Lambda Calculus (λ Calculus)

October 18, 2018

Smallest Universal Programming Language

FuncLang

Name, Identifier

- $e ::= x | \lambda x.e | e_0 e_1$ application

Func

<expression>	:=	<name> | <function> | <application>
<function>	:=	λ <name>.<expression>
<application>	:=	<expression><expression>

(lambda (x) (+ x 1)) 2

Exp Exp

theoretical

As a programming language, sometimes a concrete implementation of lambda calculus also supports predefined constants such as '0' '1' and predefined functions such as '+' '*'; we add parenthesis.

λx . expression

expression expression

Examples

$\lambda y.y$

- $\lambda x.x$ (lambda abstraction: building new function)
- $(\lambda x.x)y$ (application)

$(\lambda x.x)\ y$

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

What is λ Calculus and Why It Is Important?

1. A mathematical language; A formal computation model for functional programming; a theoretical foundation for the family of functional programming languages.
2. Study interactions between functional abstraction and function applications; study some mathematical properties of effectively computable functions
3. By Alonzo Church in the 1930s
4. In 1920s - 1930s, the mathematicians came up different systems for capturing the general idea of computation:
   - Turing machines – Turing
   - m-recursive functions – Gdel
   - rewrite systems – Post
   - the lambda calculus – Church
   - combinatory logic – Schnfinkel, Curry

   These systems are all computationally equivalent in the sense that each could encode and simulate the others.

The Mathematical Precursor to Scheme

Mathematical formalism to express computation using functions:

- Everything is a function. There are no other primitive types—no integers, strings, cons objects, Booleans ... If you want these things, you must encode them using functions.
- No state or side effects. It is purely functional. Thus we can think exclusively in terms of the substitution model.
- The order of evaluation is irrelevant.
- Only unary (one-argument) functions. No thunks or functions of more than argument.

Implementation in Scheme/DrRacket

- Syntax implemented in Scheme:

$$
\begin{array}{rcll}
e & \rightarrow & x & \text{Variable} \\
 & | & (\lambda\ (x)\ e) & \text{a lambda expression} \\
 & | & (e\ e) & \text{Application}
\end{array}
$$

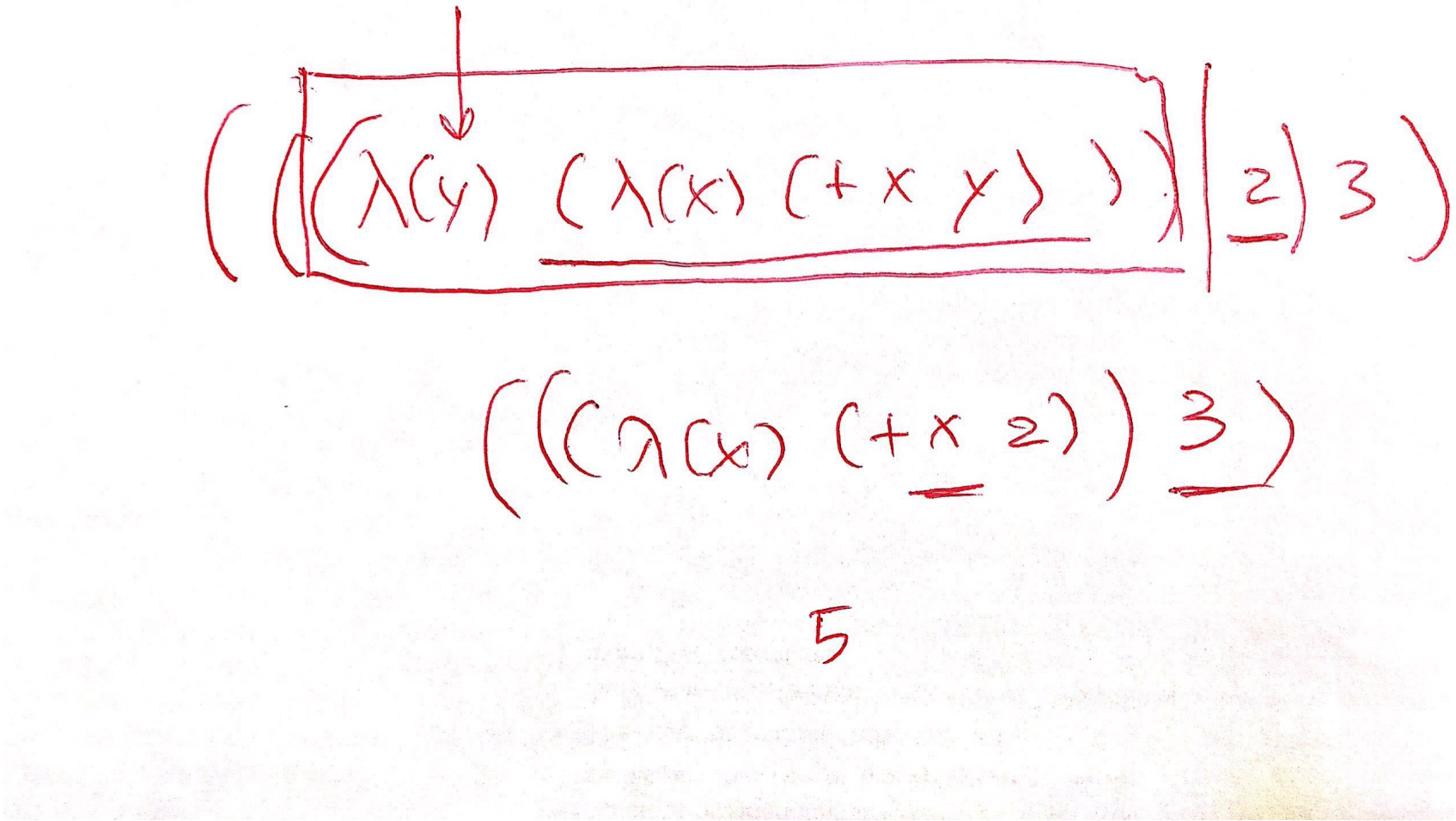
$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$

Compute $.(2)$ where $.(x) = x + 1$

(lambda (x) e)

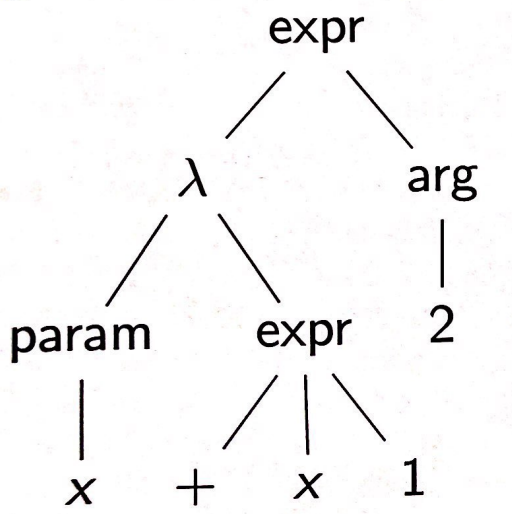
slightly different syntax

Example

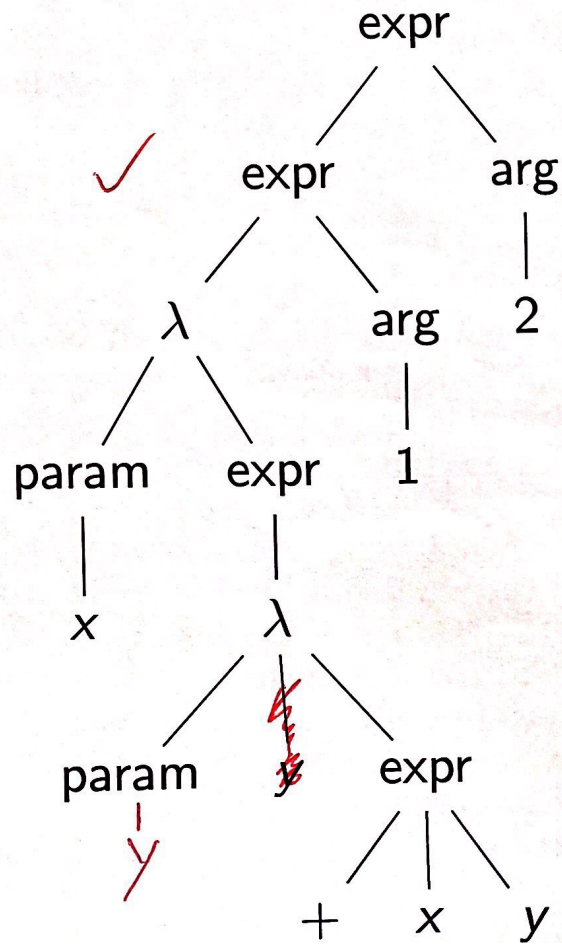


The AST View (Simplified)

$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$



$(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)\ 2)$



Another view: Imperative

λ-expression	Function Definition	Invocation

$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$	.(x) {
	  x + 1
	}
		.(2) = 2 + 1

$(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)\ 2)$	.(x) {
	  ..(y) {
	    x + y
	  }
	}
		.(1) = ..(y) {
		    1 + y
		  }
		..(2) = 1 + 2

What have we seen so far...

all the operations in language are supported

Anonymous function, functions as first-class elements, inner functions, formal parameters, actual arguments.

functions can be invoked whenever

Results are procedures

Bound and Free Variable: Imperative View

Bound Variable

A bound variable is one which appears in an expression after it has appeared in a λ.

λ-expression	Function Definition	Bound Variables

$(_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x$	`.(x) {`
	`  x + 1`
	`}`
		x

$(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x$	`.(x) {`
	`  ..(y) {`
	`  x + y`
	`  }`
	`}`
		x, y

$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y$	`..(y) {`
	`  x + y`
	`}`
		y

Free Variables

Any variable that is not bound is free.

How about

$$(((\,(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (_x\ \lambda\ (x)\ (+\ x\ y)\ )_x\ )_y\ )_x\ 1)\ 2)\ 3)\ )$$

```
.(x) {
  ..(y) {
    ...(x) {
       x + y
    }
  }
}
```

```
.(1) = ..(y) {
           ...(x) {
              x + y
           }
       }
```

Resolving Name Capture

α-Conversion

Rename variables

$$((((_x \lambda (x) (_y \lambda (y) (_x \lambda (x) (+ x y))_x)_y)_x 1) 2) 3))$$

$$((((_x \lambda (x) (_y \lambda (y) (_z \lambda (z) (+ z y))_z)_y)_x 1) 2) 3))$$

Currying

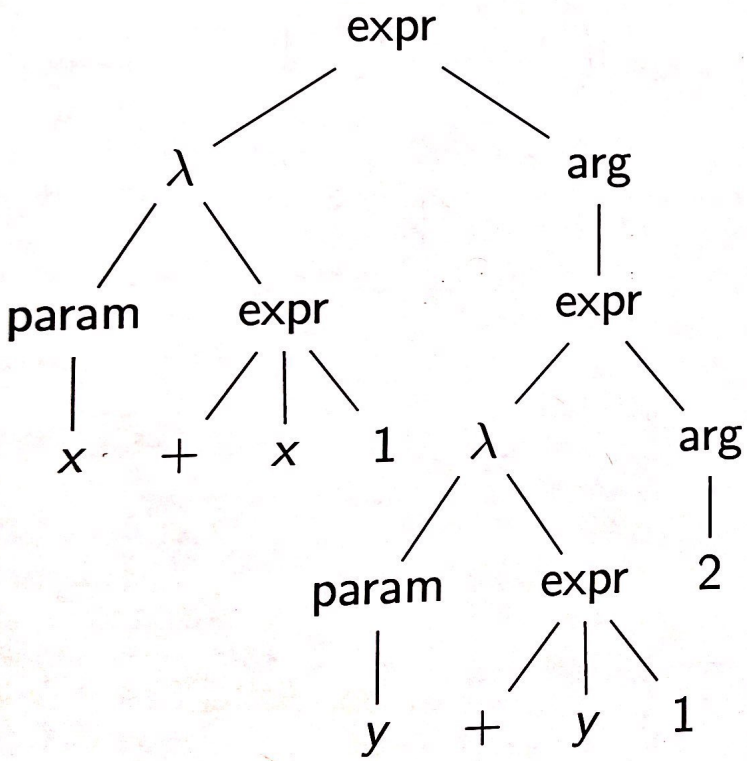
Pure lambda calculus pairs one variable with one λ

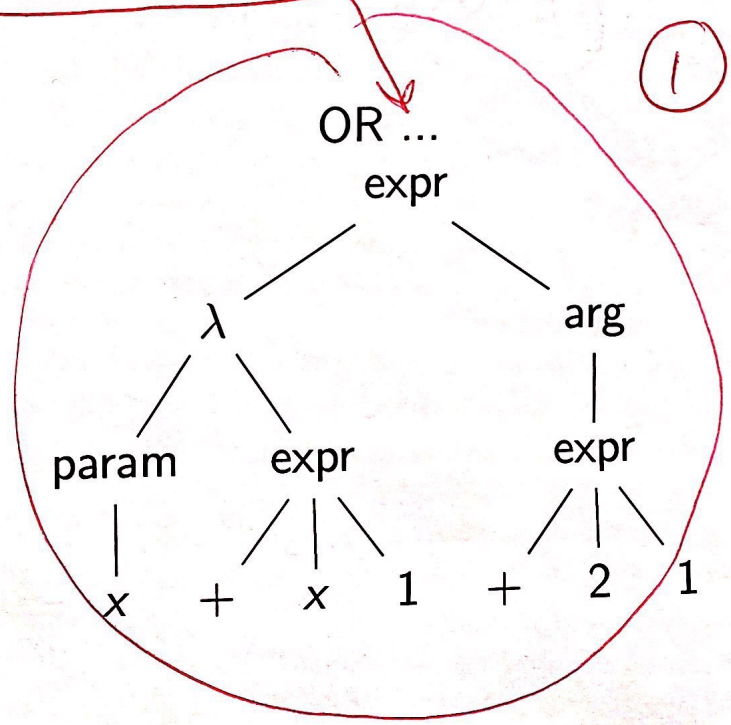
- Functions with many parameters
  $(_{x\ y}\ \lambda\ (x\ y)\ e\ )_{x\ y}$: two formal parameters x and y
- Semantically equivalent expression: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ e\ )_y\ )_x$

Concept introduced by Haskell Curry.
More on this in subsequent lectures.

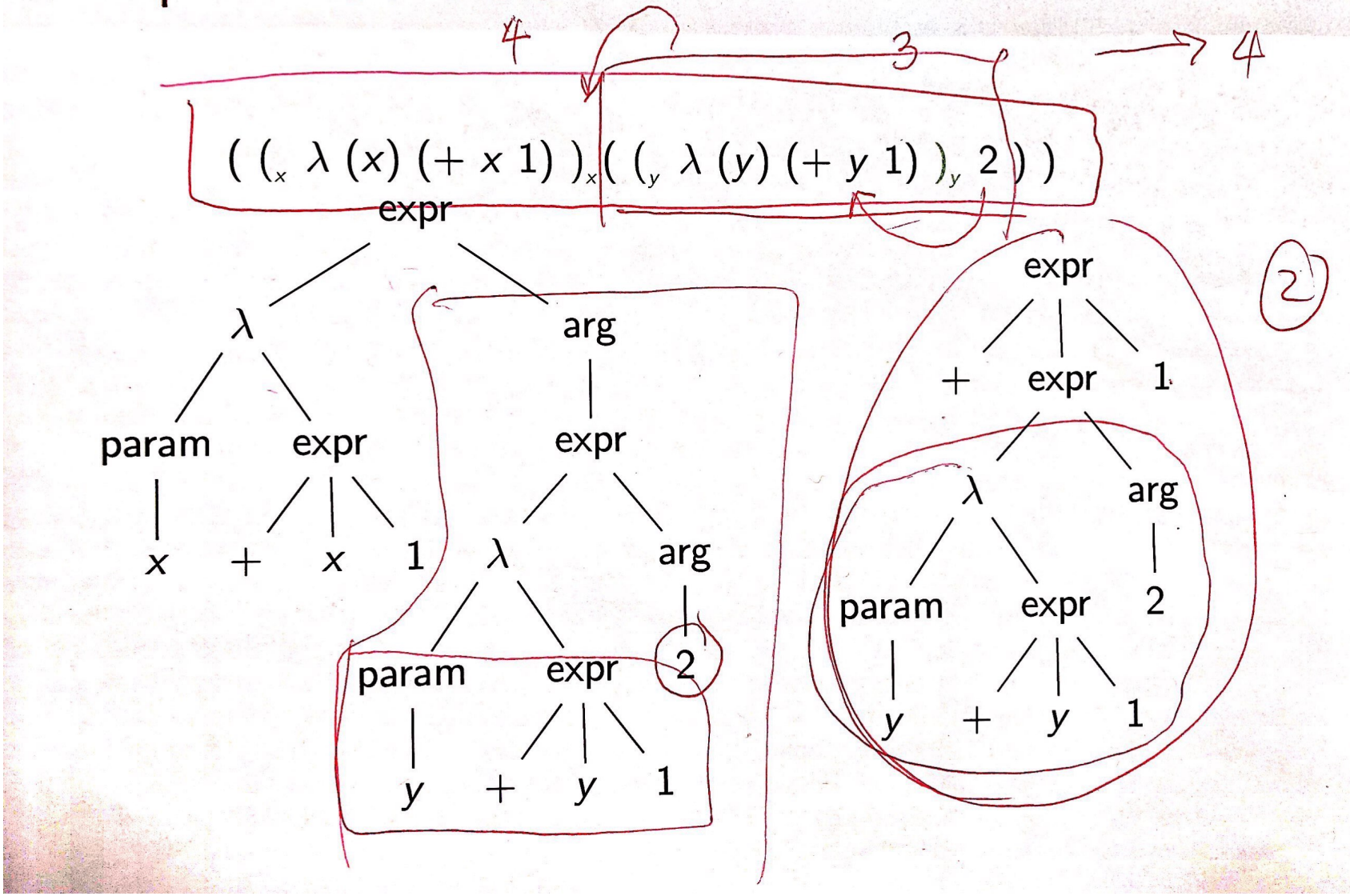
Examples: AST View

$$((_x \ \lambda \ (x) \ (+ \ x \ 1) \ )_x (\ (_y \ \lambda \ (y) \ (+ \ y \ 1) \ )_y \ 2 \ ))$$

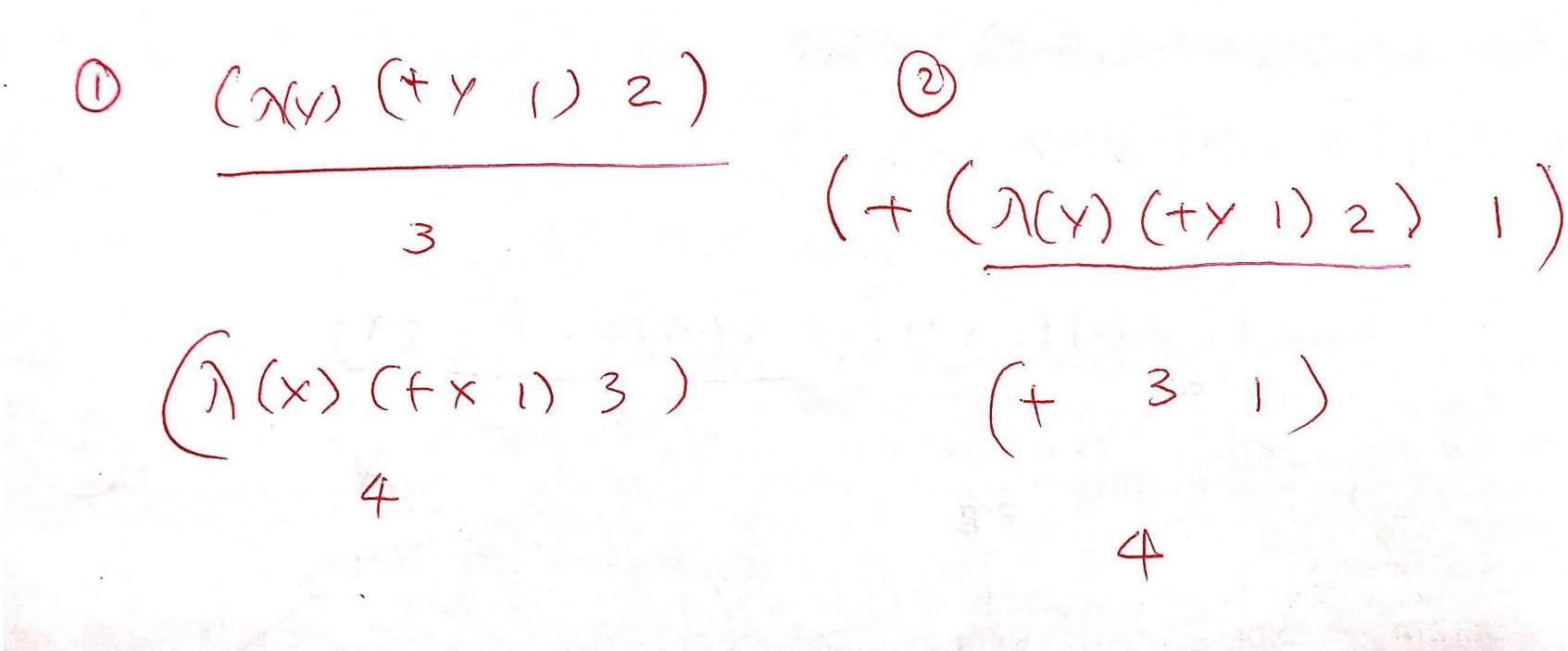




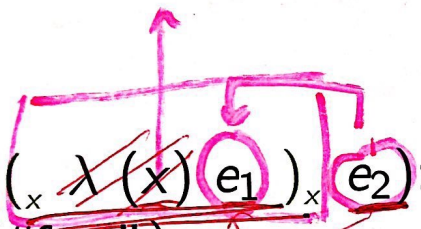
Examples: AST View



Example



Formal Semantics of the Language



- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$ (β-reduction)

$$((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)$$

$$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y[x \mapsto 1]$$

$$(_y\ \lambda\ (y)\ (+\ 1\ y)\ )_y$$

Ordering in Evaluation

$$(\ (_x \ \lambda \ (x) \ (+ \ x \ 1) \ )_x (\ (_y \ \lambda \ (y) \ (+ \ y \ 1) \ )_y \ 2 \ ) \ )$$

After β-reduction

Either $(+ \ (\ (_y \ \lambda \ (y) \ (+ \ y \ 1) \ )_y \ 2 \ ) \ 1)$

Or $(\ (_x \ \lambda \ (x) \ (+ \ x \ 1) \ )_x (+ \ 2 \ 1) \ )$

Recap

- $(_x\ \lambda\ (x)\ e\ )_x$: a lambda expression representing definition of function
- $(\ (_x\ \lambda\ (x)\ e\ )_x\ p)$: a lambda expression representing application of a function.
  - Formal parameter: x
  - Actual argument: p
  - Computation: $e[x \mapsto p]$, replace free occurrences of x in e with p (β-reduction)
- Order of β-reduction does not impact the result if each β-reduction terminates

  $(\ (_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x(\ (_y\ \lambda\ (y)\ (+\ y\ 1)\ )_y\ 2\ )\ )$

Syntax Revisited

$$
\begin{array}{rcll}
e & \rightarrow & x & \text{Variable} \\
 & | & (_x\ \lambda\ (x)\ e\ )_x & \text{a lambda expression} \\
 & | & (e\ e) & \text{Application}
\end{array}
$$

What about the data? Boolean, Integers, ...

Examples

What is the result of

$$((_x\ \lambda\ (x)\ x\ )_x\ (_y\ \lambda\ (y)\ y\ )_y\ )$$

y

0

$(\lambda(y)\ y)$

$((\lambda(y)\ y)\ (\lambda(z)\ z))$

Examples

What is the result of

$$((_x \ \lambda\ (x)\ x\ )_x \ (_y \ \lambda\ (y)\ y\ )_y \ )$$

- $(_x \ \lambda\ (x)\ x\ )_x$: identify function. The function applied to any entity returns the entity itself.

Adding and subtracting 0 from an arith. expression returns the expression
Multiplying and dividing by 1

Examples

- $(_x\ \lambda\ (x)\ (x\ x)\ )_x$: self application function. The function when applied to an entity, applies the entity to itself.
  What is the result of $((_x\ \lambda\ (x)\ (x\ x)\ )_x\ 3)$?

  What is the result of $(\ (_x\ \lambda\ (x)\ (x\ x)\ )_x\ (_y\ \lambda\ (y)\ y\ )_y\ )$

Examples: Function Application

Expression → x |
λx . expression |
expression expression

body

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$: Application of function f on x.

What about

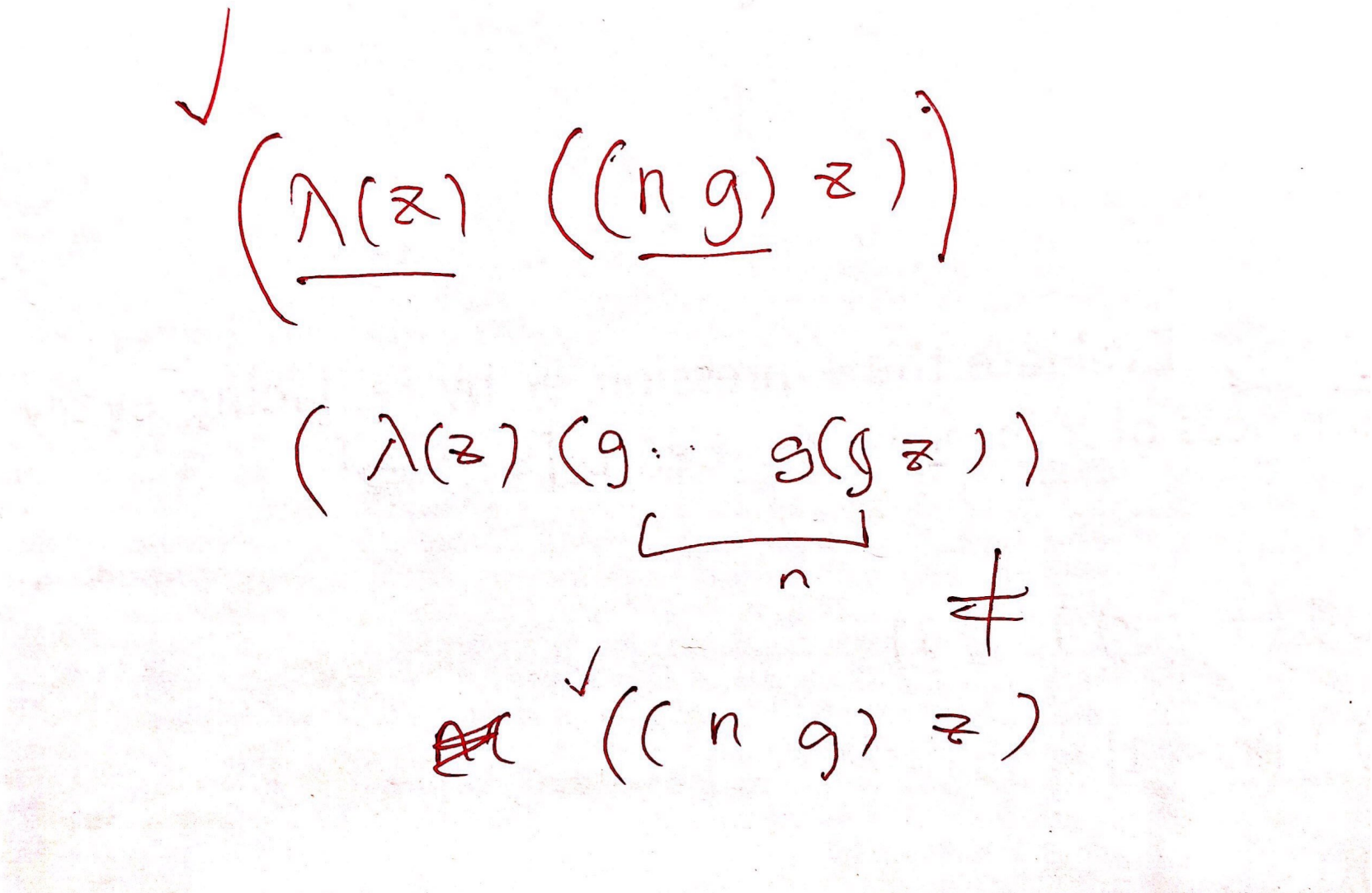
actual

$(\ ((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f\ (_y\ \lambda\ (y)\ y\ )_y)\ (_z\ \lambda\ (z)\ (z\ z)\ )_z\ )$?

let $g = (_y\ \lambda\ (y)\ y\ )_y$ and $v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$. Then,

result is $(g\ v) = (\ (_y\ \lambda\ (y)\ y\ )_y\ v) = v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$

Examples for Function and Application



Encoding Natural Numbers

A natural number is represented by the number of application of some function on some entity.

A natural number function takes two arguments (function and entity on which the function is to be applied).

Church Encoding

Representing data and operations using functions

- non-negative integers, 0, 1, 2 ...
- booleans, true, false, and, or, not, ite
- pairs
- Rational numbers may be encoded as a pair of signed numbers.
- Real numbers may be encoded by a limiting process that guarantees that the difference from the real value differs by a number which may be made as small as we need

SUCC

ADD

Natural Numbers (Church Numerals)

Encoding of numbers: 0, 1, 2, . . . , as functions such that their semantics follows the natural number semantics.

Intuition: The number n means how many times one can do certain operation.

Encoding Natural Numbers

zero $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$

one $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

two $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f$

n $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ \ldots(f\ x)\ldots))\ )_x\ )_f$

Assume f is operation and x is the object on which the operation is done.

Encoding Natural Numbers

zero $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$

one $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

two $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f$

n $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ \ldots(f\ x)\ldots))\ )_x\ )_f$

Assume f is operation and x is the object on which the operation is done.

E.g.: f is adding '1' to the list

E.g.: x is an empty list

Then,
meaning of zero is empty list ( )
meaning of one is (1)
meaning of two is (11)
meaning of three is (111)

Example

What is the semantics of

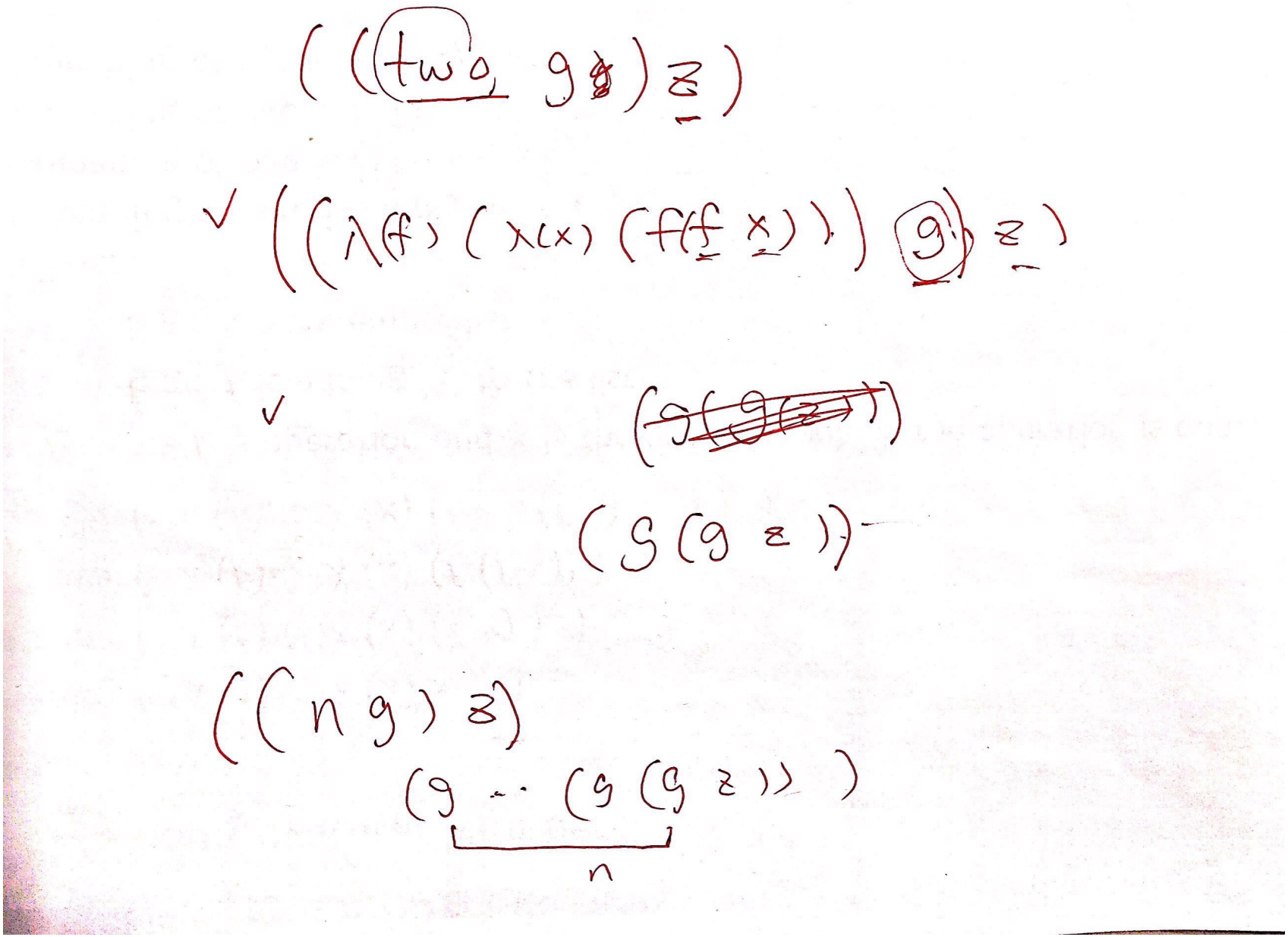
- $((two\ g)\ z)$: two applications of g on z.

$$(((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f\ g)\ z) = (g\ (g\ z))$$

- $((n\ g)\ z)$: n applications of g on z, where n is a natural number.
- $(_z\ \lambda\ (z)\ ((n\ g)\ z)\ )_z$: n applications of g on the formal parameter z, where n is a natural number. This result is a function (z if the formal parameter of the function).
- $(\ (_z\ \lambda\ (z)\ ((three\ f)\ z)\ )_z\ two)$: ? Homework.

((f(ff z)) two f f f two.
tw.

Example



Encoding Natural Number

succ zero = one

succ one = two

- successor function: succinct representation of any number
- addition
- multiplication
- subtraction

> self study if interested

succ zero $(\lambda f)(\lambda x\ x)$ =

one $(\lambda f)(\lambda x\ (f\ x))$

Natural Numbers Encoding in λ-calculus.

(P1)

Representation of zero $(_f \lambda(f)\ (_x \lambda(x)\ x)_x)_f$.

one $(_f \lambda(f)\ (_x \lambda(x)\ (f\ x))_x)_f$.

(* Review lecture slides).

Each numbers is represented by the number of time some function is applied to some entity.

* For instance:

(1)

$((\text{two}\ g)\ z)$

$= ((_f \lambda(f)\ (_x \lambda(x)\ (f\ (f\ x))_x)_f\ g)\ z)$

$= (g\ (g\ z))$ after two β-reductions.

* What is the meaning of (two g).

$(\text{two}\ g)$

$= (\ (_f \lambda(f)\ (_x \lambda(x)\ (f\ (f\ x))_x)_f\ g)$

$= (_x \lambda(x)\ (g\ (g\ x))_x)$

= function which takes a parameter x and returns the result of applying the function g twice on x.

* We want to develop a successor function succ which takes a natural numbers as an argument and returns the next natural numbers.

Example: (succ zero) should return $(_f \lambda(f)\ (_x \lambda(x)\ (f\ x))_x)_f$

= one

(succ one) should return $(_f \lambda(f)\ (_x \lambda(x)\ (f\ (f\ x)))_x)_f$

= two

$((\text{two}\ g)\ z)$

$\lambda(n)\,(\lambda(f)\,(\lambda(x)\,(f\,((n\,f)\,x))))$ zero

$(\lambda(f)\,(\lambda(x)\,(f\,((\text{zero}\ f)\,x))))$

$(\text{zero}\ f)$ $(\lambda(f)\,(\lambda(x)\,x))=$

$\lambda(y)$

(p2) As succ - function requires one argument and returns a natural number representation which takes two arguments, its form should be

$(_n \lambda(n)\ (_f \lambda(f)\ (\lambda(x)\ \underline{\qquad}\ ))))_{x\ f\ n.}$

n

n+1

This will be the definition

Now, lets think about definition

As 'n' the natural number is n-application of a function.

Its successor must be (n+1)- application of the same function.

This can be represented as

$(f\ ((n\ f)\ x))$ $\qquad$ $f \ldots f\ x$ (f repeated $n+1$ times)

Why is that ?

$((n\ f)\ x)$: n application of f on x.
(Look at Equation (1))

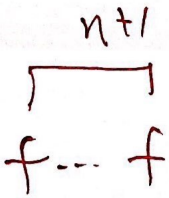
$\therefore (f\ ((n\ f)\ x))$: n+1 application of f on x.

Therefore,
succ function is

(2) $(_n \lambda(n)\ (_f \lambda(f)\ (\lambda(x)\ (f\ ((n\ f)\ x)))))_{x\ f\ n.}$

(* Do it yourself (succ (succ zero)) *)

Successor Function



succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$ zero

(succ zero)

$= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$ zero)

$= (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((\text{zero}\ f)\ x))\ )_x\ )_f$

$(\text{zero}\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$

Therefore,

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((\text{zero}\ f)\ x))\ )_x\ )_f =$

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((_y\ \lambda\ (y)\ y\ )_y\ x))\ )_x\ )_f =$

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f =$

one

(p3) Addition operation/function takes two natural numbers as arguments and produces a new natural number whose semantics/representation is the sum of two inputs.

As add requires two arguments and returns a natural number representation which takes two other arguments, the form of add function is

$(_m \lambda(m)\,(_n \lambda(n)\,(_f \lambda(f)\,(_x \lambda(x)$ ____ $)))))_{x\,f\,n\,m}$

This will be the definition

The definition is constructed as follows

Step 1 ((m succ) n) generates

(succ (succ (succ ... (succ n)) ...))

m-times

(* see Equation ① again)

This produces a natural number equivalent to $m+n$.

Step 2 It should be applied to some function f to represent the natural number

(((m succ) n) f) x)

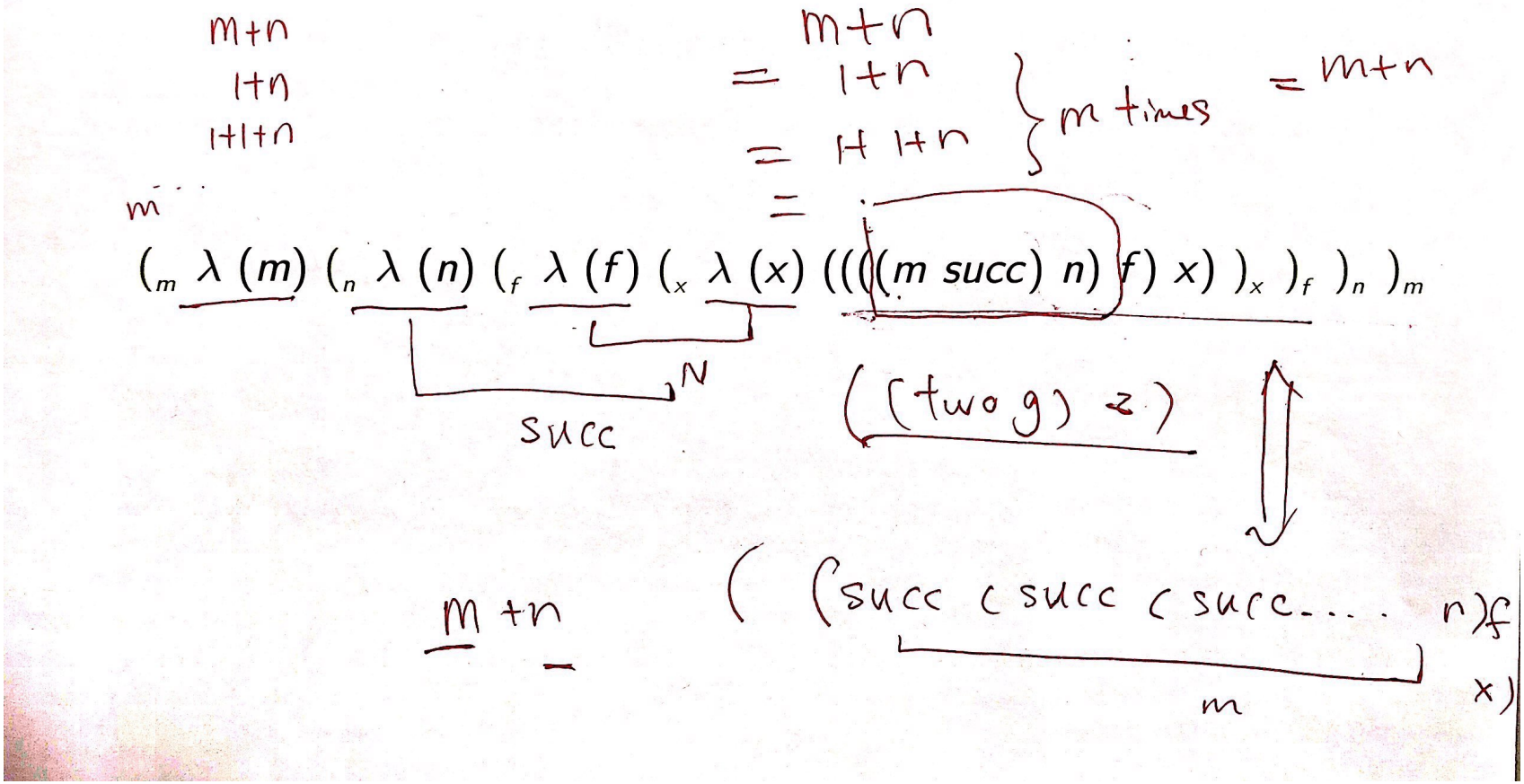
Therefore, add function is

③ $(_m \lambda(m)\,(_n \lambda(n)\,(_f \lambda(f)\,(_x \lambda(x)\,((((m\ succ)\ n)\ f)\ x))))_{x\,f\,n}$

(* Do it yourself ((add (succ zero)) (succ zero)) *)

Optional Page:

More with successors



More with successors

add: $({}_m\ \lambda\ (m)\ ({}_n\ \lambda\ (n)\ ({}_f\ \lambda\ (f)\ ({}_x\ \lambda\ (x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x\ )_f\ )_n\ )_m$

Apply *succ* m times to create a function that is applied n. E.g., $m = 2$ and $n = 3$ results in *succ* of *succ* of 3, which is 5.

Booleans

true: $({}_x\ \lambda\ (x)\ ({}_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $({}_x\ \lambda\ (x)\ ({}_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $({}_c\ \lambda\ (c)\ ({}_t\ \lambda\ (t)\ ({}_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$(((ite\ true)\ s_1)\ s_2) =$

$((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) =$

$(((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ t)\ e)\ )_e\ )_t\ s_1)\ s_2) =$

$((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2) =$

$((true\ s_1)\ s_2) =$

$(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x\ s_1)\ s_2) =$

$((_y\ \lambda\ (y)\ s_1\ )_y\ s_2) = s_1$

$((ite\ false\ s_1)\ s_2) = s_2$

$((ite\ false\ s_1)\ s_2)$

Boolean Operators

not a: if a then false else true. $(((ite\ a)\ false)\ true)$

and a b: if a then b else false. $(((ite\ a)\ b)\ false)$

or a b: if a then true else b $((ite\ a)\ true)\ b)$

What is the adequate set of operators for boolean logic?

Do we program in lambda calculus

No

The objective to learn about Lambda Calculus:

- Better understanding of functional computation and functional programming
- Design of new languages/new features to existing languages

Review and Further Reading

- Concepts: Lambda abstraction and function application, high order functions
- bound and free variables
- currying
- $\beta-$reduction and $\alpha-$conversion
- church encoding

Further reading: pass by name, pass by value, lazy evaluation

- Lambda calculus examples:
  `https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/lambda-calculus-handout.pdf`
- Programming Languages: Lambda Calculus - 1
  `https://www.youtube.com/watch?v=v1IlyzxP6Sg`
- Programming Languages: Lambda Calculus - 2
  `https://www.youtube.com/watch?v=Mg1pxUKeWCk`