CS4221 - Computer Science

Lecture Set 2: Design

- Writing programs
  - Java?
  - Too inconsistent
    - Mix of pre/post/infix (“mixfix”)
    - i++; ++i; i+i;
    - i=i+++++i;
  - Lots of extra syntax
    - public static void main(String[] args)..

If i was 3

  Answer = 8.

  i++ gives 4

  ++i gives 4

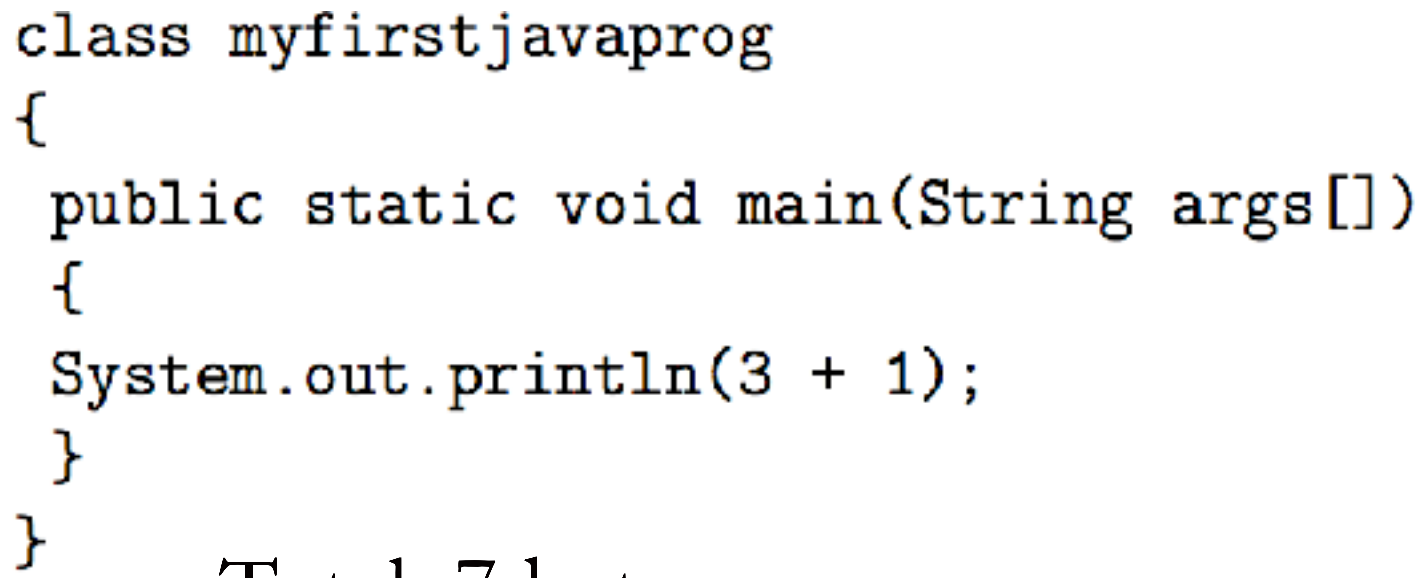
  Addition gives 8.

j=i+++++i;

  i = 5

- Lots of extra syntax
    - public static void main(String[] args)..
- “Syntactic Sugar”
    - Doesn’t enhance functionality
    - (Allegedly) makes program easier to read
- Is there any place for Imperative Programming?
    - Yes, its just not as useful as you might believe...
- Would Dermot agree?
    - Probably not...

- Functional Programming
  - Write programs based on expressions/functions



```
class myfirstjavaprog
{
 public static void main(String args[])
 {
 System.out.println(3 + 1);
 }
}
```

Source: 133 bytes.
Binary: 14,578 bytes.

  - > (+ 3 1)

Total: 7 bytes.

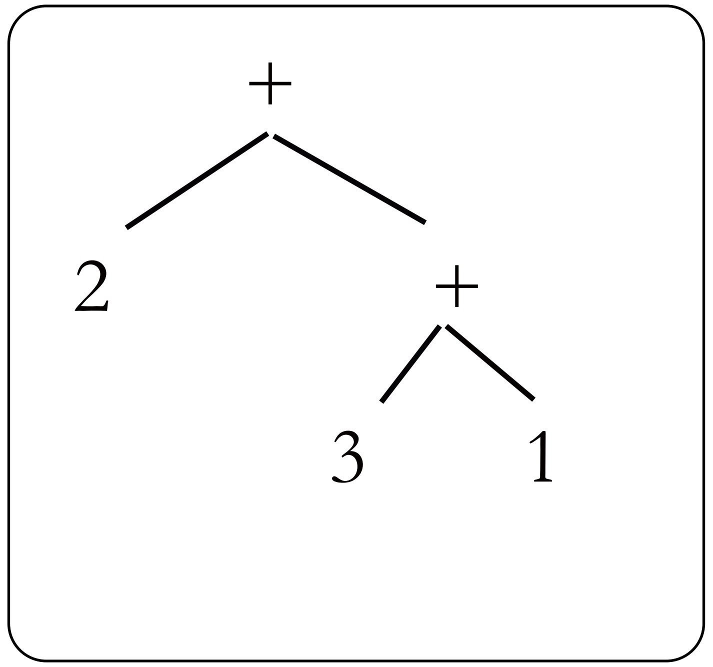
  - Very little syntax
    - Short programs
    - Mobile Devices
    - Embedded controllers

- Relevance?
  - (Almost) anything can be expressed with an AST.
  - Good FP language copies AST with little overhead
  - Racket
    - Dialect (subset) of *Lisp*
    - http://racket-lang.org/
  - Imperative languages (Java/C++) must be compiled
  - Racket
    - Compilable
    - Interpretable

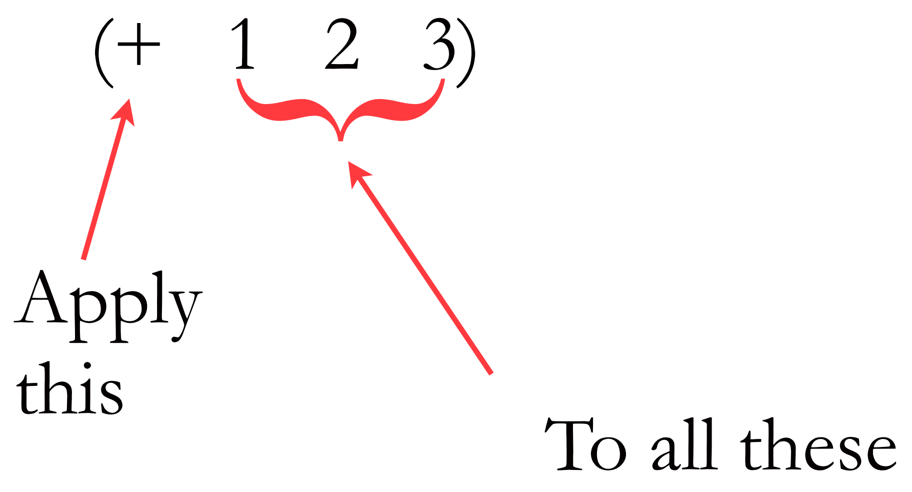
- Used interactively
- Machine reacts to each line of I/P
- Very fast to develop

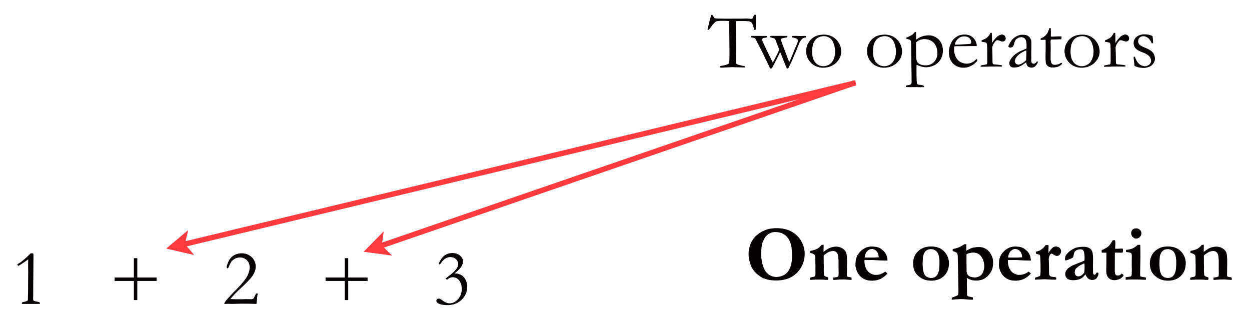
- >
- > (+ 3 1)
- 4
- + 3 1
- => Error (sort of)
- + 2 + 3 1
- > (+ 2 (+ 3 1))
- Clearer what operates on what

Always use brackets



- (+ 1 2 3)
- 6
- Not all operators are binary
  - Yet another reason not to use infix





How many operands do operators take?
- Depends..
  - Only one (sqr, sqrt)
  - Only two (divide)
  - One or more (most)

- (- 1).. -1
- (- 3 2 1) .. 0
- (- 5 1 1)
  - 3
- Take *tail* of list of numbers from *head*
- $(- \underset{\uparrow \atop \text{Head}}{8}\ \underbrace{7\ 6..)}_{\text{Tail}}$

- Heads and tails
  - car = first item in list
  - cdr = rest of the list
  - (car (a b c d))
  - a
  - (cdr (a b c d))
  - (b c d)
  - (car ($\underline{\text{cdr (a b c d}}$))
  - (car (b c d))
  - b

- Expressions vs. Lists
- (+ 1 2) vs. ‘(+ 1 2)
- 3 vs. (+ 1 2)
- (car ‘(a b c d))
- (cdr ‘(a b c d)
- (reverse ‘(a b c))
- (c b a)
- (list ‘a ‘b)
- (a b)

- Examples of ‘
  - > (+ 2 1)
  - 3
  - > ‘(+ 2 1)
  - (+ 2 1)
  - > ‘b
  - b
  - > b
  - Error: reference to undefined identifier: b

- (- 10 3 2 1)
- Algorithm
  - Get car and cdr
  - Subtract first item in tail from the head
  - (subtract car of the cdr from the car)
  - car: 10; cdr: (3 2 1);car of cdr: 3.
  - set head equal to answer, i.e. 7
  - Repeat while there is something in the tail

Expression	car	cdr	car (cdr)
(- 10 3 2 1)	10	(3 2 1)	3
(- 7 2 1)	7	(2 1)	2
(- 5 1)	5	(1)	1

- Divide
  - (/ 20 5 4 2)
  - Divide head by each item in the tail
  - (/ 4 4 2)
  - (/ 1 2)
  - ½
  - Note! Not 0.5.

Precision

0.5 == ½

0.33 != ⅓

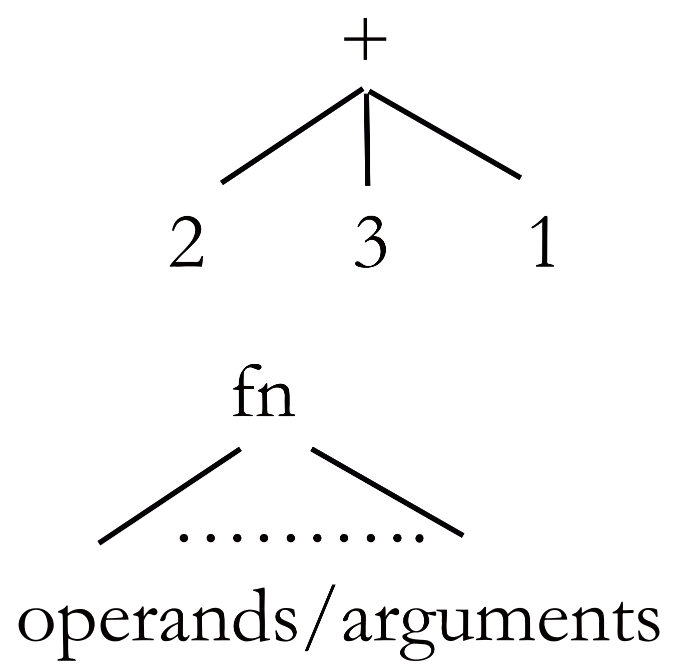
- car and cdr for Racket...
- > (+ 1 2)
- >3
- > ’(+ 1 2)
- > (+ 1 2)
- > (car ’(a b c d))
- > a
-   > (cdr ’(a b c d))
-   > (b c d)
-   > (cdr (car ’(a b c d))
- **Error**

car returns an ITEM
cdr expects a LIST

```
> (car ’(+ 1 2))
>+
> (car (+ 1 2))
cdr: expects argument of type <pair>;
given 3
> (car (a b c d))
```
Error: **a undefined**

- cdr: expects argument of type <pair>; given a

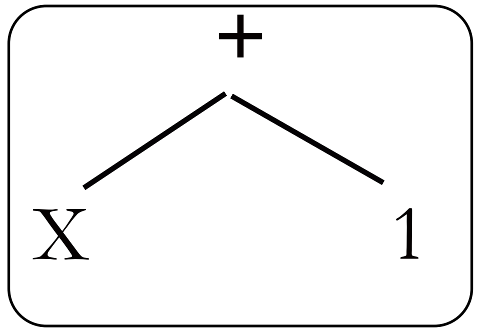
- Implications for ASTs?
  - Smaller
  - General form of AST



  - In Racket
    - (function arguments)
  - Implication
    - prefix notation ≡ AST ≡ Racket

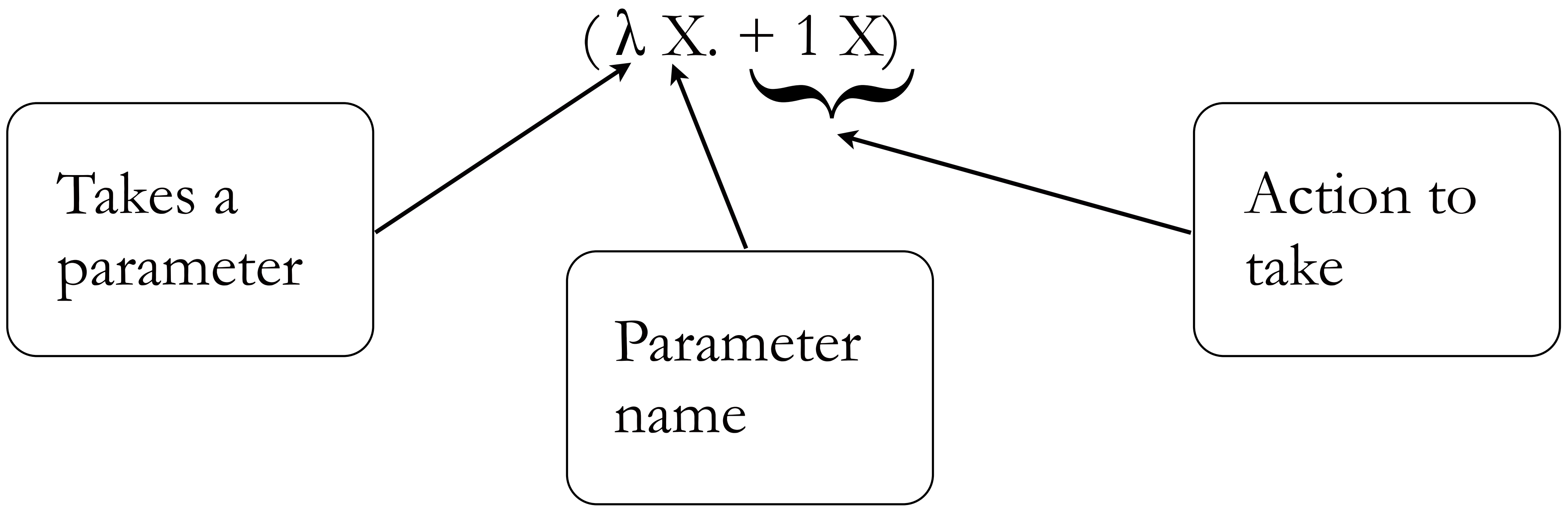
- So far
  - All operators are built in (+, -, * etc.)
  - All arguments/operands are numbers
  - AST gives the same answer
- Dynamic?
  - Behaviour depends on user input
  - Works the same way
  - e.g. Add 1 to X
  - Where does X come from?
  - Difference between
    - Add 1 to X
    - Given a value for X, add 1 to it.

$$\frac{Y_2 - Y_1}{X_2 - X_1}$$

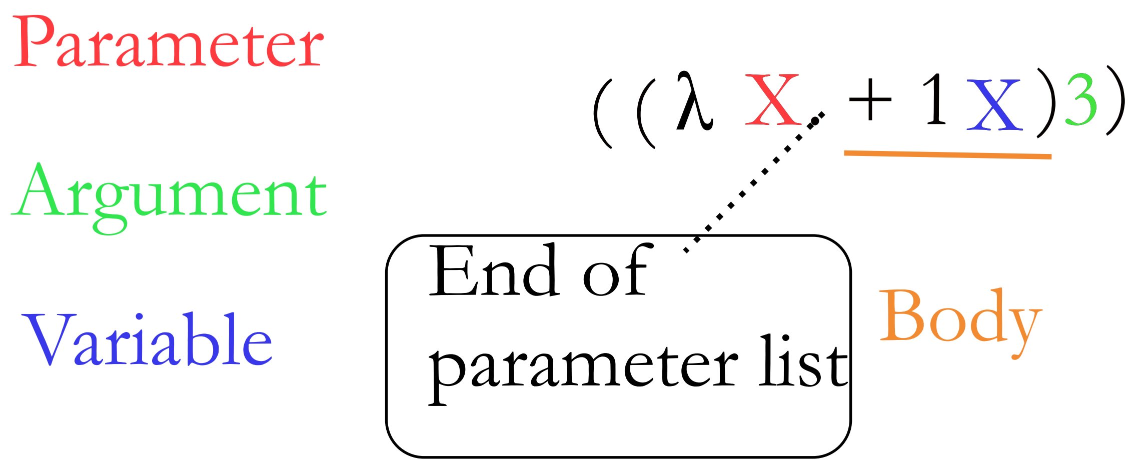


X must explicitly be given a value first

- In prefix, use Lambda Expression



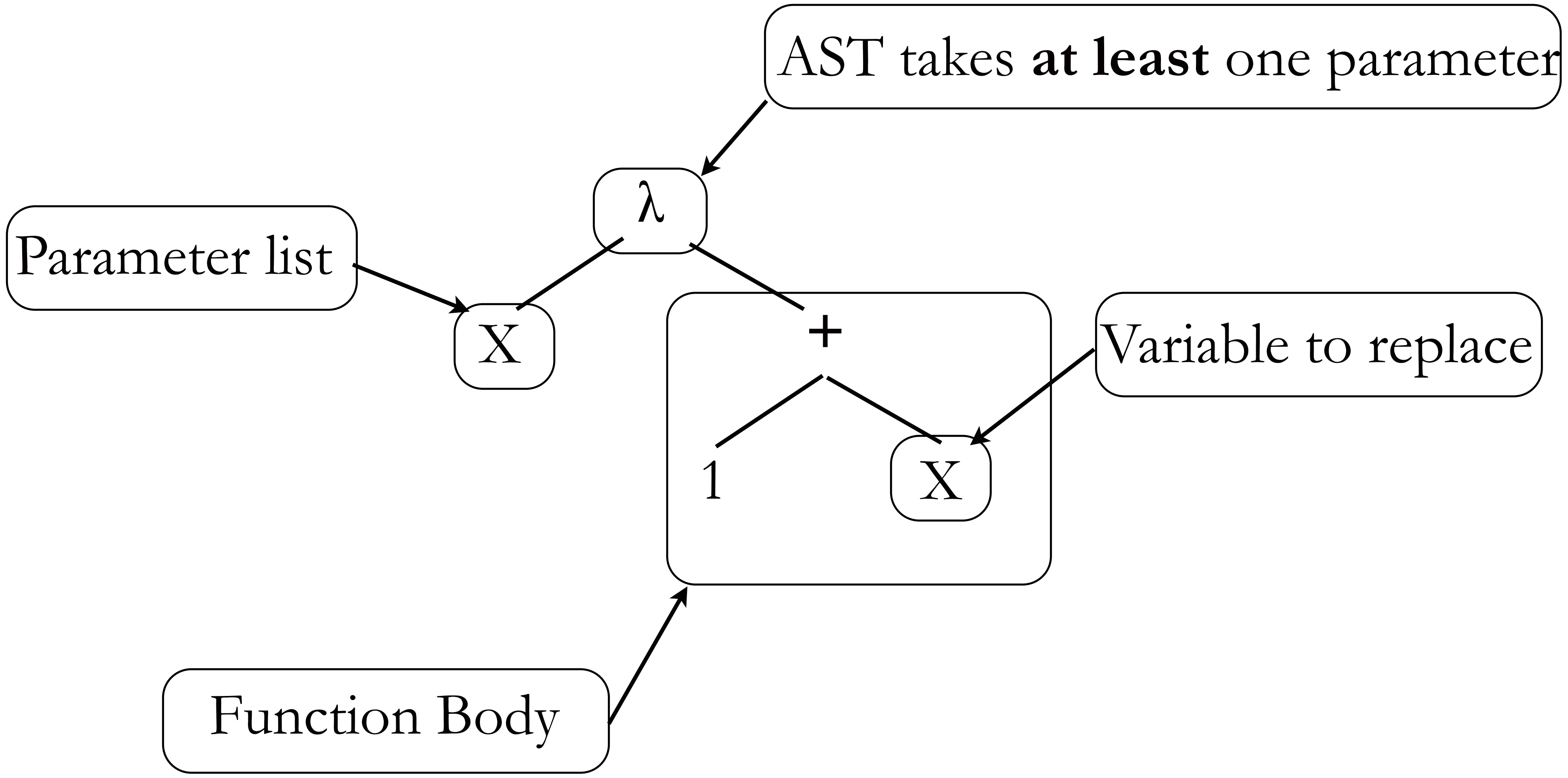
Difference between parameter and argument?
Parameter is a variable name that can take any value
Argument is an actual number/value



λ Calculus
Lambda Calculus

- λ calculus and ASTs

(λ X. + 1 X)



(λ X. + 1 X)

- Needs a value for X to do anything
- Argument names are ALWAYS one letter in λ calculus
- In Racket?
  - (lambda (x) (+ 1 x))
- Function
  - "Method" in Java
  - Usually takes one or more arguments
- Two different kinds of expressions:
  - "Reducible expression" (redex)
    - can be made simpler
  - e.g. (+ 2 1)=>3

- Two different kinds of expressions:
  - "Reducible expression" (redex)
    - can be made simpler
  - e.g. (+ 2 1)=>3
  - Expression or λ calculus
    - can't be simplified (yet)
  - (λx. + 1 x)
  - How to convert an expression into a redex?
  - Give it a value for variables/parameters

((λ x. + 1 x) 5)
Argument gets passed to parameter
Each variable in body corresponding to parameter gets replaced

(+ 1 5)
6

- Redexes
  - ((λ y. * 1 y) 2)
  - (* 1 2) => 2
  - ((λ xy. + x y) 2 3)
  - Take **in order**
    - x=2, y=3
  - ((λ y. + 2 y) 3)
  - (+ 2 3)
  - In Racket
    - ((lambda (x y) (+ x y)) 2 3)

This is a "β (**beta) reduction**"

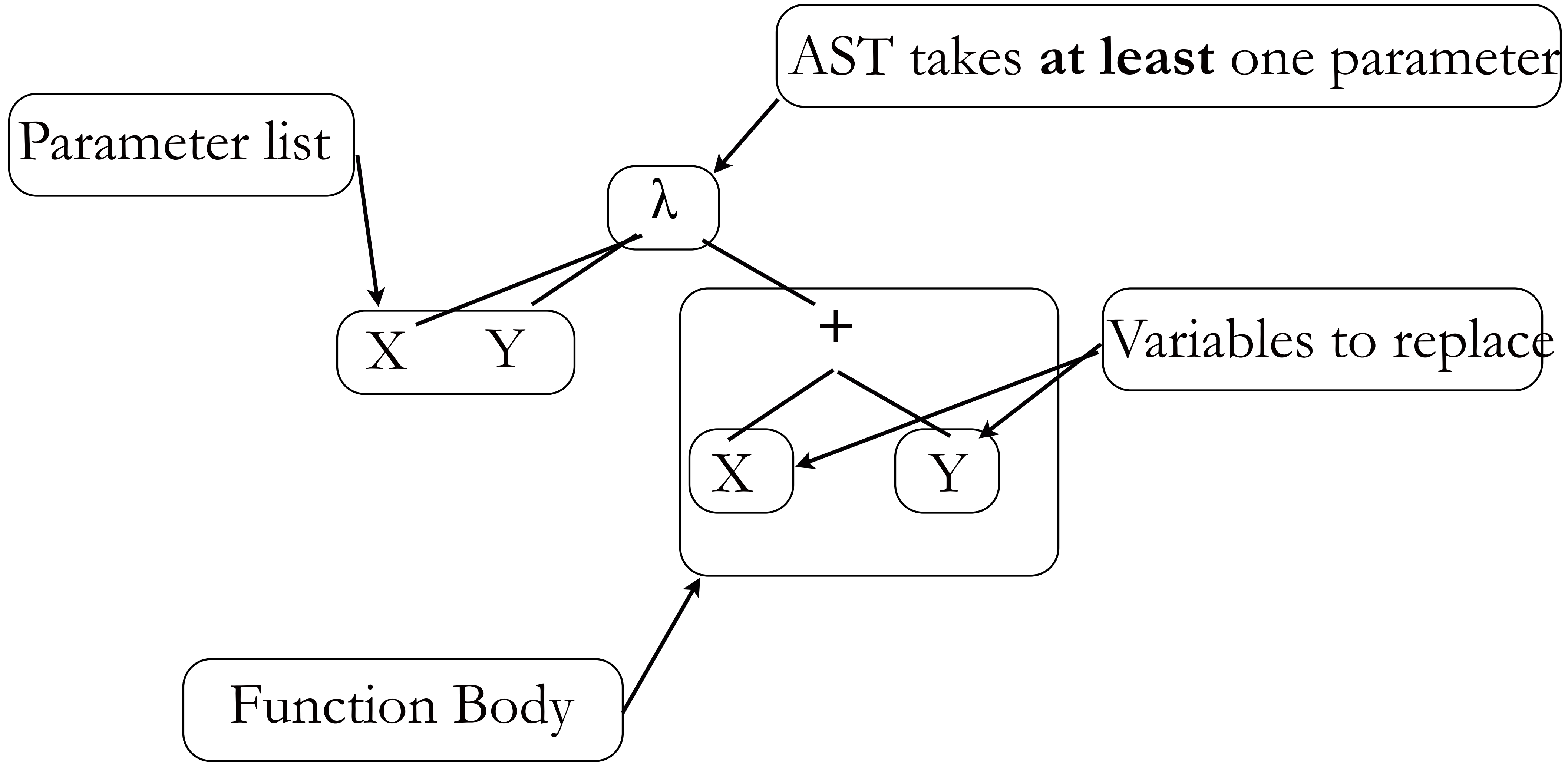
Is ((λxy. * x y)2) a redex?
Simplify to: (λy. * 2 y)

((λx. x) 2)
2

What if the argument is a redex?
((λx. +3 x)(+ 2 2))
Evaluate argument first (usually):
  ((λx. +3 x) 4)
  (+ 3 4) => 7
Bring in "as is"
  (+ 3 (+ 2 2))

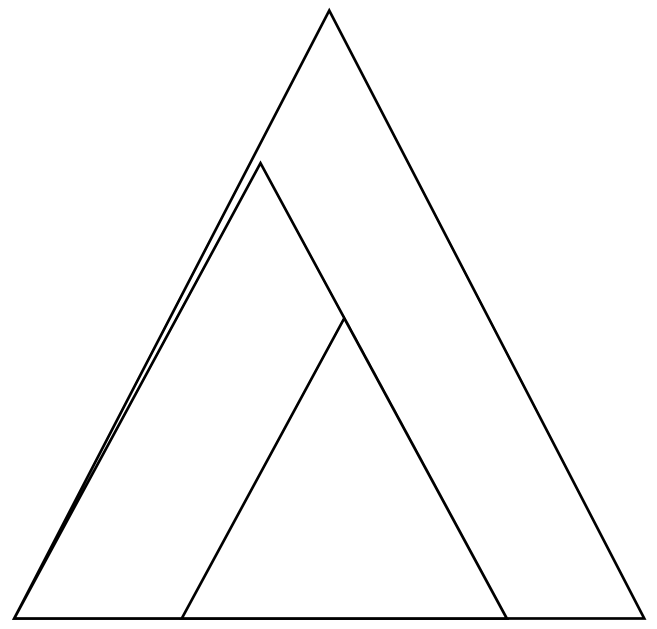
- λ calculus and ASTs

(λ XY. + X Y)



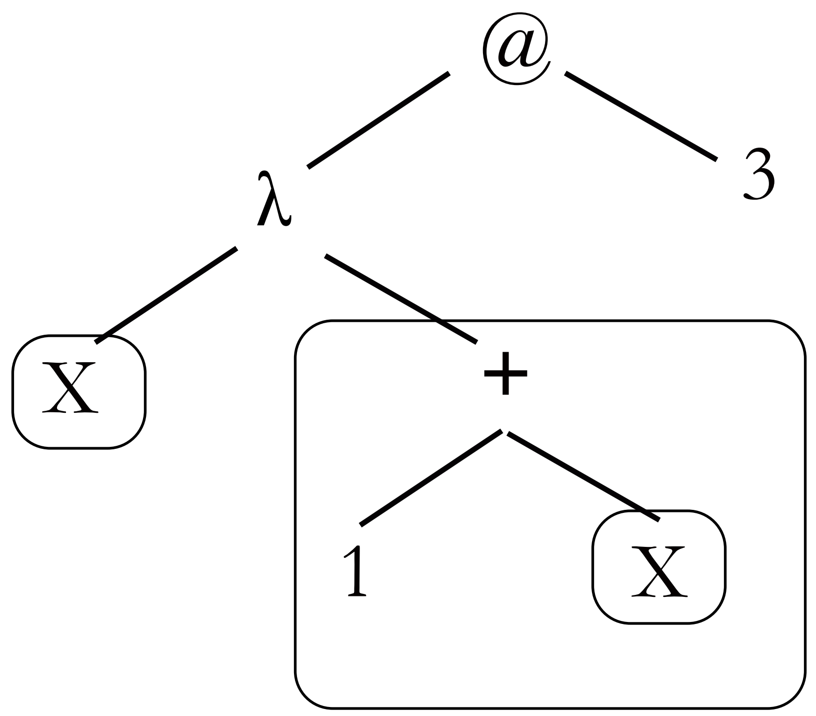
- In general (for the moment…)
  - Evaluate the **innermost** redex first
  - i.e. the most deeply nested and furthest to the **right**
    - (λx. x)($\underline{+\ 1\ 1}$)
  - Implications for ASTs?
    - Need a new node: “application” (@)
    - i.e. apply λ expression to one or more arguments.

Body (simple AST)
λ (parameter list)
@ (argument list)



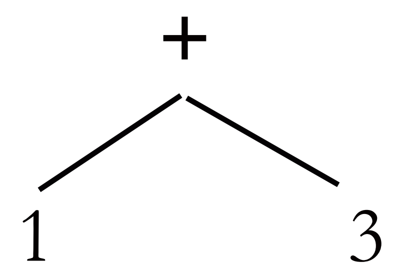
- λ calculus and ASTs

((λ X. + 1 X) 3)



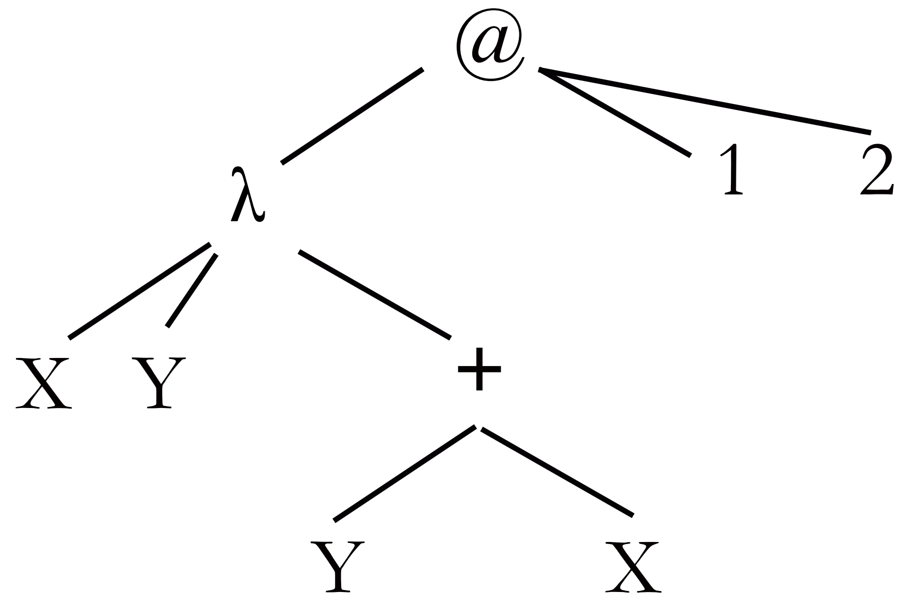
- λ calculus and ASTs

((λ X. + 1 X) 3)



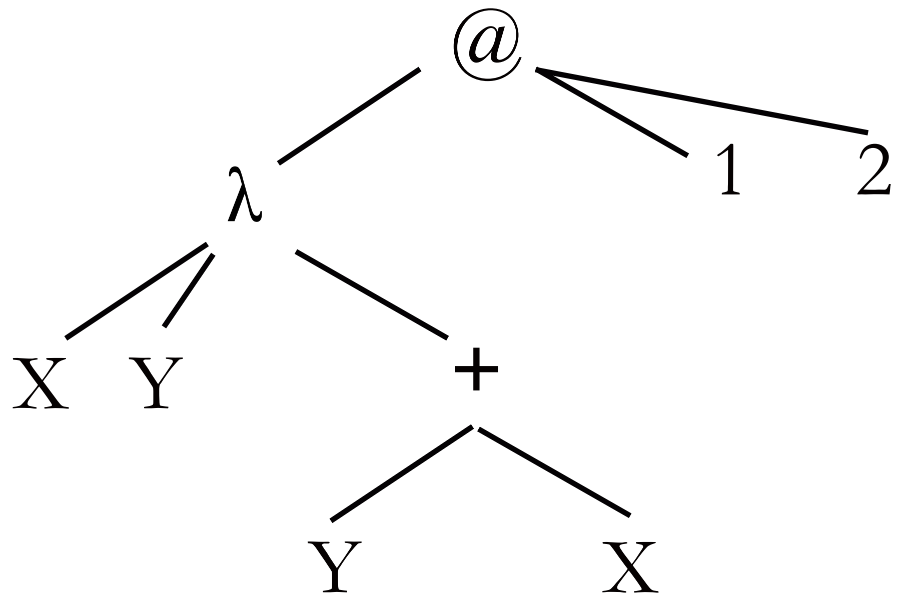
- λ calculus and ASTs

((λ XY. + Y X) 1 2)



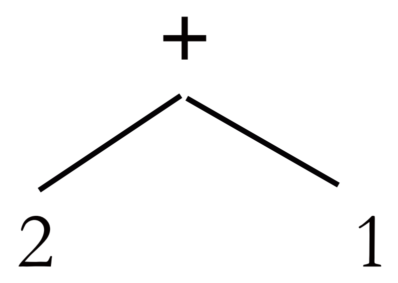
- λ calculus and ASTs

((λ XY. + Y X) 1 2)



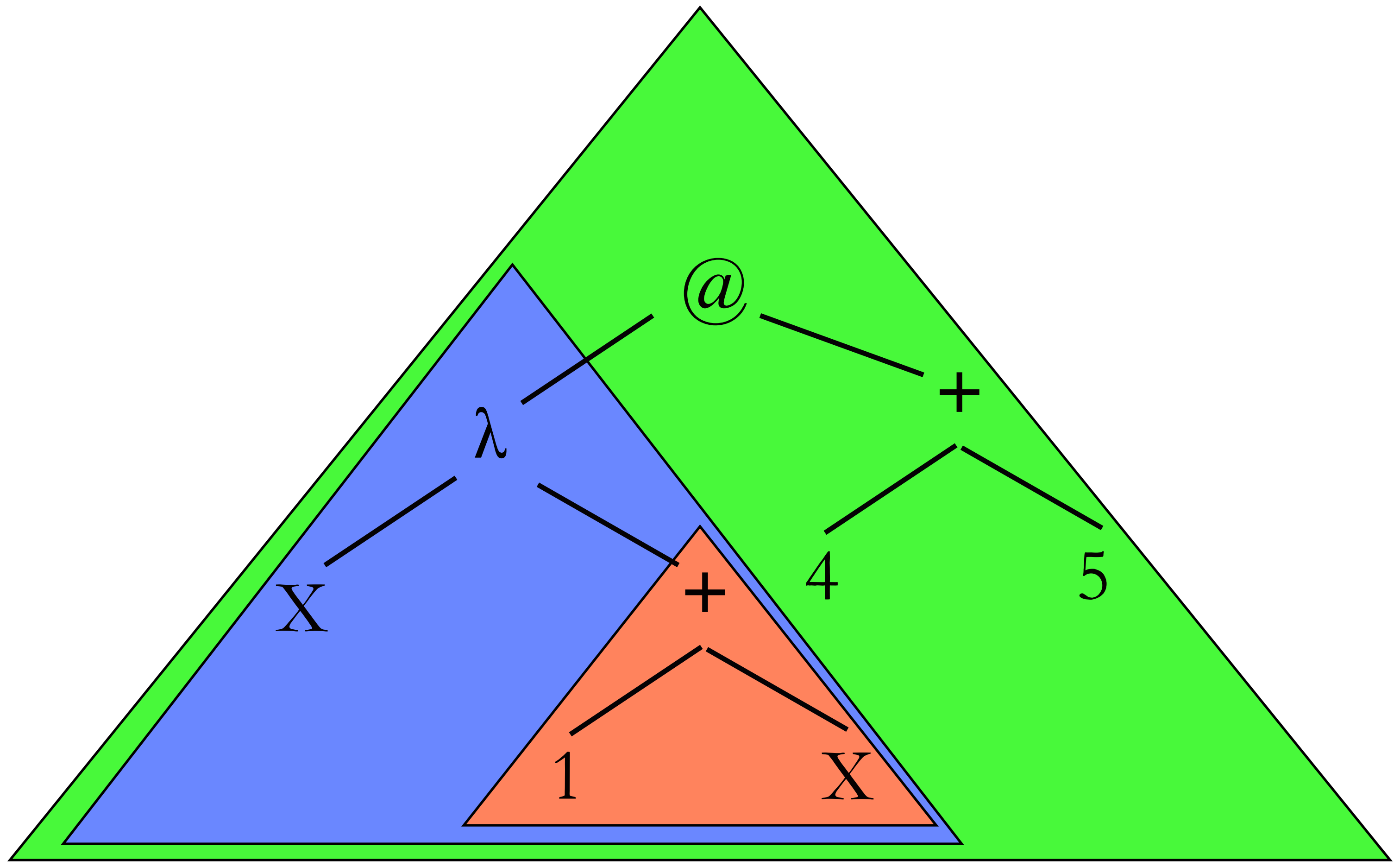
- λ calculus and ASTs

((λ XY. + Y X) 1 2)



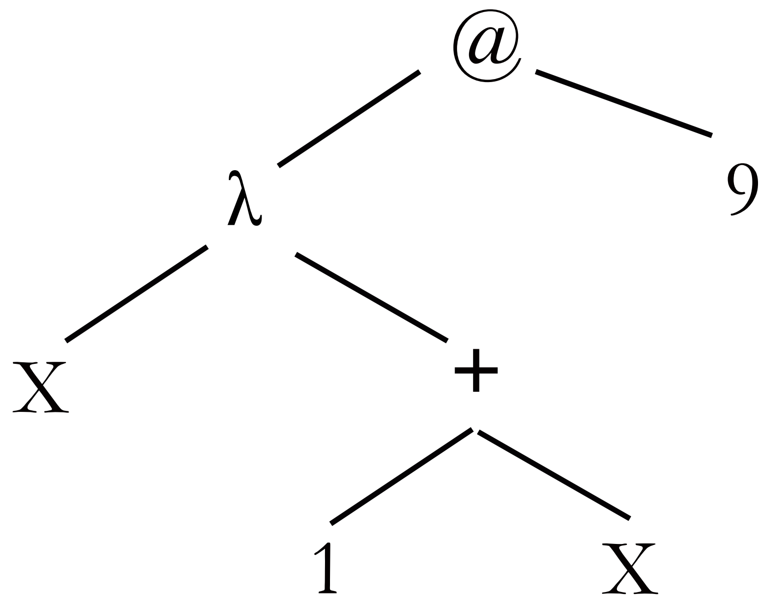
- λ calculus and ASTs

((λ X. + 1 X) (+ 4 5))



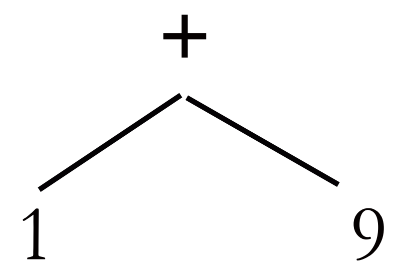
- λ calculus and ASTs

((λ X. + 1 X) (+ 4 5))

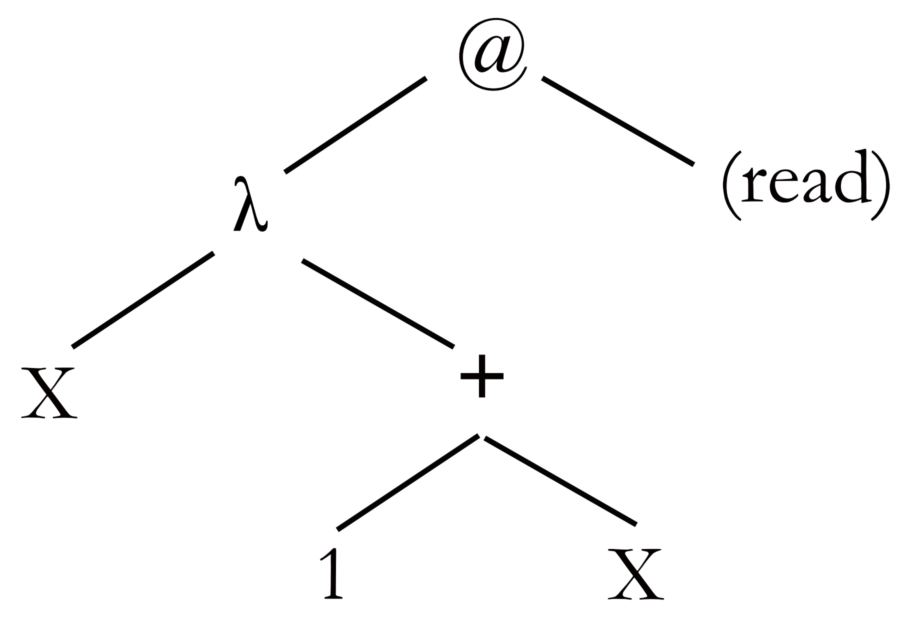


((λ X. + 1 X) (+ 4 5))

- λ calculus and ASTs



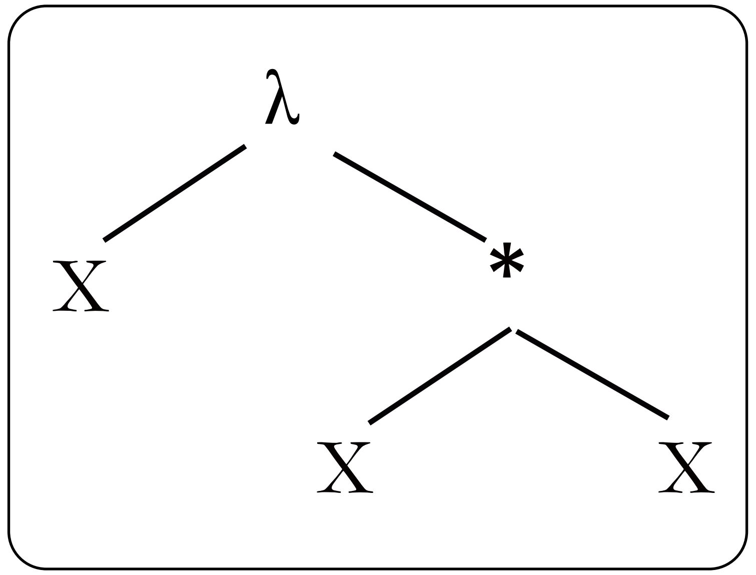
- Racket
  - > (lambda(x) (+ 1 x))
  - #<procedure>
  - Meaning?
    - Expression it can't reduce
  - > ((lambda(x) (+ 1 x)) (<u>read</u>))
  - > ((lambda(x) (+ 1 x)) 5)
  - 6



- Variables and Types
  - Racket uses Type Inference
    - Guesses at what the type should be e.g. expects an integer
    - Why? "+ 1"
      - However, (λ x. x) can take anything
- "Operator overloading"
  - Operator behaves the same regardles of type
    - e.g. (+ x y) adds two things
      - Doesn't matter what type they are.
      - Another view of +:
        - ((λxy. + x y)A B)
        - ((λxy. - x y) A B)

```
(lambda(x) (+ 1 x))
```

- Too long winded to write λ expression each time
  - Not every operator is built in..
  - sqr: (λx. * x x)
  - but, (sqr x) is more useful
- This is called a "function"
  - Difference between this and supplied operators?
  - None.
  - (define sqr (lambda (x) (* x x)))
  - > (sqr 3)
  - (define add (lambda (x y) (+ x y)))
  - > (add 2 1)
  - 3



Variables
Arguments
Parameters
Variables

- > (define x 3)
- > x
- 3
- > (+ x 3)
- 6
- > x
- 3
- > (define x (+ x 3))
- > x
- 6
- (define cube (lambda (x) (* x (sqr x))))
- Strictly speaking, λ calculus doesn't name functions

How does Racket know it's not a fn?
Type inference
No lambda part
Functions are λ expressions with names
λ expressions are expressions with params

	Functional	**Imperative**
# funs	Large	Small
Size funs	Small	Large
Params?	Yes	Yes
Return	Always	Usually

- Which is better?
  - Easier to debug if functions are “tightly coupled”
    - i.e. all instructions in a function are related.
  - Much easier to manipulate functions in functional language….
  - Possible to program imperative programs in a functional style
- Aim of this course?
  - Figure out what problem is
  - Break up problem (into functions)
- Code up problem (in either Racket or Java)

- Scope
  - where something takes effect
  - scope depends on length of line
  - variables have scope
  - (λ x. x)
  - (λ x. y)
  - > (define x 2)
  - > (define add1 (lambda (y) (+ 1 y)))
  - > x
    - 2
  - > y
    - reference to undefined identifier: y

$\sqrt{4} + 5$	7
$\sqrt{4 + 5}$	3

```
> (add1 x)
4
> x
3
```

add1 returns value of 1 + arg
Doesn't modify argument.

```
>y
Error: undefined variable
```

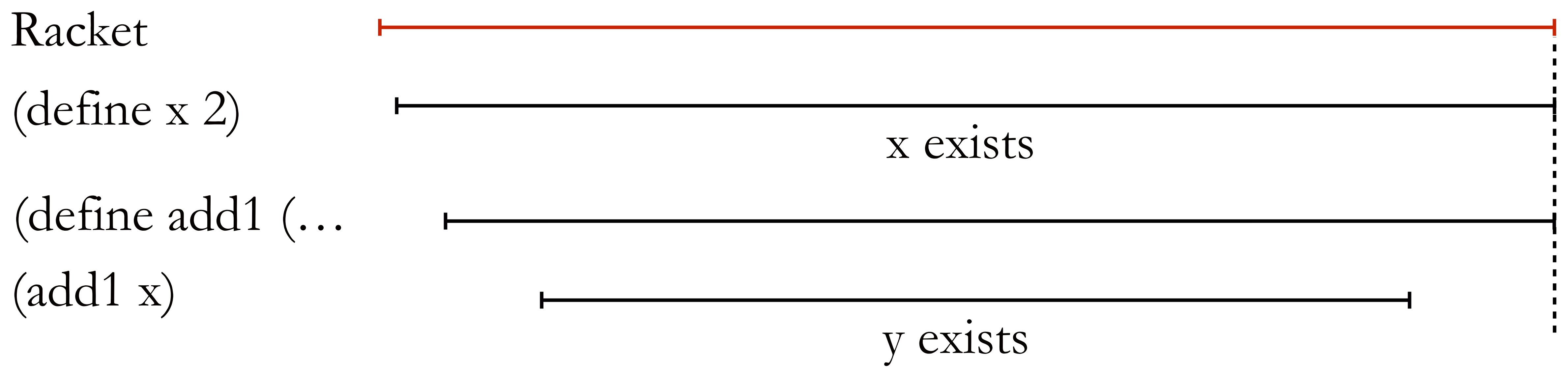
y only exists while fun is running, disappears after that

y only exists in add1 function

- Details
  - (define add1 (lambda (y) (+ 1 y)))
  - (add1 x)
  - add1 is called
  - variable y is created
  - argument (contents of x) copied in
  - body executed
  - y is thrown away
  - Limited “lifetime”
  - Why does x still exist?
  - Defined in its own right

Global variables
    Can be seen anywhere
Local variables
    Only seen in defining function

- Lifetime diagrams



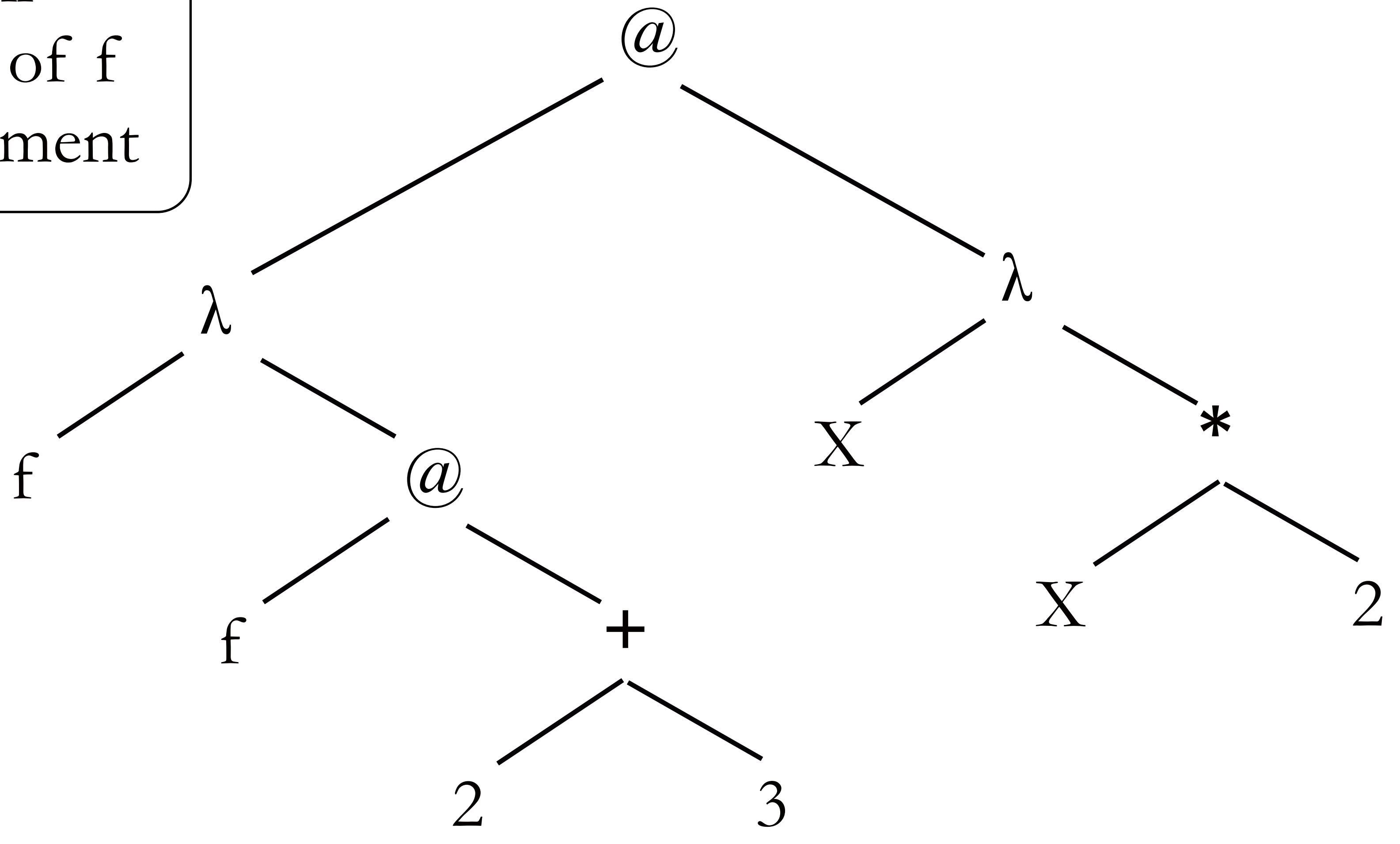
X exists as long as Racket is running
add1 exists as long as function takes
y exists (almost) as long as add1

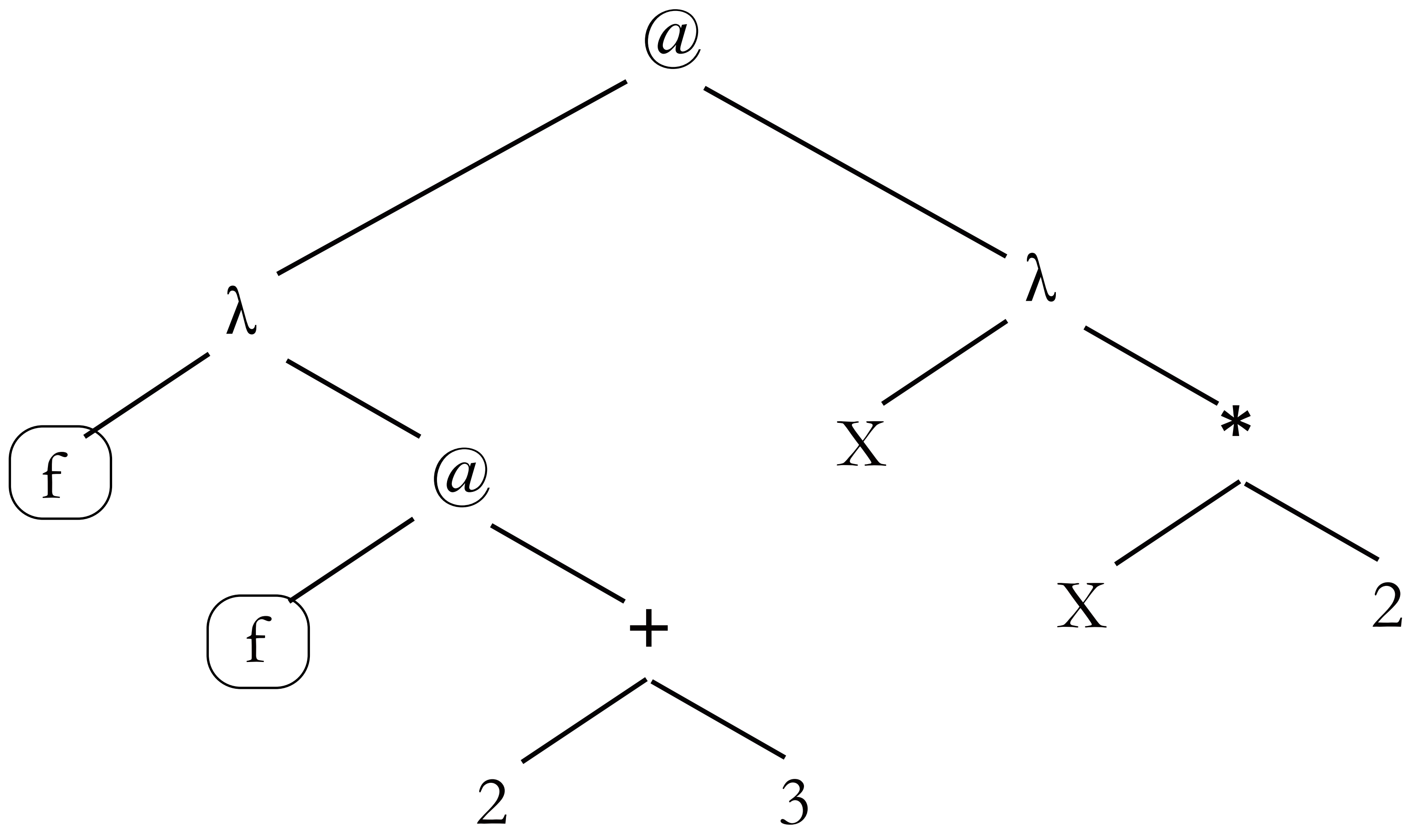
- (λ x. * x 2)
- ((λ x. * x 2) 3)
- (* 3 2)

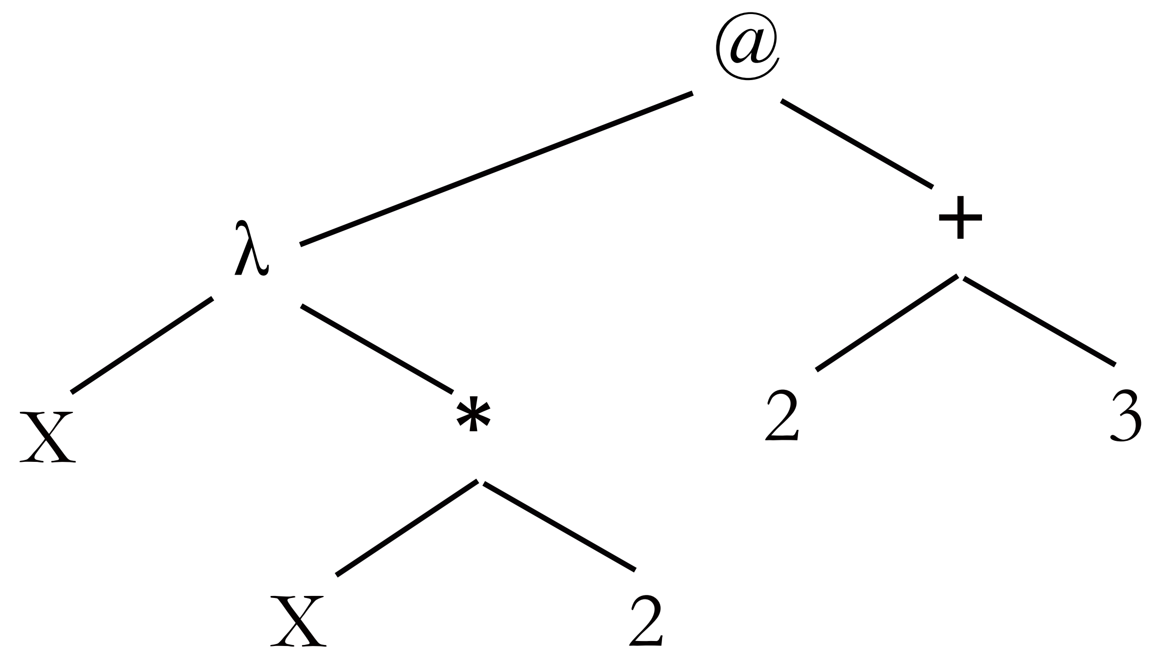
(λ f. f $\underline{\text{(+ 2 3)}}$)

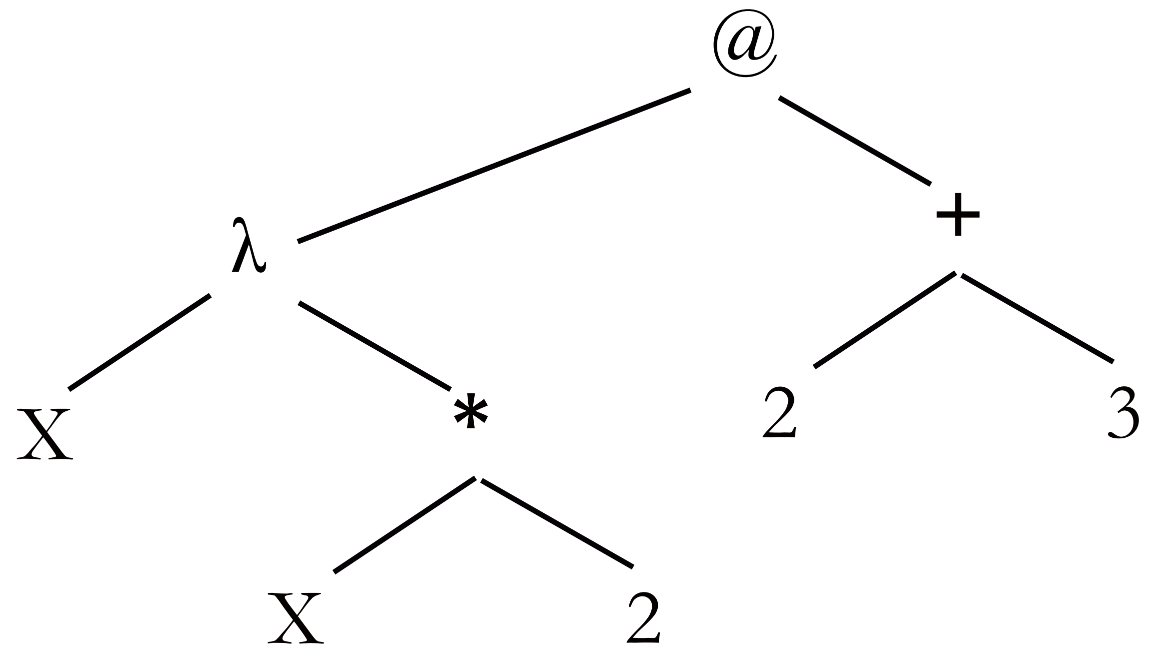
((λ f. f (+ 2 3))(λ x. * x 2))

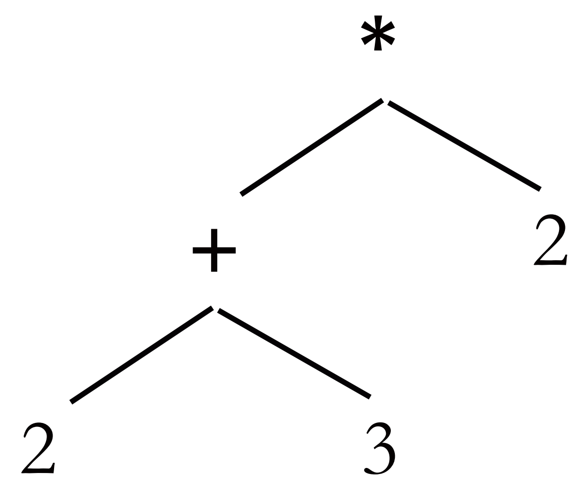
Replace all instances of f with argument




@
λ
f
@
f
+
2
3
λ
X
*
X
2


@
λ
+
X
*
2
3
X
2


@
λ
+
X
*
2
3
X
2


*
+
2
2
3