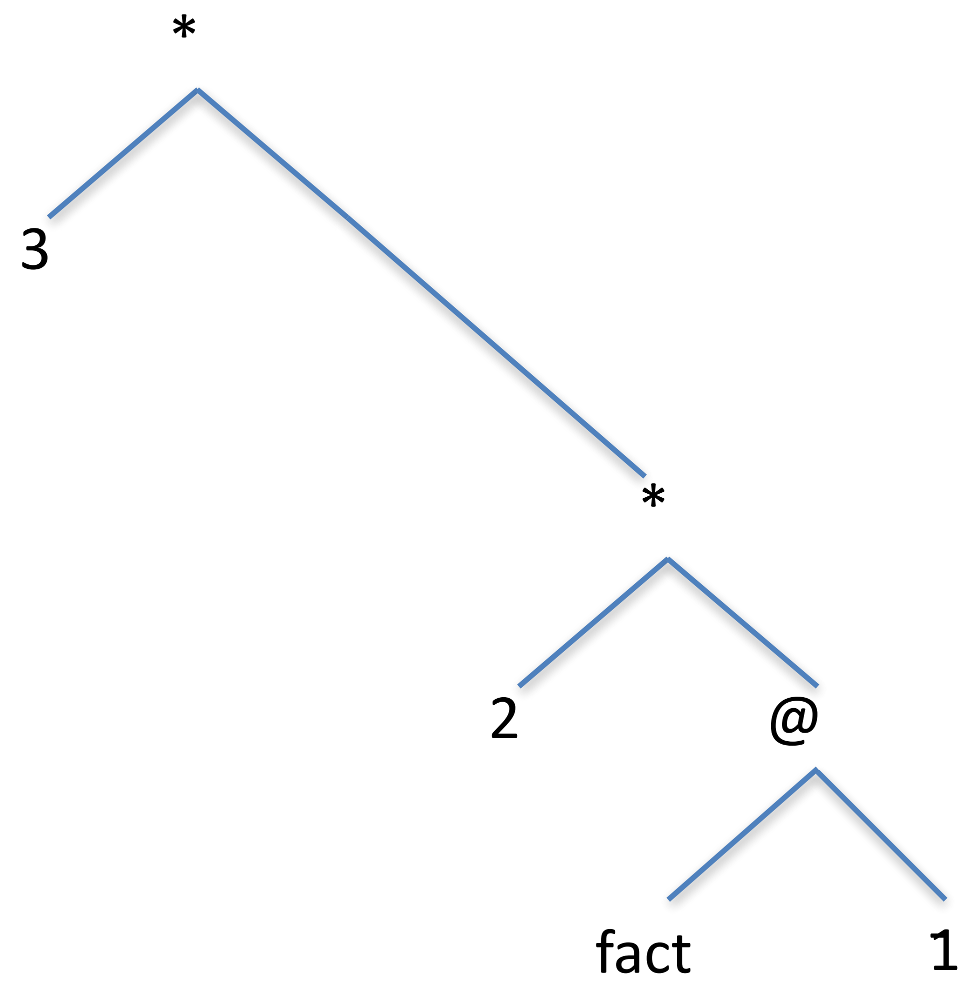
AST (fact 3)



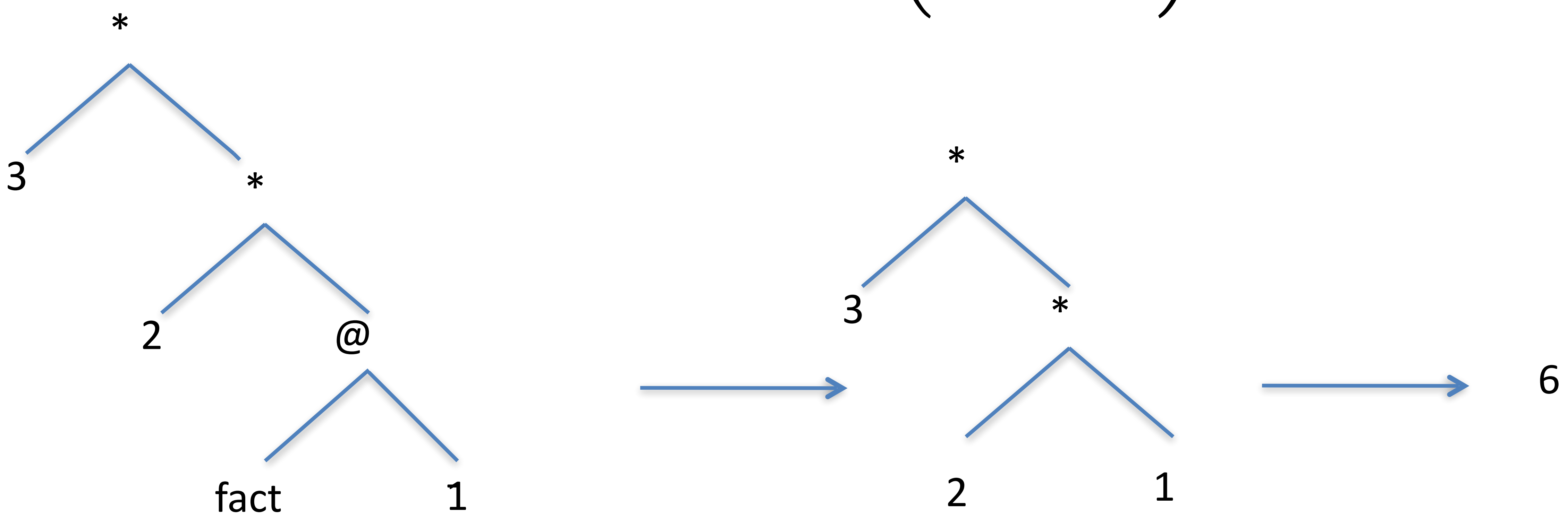
AST (fact 3)



AST (fact 3)

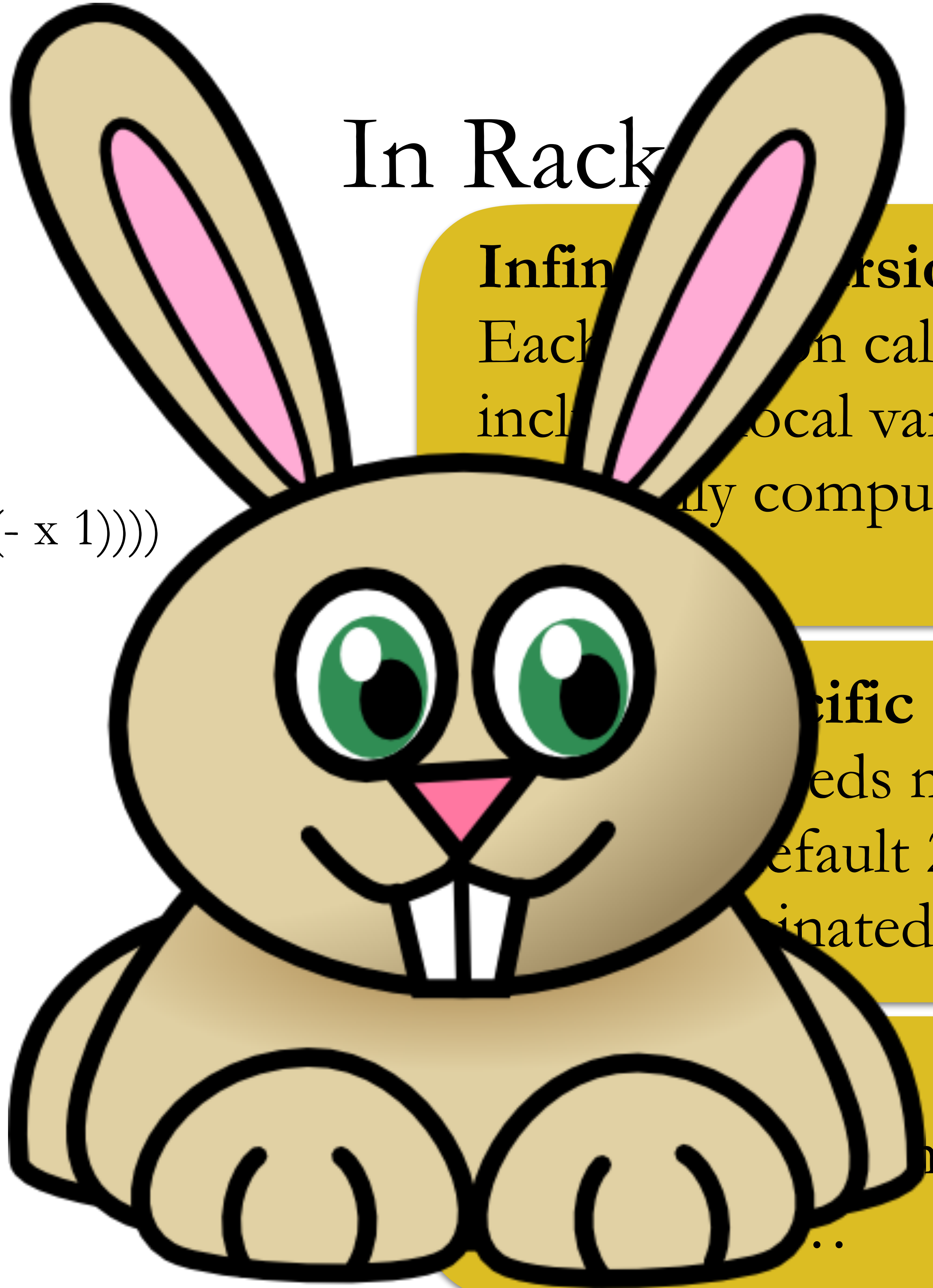


AST (fact 3)



In Rack

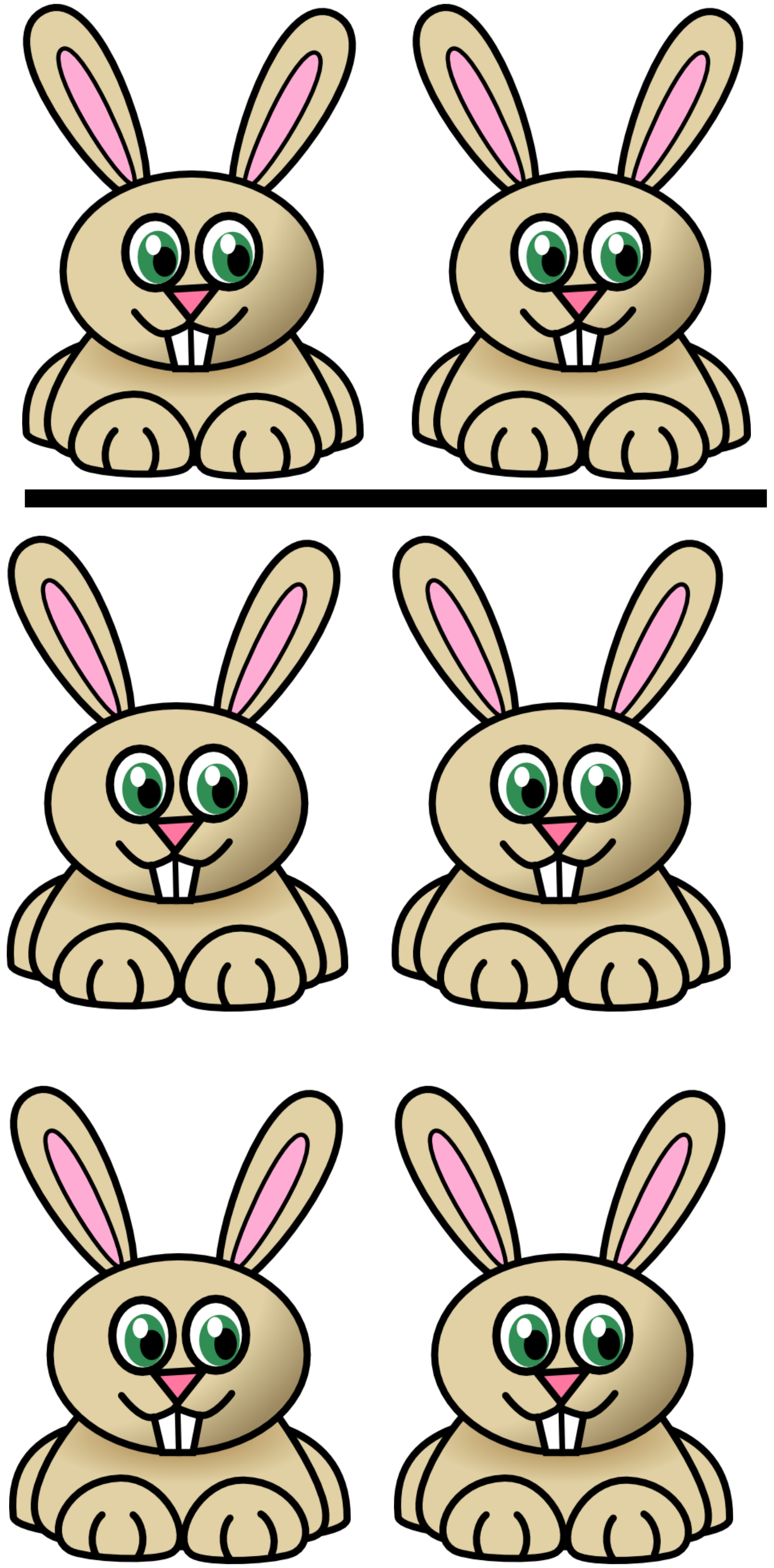
- (define fact
    (lambda (x)
        (if (= x 1)
         1
         (* x (fact (- x 1))))
)
)
  - (fact 3)
    - 6
  - (fact 5)
    - 120
  - (fact -1)
    - Infinite recursion



Infin rsion
Each n call incurs overhead,
incl ocal variables…
ly computer runs out of

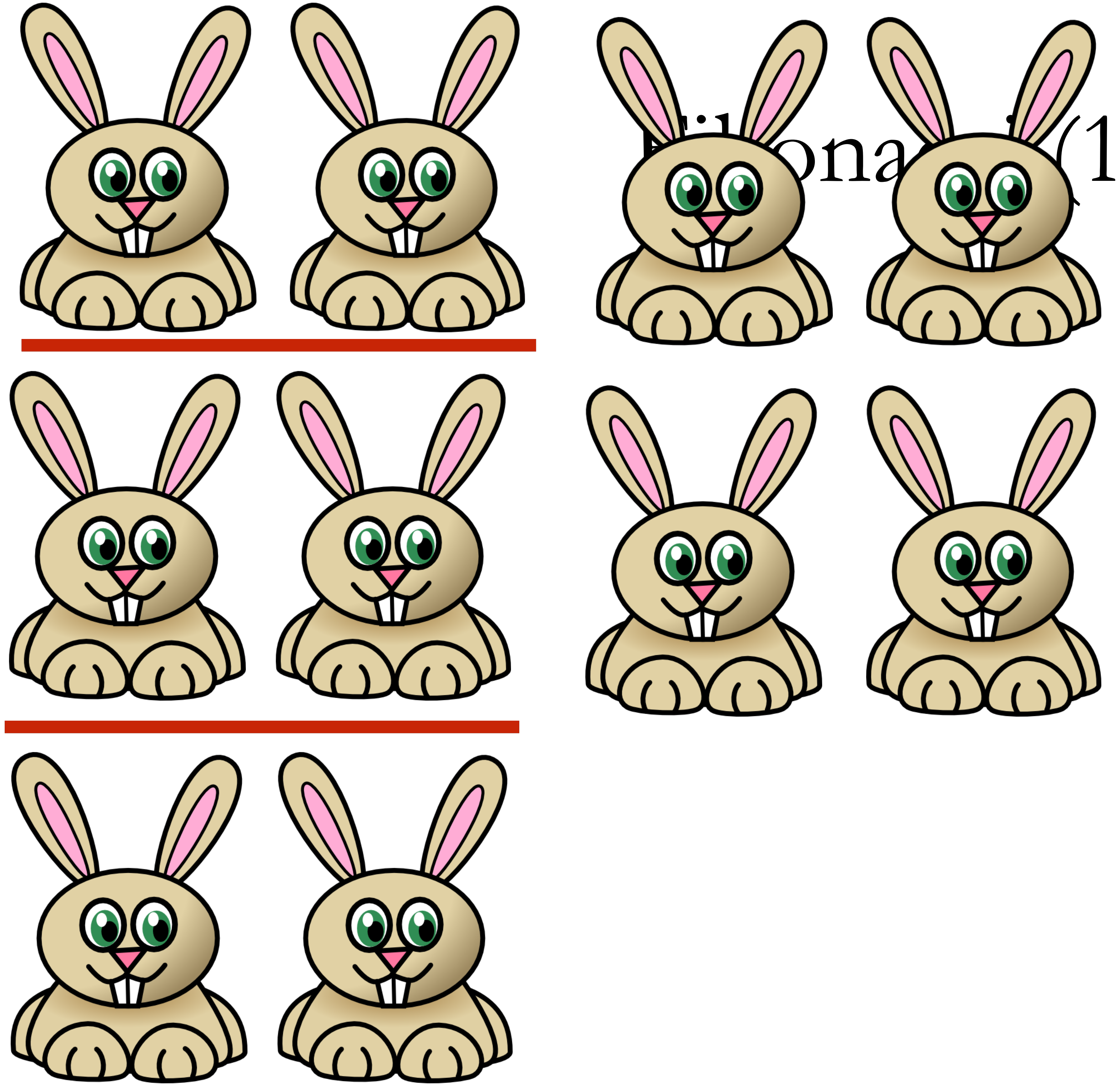
cific
eds maximum *allowed*
efault 240MB)
inated.

n to
..

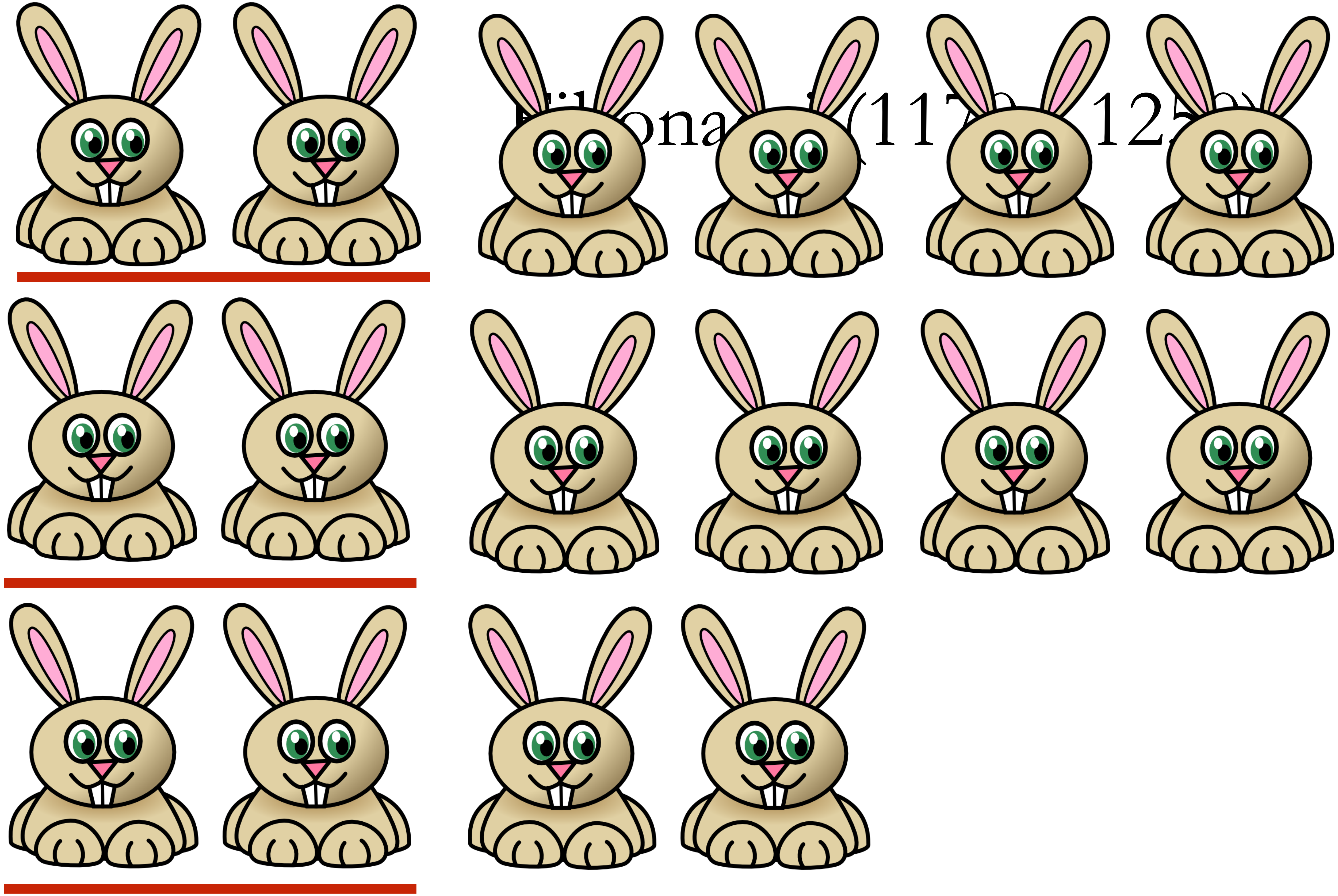


Fibonacci (1170 - 1250)

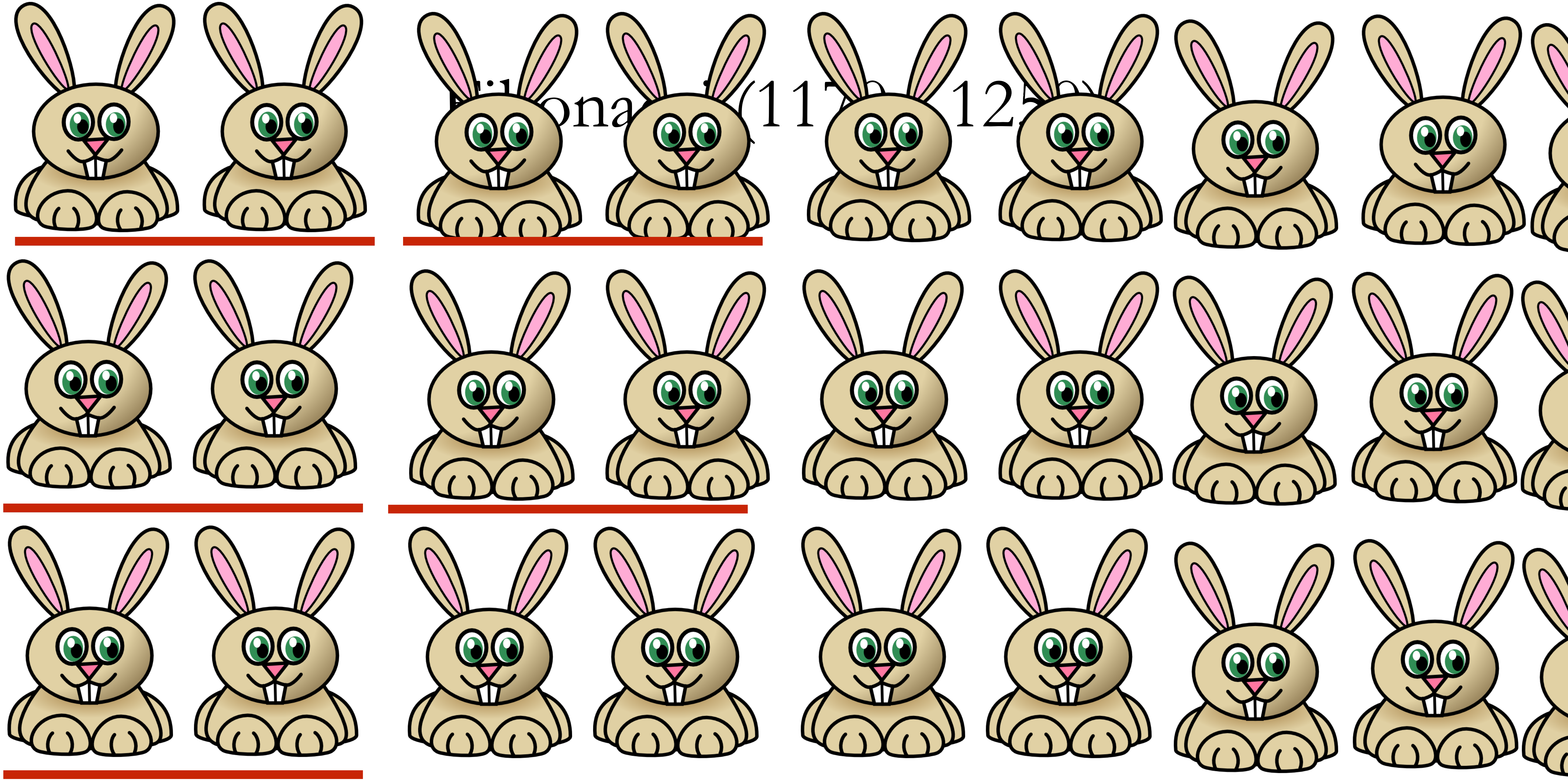
Fibonacci (1170 - 1250)



Fibonacci (1170 – 1250)



Fibonacci (1170 - 1250)



Fibonacci Series

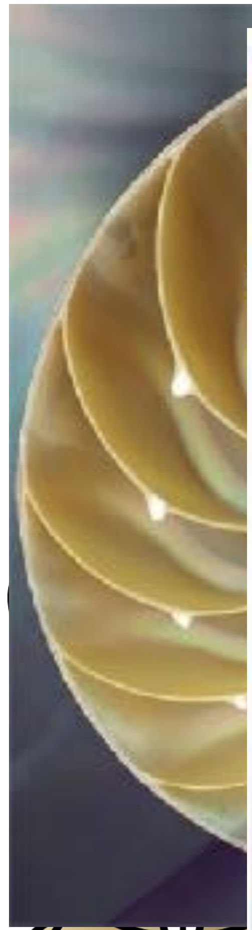
Model of rabbit population growth

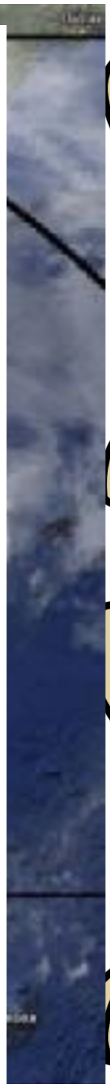
- Start with one pair
- Rabbits can mate at the age of one month
- Gestation period is one month
- Two rabbits produced each time
- Equal number of male and female rabbits
- Rabbits never die

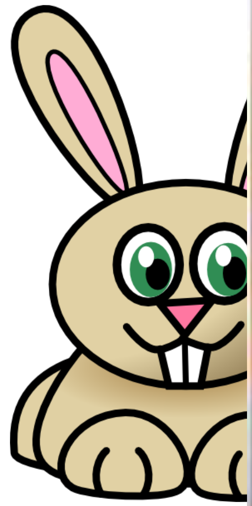
How many pairs will there be in one year?

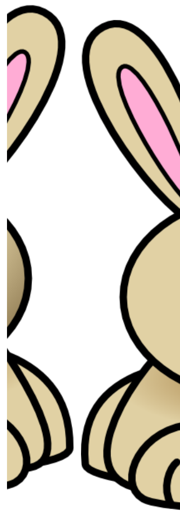
F0	F1	F2	F3	F4	F5	F6
0	1	1	2	3	5	8

$$F_n = F_{n-1} + F_{n-2}$$









Fibonacci Series

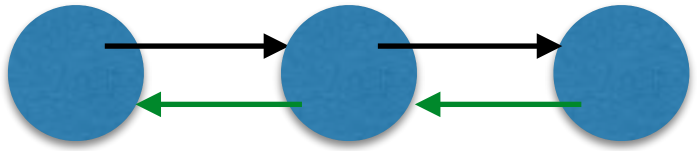
```
(define fib
    (lambda (x)
        (if (<= x 2)
        1
        ( + (fib (- x 1))
            (fib (- x 2))
        )))
```

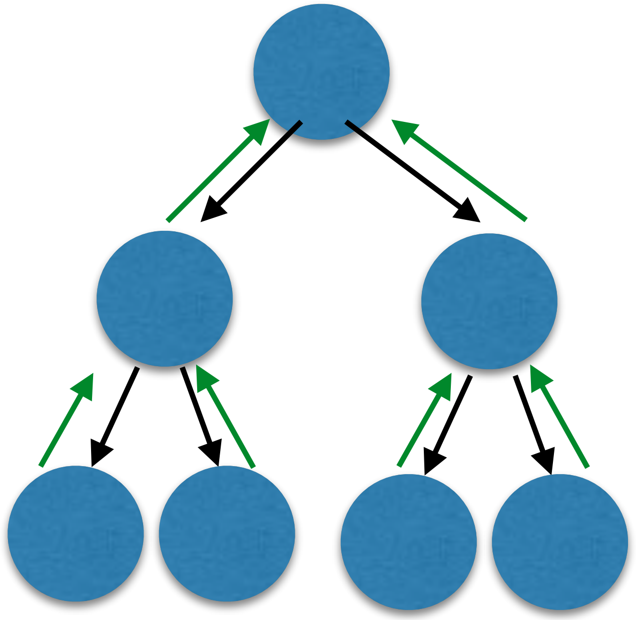
Base Case

General Case

Fibonacci Series

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



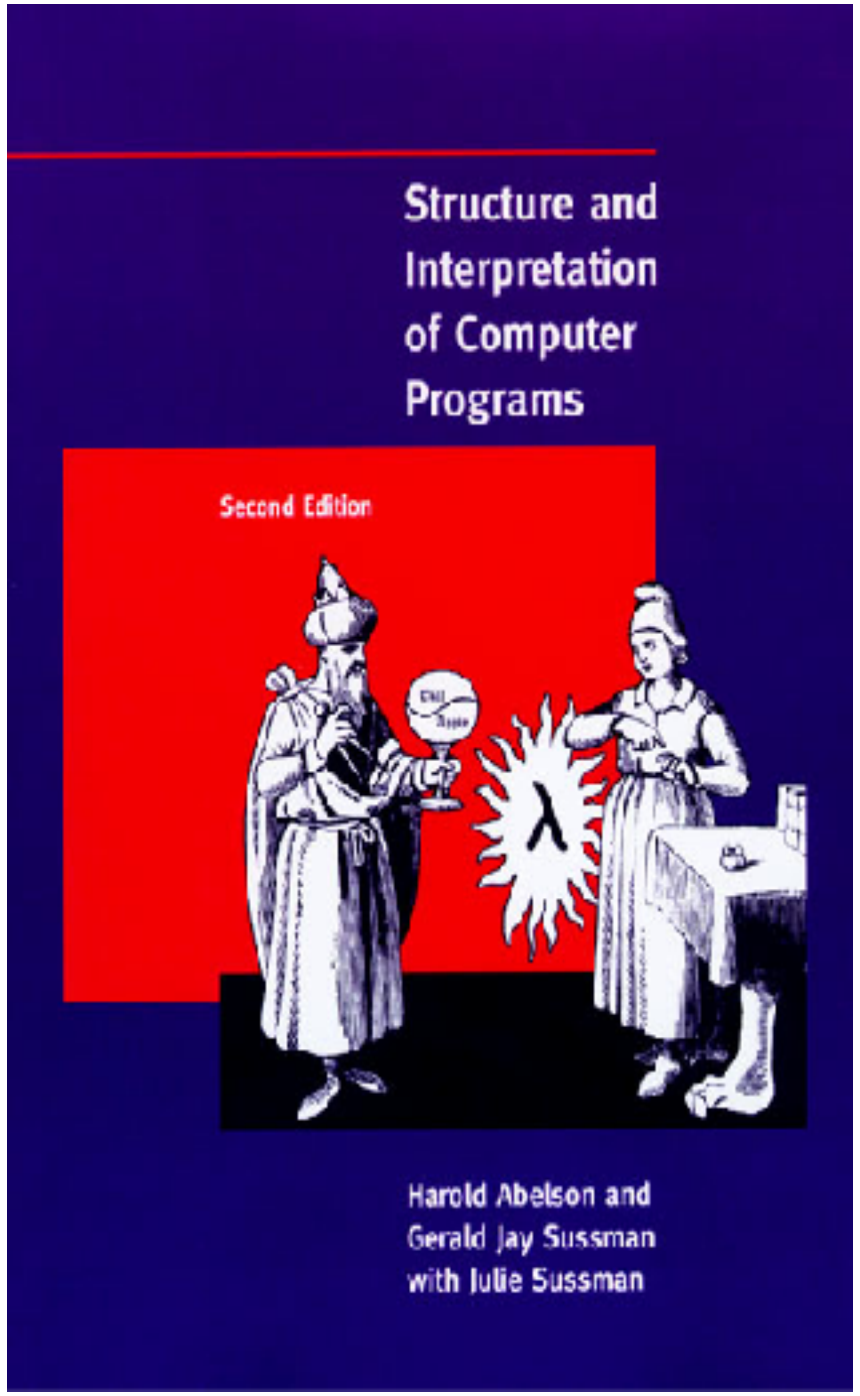


Base Case

General Case

Additional Reading on Recursion

- Given in “Reference Material”
  - On the class website
  - **<u>Section 1.2 Procedures and the Processes They Generate</u>**
- Implement and understand two different implementations of **factorial.**
- Also attempt Exercise 1.9.



Recursion and Iteration (loops)

- Iteration *may sometimes* replace recursive function

```
int fact=1;
for (int j=arg; j>1; j--)
 fact = fact * j;
```

Web crawler/spider/Googlebot
Visit every page in a hierarchy

- But not always!
  - Sometimes not trivial to replace a recursive function.
    - For example browsing a tree of item categories on argos.ie or amazon.com
    - Useful exercise: implement Fibonacci in Java

Fibonacci Loop in Java

```
public static int fibonacciLoop(int number) {
        if (number == 1 || number == 2) {
                return 1;
        }
        int fibo1 = 1, fibo2 = 1, fibonacci = 1;
        for (int i = 3; i <= number; i++) {
                fibonacci = fibo1 + fibo2;
                fibo1 = fibo2;
                fibo2 = fibonacci;

        }
        return fibonacci;
}
```

Base Cases

Initialise some variables

Sum of two previous numbers

Prepare for next iteration

Final result

Run time

number	fibonacci	fibo1	fibo2
(initial)	1	1	1
3	2	1	2
4	3	2	3
5	5	3	5
6	8	5	8
7	13	8	13

Solving problems recursively

- Identify the base case; then identify the general case
- Not always easy
  - General case may be difficult to formulate
- Example: Add numbers from *0* … *n*.
  - Base Case/Terminating Case/Simple case
    - *0* … nothing to add
    - i.e. sum(0) = 0
- General case:
  - sum(n) = n + (n – 1) + (n – 2) + …. + 0
  - sum(n-1) = (n – 1) + (n – 2) + …. + 0
  - Thus, sum(n) = n + sum(n-1)

Solving problems recursively

- Putting it together:
  sum: λn. if (= 0 n) 0
  (+ n (sum (- n 1)))
- Another view: recognise a sequence
  - n 0 1 2 3 4 ...
  - sum(n) 0 1 3 6 10 ...

  - n 0 1 2 3 4 ...
  - fact(n) 0 1 2 6 24 ...
  - Write a recursive function that generates the sequence
  - i.e. for a given value of n, it produces sum(n) or fact(n).

Generating Functions from Sequences

- Using λ calculus & recursion for design
  - Try to describe what is happening with sequence
- Example: explain the following sequence
  - n 1 2 3 4 ...
  - f (n) 1 5 9 13 ...
  - Base?
    - f(1) = 1
  - General?
    - No easy way to spot; however, *usually* f(n) is somehow related to f(n-1)
    - **Here**, each number is 4 bigger than the previous one.
    - Therefore, f(n) = f(n-1) + 4

- Mathematically:
  - $f(1) = 1$
  - $f(n) = f(n-1) + 4$
- Recursive λ calculus function:
  - f : λn. if (= n 1) 1
    (+ (4 (f (- n 1))))
- Another example:
  - n 1 2 3 4 ...
  - f (n) 1 5 13 29 ...
  - f(1) = 1. (won't always be ....)
  - f(n) = ?
  - *Usually* f(n) = calc(n) + f(n-1) (but not always...)
- Write out:
  - n 1 2 3 4 ...
  - f (n) 1 5 13 29 ...
    - $f(1) = 1$
    - $f(2) = 5 = f(1) + 4$
    - $f(3) = 13 = f(2) + 8$
    - $f(4) = 29 = f(3) + 16$
    - $f(5) = 61 = f(4) + 32$
  - 4, 8, 16... powers of 2.
- Aside:
  - Power of two in λ calculus?
    - λx. (* x x)
    - Only squares; need to generate higher powers of two

- A more useful function:
  - (pow x y)   (i.e. x^y)
  - Base case: $x^0 = 1$, thus (pow x 0) = 1
  - General case: (* x (pow x (- y 1))
    - Because $x^y = x * x^{y-1}$
- Notice:
  - Two variables
  - Only one controls recursive call
- Recursive λ calculus function:
  - pow : λxy. if (= 0 y) 1
    (* x (pow x (- y 1)))

Trace Execution

pow : λxy. if (= 0 y) 1
(* x   (pow x (- y 1)))

Each recursive call to f uses another recursive function (pow)

- Evaluate 3^2
  - (pow 3 2)
  - if (= 0 2) 1 (* 3 (pow 3 1))
    if (= 0 1) 1 (* 3 (pow 1 0))
    if (= 0 0) 1 (* 3 (pow 3 -1))
  - (* 3 (* 3 1)) = 9
- Back to original question:
  - f(1) = 1
  - f(n) = f(n-1) + 2^n
- Recursive λ calculus function:
  - f : λn. if (= n 1) 1
    (+ (f (- n 1)) (pow 2 n))

Procedures and Processes

- *Procedures*: another term for functions.
- Function call generates a computational *process*
  – i.e. a set of steps required to execute the code
- Important to understand this process to become an expert programmer
  - i.e. not all code is executed
  - sometimes code is executed multiples times
- Possible to examine the *shape* it generates.

- Reminder: **factorial**
  - (fact n) = (* n (fact (- n 1))) (General Case)
  - Example execution: (fact 4)

Space (Memory Required)

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2  1)))
(* 4 (* 3  2))
(* 4  6)
24
```

Time

<u>No. of steps</u>

2***n** = 2 * **4**

Memory Footprint

The maximum amount of memory used

Factorial with a non-recursive process

- Avoid deferred operations:
  - Keep a running product with every recursive call
  - Much like with loops/iterations. Recall:

```
int product=1;
int counter=1;
while (counter <= n){
   product = product * counter;
   counter++;
}
```

Counter	Product
1	1
2	2
3	6
4	24

<u>Note</u>: No deferred operations => iterative process.

Factorial: non-recursive process with a recursive function

- (facto counter product ***n***): (say ***n*** = 4)

Space (Memory Required)

steps
n = 4

Time

  - (facto 1 1 ***4***)
  - (facto 2 2 ***4***)
  - (facto 3 6 ***4***)
  - (facto 4 24 ***4***)

n is the factorial we're calculating
product is the running total
counter is the number of steps

- Constant Memory/space required
  - Because no deferred operations.
  - Much like with loops.
  - Hence, an ***<u>iterative process</u>***.

A recursive function with an iterative process

```
(define facto (lambda (counter product n)
(if (> counter n)
  product
  (facto  (+ counter 1)
        (* counter product)
        n
  )
)
)
)
```

Final Exam

- 2.5 hours
- Answer four out of five questions
- 2 questions involving recursion.
- All material is examinable
  - Some questions based on practical/tutorial questions
  - Some general questions
  - Some definitions
  - Understanding rather than memorising

Final Exam

- Read questions carefully before starting
- Revisit mid-term questions carefully.
- Show **all** your work
- Calculators are permitted, but only actual calculators
- **Always** explain definitions with examples.
- No labs/tutorials in week 13.
- Check class website every day before the exam