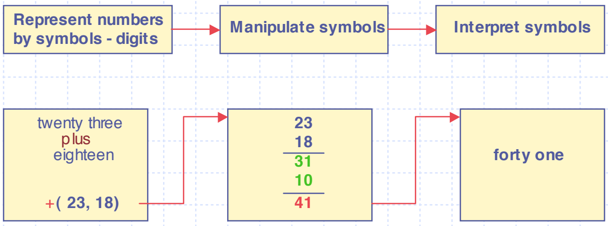
Alonzo Church
1903 - 1995

Model of Computation
based on λ–calculus

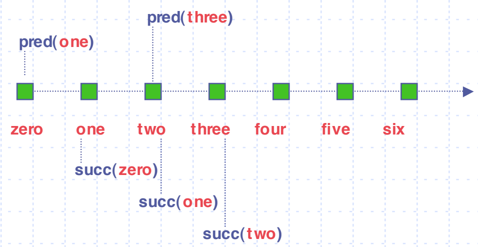


reason

manipulation of symbols can be mechanized - it does not require thinking

NN - natural numbers [0, 1, 2, …)

1. Zero recognition
2. Every ■ has <u>one and only one</u> successor
3. Every ■ except zero has <u>one and only one</u> predecessor



$+ (x, y) ::= x \mid_{y=o}, \quad + (succ (x), pred (y))$

+		
+	5	7
	6	6
	7	5
	8	4
	9	3
	10	2
	11	1
	12	0

$- (x, y) ::= x \mid_{y=o}, \quad - (pred (x), pred (y))$

-	5	2
	4	1
	3	0

$* (x, y) ::= 0 \mid_{y=o}, \quad + (x, * (x, pred (y)))$

$< (x, y)$ <u>iff</u> $\exists\, k \in NN: + (x, k) = y$

$< (3, 5)$ since $2 \in NN$ and $+ (3, 2) = 5$

NN - representation & interpretation

SYNTAX

numeral ::= digit | numeral digit
digit ::= { 0, 1, 2, 3, …, r-1}

SEMANTICS

0 → zero
1 → one
2 → two
…

single digit numerals

$\ldots d_3 d_2 d_1 d_0$

$d_0 * r^0$
$+ \; d_1 * r^1$
$+ \; d_2 * r^2$
$+ \; d_3 * r^3$
:

multi digit numerals

the meaning of the *composite numeral* is inferred
from the meaning of its *constituent parts*

meaning is a function that maps a string of d's into a unique number

OPERATIONS

+	4	2
	5	1
	6	0

Great, but what about + 13137 29251 ?

Answer Part 1: automate

$12137 = 1*10^4 + 3*10^3 + 1*10^2 + 3*10^1 + 7*10^0$

$29251 = 2*10^4 + 9*10^3 + 2*10^2 + 5*10^1 + 1*10^0$

since **(a + b) + (x + y) = (a +x) + (b + y)** and **(ax + bx = (a +b)x**

1	3	1	3	7	
2	9	2	5	1	
3	2	3	8	8	*sum*
1	0	0	0	0	*carry*
4	2	3	8	8	

Answer Part 2: automate further

32	16	8	4	2	1		
	1	0	1	1	1		23
	0	1	0	1	1		11
	1	1	1	0	0	*sum*	
	0	0	1	1		*carry*	
	1	1	0	1	0		
0	0	1	0	0			
0	1	0	0	1	0		
0	1	0	0	0			
0	0	0	0	1	0		
1	0	0	0	0			
1	**0**	**0**	**0**	**1**	**0**		34
0	0	0	0	0			
32	16	8	4	2	1		

Answer Part 3: automate further still

x \ y	0	1	
0	0	1	*sum*
	0	0	*carry*
1	1	0	*sum*
	0	1	*carry*

sum	0	1
0	0	1
1	1	0

↑ **x_or**

carry	0	1
0	0	0
1	0	1

↑ **and**

	1	**0**	**1**	**1**	**1**	a
	0	**1**	**0**	**1**	**1**	b
	1	1	1	0	0	a := a **x_or** b
	0	0	1	1		b := **l_shift** (a **and** b)
	1	1	0	1	0	.
0	0	1	0	0		.
0	1	0	0	1	0	.
0	1	0	0	0		
0	0	0	0	1	0	
1	0	0	0	0		
1	**0**	**0**	**0**	**1**	**0**	
0	0	0	0	0		

Integers

on Cartesian Product $\mathbf{NN} \times \mathbf{NN} = \{(m, n): m, n \in \mathbf{NN})$

define a relation $\approx$ $(m_1, n_1) \approx (m_2, n_2)$ **iff** $(m_1 + n_2) = (m_2 + n_1)$

$\approx$ is relation of equivalence since it is **reflexive**, **symmetric** and **transitive**

Reflexive $(m, n) \approx (n, m)$ **since** $m + n = n + m$

Symmetric

if $(m_1, n_1) \approx (m_2, n_2)$ **then** $m_1 + n_2 = m_2 + n_1$ *by definition*

and $m_2 + n_1 = m_1 + n_2$

hence $(m_2, n_2) \approx (m_1, n_1)$

Transitive

suppose we have $(m_1, n_1) \approx (m_2, n_2)$ **and** $(m_2, n_2) \approx (m_3, n_3)$

by definition

$m_1 + n_2 = m_2 + n_1$

$m_2 + n_3 = m_3 + n_2$

adding sides

$m_1 + \cancel{n_2} + \cancel{m_2} + n_3 = \cancel{m_2} + n_1 + m_3 + \cancel{n_2}$

hence $m_1 + n_3 = n_1 + m_3$

and so $(m_1, n_1) \approx (m_3, n_3)$

relation of equivalence in a non-empty set X divides this set into disjoint, non-empty subsets (classes of equivalence) in the following way:

two elements x, y ε **X belong to the same class iff** x ≈ y

|x| = {y ε **X :** x ≈ y}

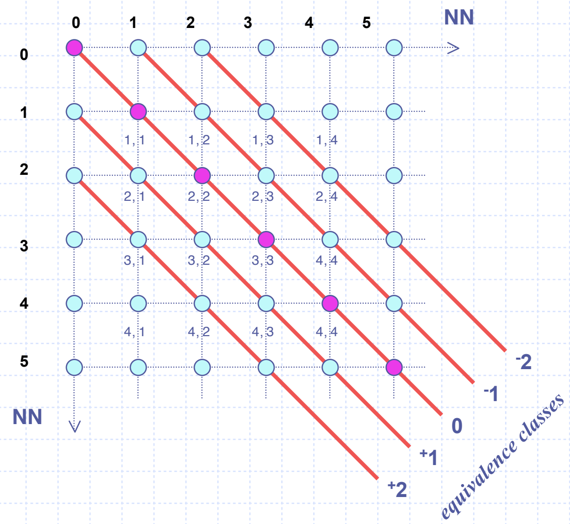
integer may thus be defined as an equivalence class:

|(m, n)| = {(p, q) ε (**NN** × **NN**) **:** (m, n) ≈ (p, q)}

|(1, 1)| = {(0, 0), (1, 1), (2, 2) …} **integer zero**

|(1, 0)| = {(1, 0), (2, 1), (3, 2) …} **integer** $^{+}$**1**

|(0, 1)| = {(0, 1), (1, 2), (2, 3) …} **integer** $^{-}$**1**



Integer addition

$|(m_1, n_1)| \oplus |(m_2, n_2)| = |(m_1 + m_2), (n_1 + n_2)|$

Integer multiplication

$|(m_1, n_1)| \otimes |(m_2, n_2)| = |\ (m_1 \times m_2 + n_1 \times n_2)\ ,\ (m_1 \times n_2 + n_1 \times m_2)\ |$

NN-arithmetic is isomorphic to **IN-arithmetics**

Rational numbers can be defined similarly where

relation $\approx$ $(p_1, q_1) \approx (p_2, q_2)$ **iff** $(p_1 \otimes q_2) = (p_2 \otimes q_1)$
$p, q \in$ **IN**

We can then
represent, reason about, process
the **numbers** by using **numerals**
i.e. in detachment from their meaning

It took us around 7000 years to be able to do so

Can we do the same with more abstract symbols ?

Suppose we need to evaluate the expression

(7 + x) * (8 + 5 * x) for **x = 4**

→ (7 + 4) * (8 + 5 * 4)
→ (7 + 4) * (8 + 20)
→ (7 + 4) * 28
→ 11 * 28
→ 308

→ (7 + 4) * (8 + 5 * 4)
→ 11 * (8 + 5 * 4)
→ 11 * (8 + 20)
→ 11 * 28
→ 308

Church-Rosser property - the order of evaluations is immaterial

Similar evaluation applied to symbols

first (sort (append (BANANA, LEMMON) (sort (GRAPE, APPLE, KIWI))))

first (sort (append (BANANA, LEMMON) (APPLE, GRAPE, KIWI)))

first (sort (BANANA, LEMMON, APPLE, GRAPE, KIWI))

first (APPLE , BANANA, GRAPE, LEMMON, KIWI)

APPLE

Do variable names matter?

$\int x\, dx \longleftrightarrow \int y\, dy$

not (A or B) ≡ (not A) and (not B) ⟷ not (X or Y) ≡ (not X) and (not Y)

but

$\int x \sin y\, dx \not\longleftrightarrow \int y \sin y\, dy$

What are the rules for?

$a(b + c)^2 - (ab^2 + ac^2 + abc)$

$\rightarrow a(b^2 + 2bc + c^2) - (ab^2 + ac^2 + abc)$

$\rightarrow (ab^2 + 2abc + ac^2) - (ab^2 + ac^2 + abc)$

$\rightarrow ab^2 + 2abc + ac^2 - ab^2 - ac^2 - abc$

$\rightarrow 2abc - abc$

$\rightarrow abc$

complex expression

simple(r) expression

f(x) = x + 5   + (x, 5)   or   **plus** (x, 5)

the meaning
(function abstraction)

lambda expression → f = λx. x + 5

λx. salt-cover (x)

peanuts → cover-with salt → salted peanuts

λx. salt-cover (x) (peanuts) → salt-cover_peanuts
λx. salt-cover (x) (meat) → salt-cover_meat
λx. salt-cover (x) (banana) → salt-cover_banana

λy. (λx. y-cover (x)) (sugar) → λx. sugar-cover (x)

λz. (λy. (λx. y-z (x))) (cover) → λy. (λx. y-cover (x))

λz. (λy. (λx. y-z (x))) (free) → λy. (λx. y-free (x))

(Haskell Curry,
M Schönfinkel)

currying

functions of n arguments can be represented by n-fold iteration of application

instead of applying the function to two arguments	f(X, Y)	plus (3, 5)
apply it to the first argument and then apply the result to the second argument	(f(X))Y	((plus3) 5)

more formally (λ.(xy) F = λx. λy. F

λ-expression ::= constant
| variable
| <λ-expression> < λ-expression > *application*
| λ-expression. < λ-expression > *abstraction*
| (λ-expression)

Notation

complex λ-expression	**M, N, P, Q, ...**
variables	**x, y, z, ...**
constants	**300000**
application	**+5, add (5)** **+ 2 3 ≡ ((+2) 3)**
MNPQ means	**(((MN)P))** *(association to the left)*
built-in functions	e.g. **add ,** *neither constants nor λ-functions, defined for convenience, can be evaluated*
abstraction	**λx. + 1 x** **(λx. (λy. * 5 y) (+ x 3))12**

body

λx. x y

free — variable — must know its value (from outside)

bound — variable — argument of the function

variable declaration

+x (λx. + x 3) 4

free bound

(λx. xy (λy. y))

(λx. λ y. z ((λz. z (λx. y)))

free variables

(λx. λ y. xz (y z)) (λx. y (λy. y)))

MANIPULATING EXPRESSIONS

$\lambda x.\ +\ x\ 1 \ \to_\alpha\ \lambda y.\ +\ y\ 1$

variable names are arbitrary

$\lambda x.\ (\lambda\ y.\ yx) \not\to_\alpha\ \lambda x.\ (\lambda\ x.\ xx)$

but

$\lambda x.\ \underbrace{(\lambda\ y.\ yx)}_{E} \to_\alpha\ \lambda x.\ (\lambda\ z.\ zx)$

α -conversion rule

$\boxed{\lambda x.\ E\ \to_\alpha\ \lambda z.\ [z \leftarrow x]\ E}$

replace any bound **x** by **z** in **E**
provided that **z** doesn't occur in **E**

- $\lambda y.\ x\ \not\to_\alpha\ \lambda y.\ y$ **x** is free in
- $\lambda x.\ (\lambda\ y.\ +\ x\ y) \not\to_\alpha\ \lambda x.\ (\lambda\ x.\ +\ xx)$ **x** is free in
- $\lambda x.\ f\ x\ \not\to_\alpha\ \lambda x.\ g\ y$ **f** is free in
- $\lambda x.\ (\lambda\ y.\ x\ y) \to_\alpha\ \lambda f.\ (\lambda\ y.\ f\ y)$

- $(\lambda x. + x\ 5)\ 4 \to_\beta + 4\ 5$
- $(\lambda x. * x\ x)\ 5 \to_\beta * 5\ 5$
- $(\lambda x. 16)\ y \to_\beta 16$ whatever → ▭ → 16
- $(\lambda x. (\lambda y. * x\ y))\ 4\ 5 \to_\beta (\lambda y. * 4\ y)\ 5 \to_\beta * 4\ 5$
- $(\lambda a. a\ 2)\ (\lambda b. + b1) \to_\beta (\lambda b. + b\ 1)\ 2 \to_\beta + 2\ 1$

β -conversion rule $\boxed{(\lambda x. P)\ Q \to_\beta [\, x \leftarrow Q]\ P}$ provided that bound variables of P are distinct from free variables of Q

all (free in P) x's in P get replaced by Q

- $(\lambda x. (\lambda y. x\ y))\ y \to_\beta \lambda y. y\ y$?! WRONG

bound free

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

$\to_\beta (\lambda z. y\ z)$

$\to_\alpha \lambda x. yx$

- $\lambda x. (\lambda y.\ \text{div}\ x\ y)\ 6\ 3$

  $\to_\beta (\lambda y.\ \text{div}\ 6\ y)\ 3$

  $\to_\beta \text{div}\ 6\ 3$

  $\to_\delta 2$

δ -conversion rule

evaluation of the built-in functions

- $\lambda x.\lambda y. + x\ ((\lambda x. - x\ 4)\ y)\ 5\ 6$

  $\to_\beta \lambda x.\ \lambda y. + x\ (-\ y\ 4)\ 5\ 6$

  $\to_\beta \lambda x.\ + x\ (-\ 5\ 4)\ 6$

  $\to_\beta + 6\ (-\ 5\ 4)$

  $^*\!\!\to_\delta + 7$

multiple application of δ-conversion

- $(\lambda x. + 5\ x)\ 4 \to_\beta + 5\ \ 4 \to_\delta 9$

  $(\lambda x. \boxed{+\ 5}\ x) \to_\eta + 5$

$\boxed{(\lambda x.\ F\ x) \to_\eta F}$

η -conversion rule

provided x does not occur free in F

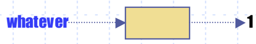
λ-expression that contains no reducible sub-expression is said to be in **normal form**

- not every expression has a normal form, for instance
  (λx. x x) (λx. x x) → (λx. x x) (λx. x x) → (λx. x x) (λx. x x) → ...

- some reduction orders are more efficient than others:

(1) **(λx. 1) (λx. x x) (λx. x x)**

(λx. 1) (whatever) → 1



but

(2) **(λx. 1)(λx. x x) (λx. x x) → (λx. 1)(λx. x x) (λx. x x) →...**

NORMAL ORDER	APPLICATIVE ORDER
(λy. (λ x. (λz. (+ z x)) 4) y) 5	**λy. (λ x. (λz. (+ z x)) 4) y) 5**
→(λ x. (λz. (+ z x)) 4) 5	**→ λy. (λ x. (+ 4 x) y) 5**
→(λz. (+ z 5)) 4	**→ λy. (+ 4 y) 5**
→(+ 4 5)	**→ (+ 4 5)**
→ 9	**→ 9**

Church-Rosser Theorem

If $\lambda\text{-exp}_1 \leftrightarrow \lambda\text{-exp}_2$ then there exists λ-exp such that

$\lambda\text{-exp}_1 \leftrightarrow \lambda\text{-exp}$

$\lambda\text{-exp}_2 \leftrightarrow \lambda\text{-exp}$

If $\lambda\text{-exp}_1 \leftrightarrow \lambda\text{-exp}_2$ and $\lambda\text{-exp}_2$ is in normal form then there exist a normal form reduction $\lambda\text{-exp}_1 \rightarrow \lambda\text{-exp}_2$

how does it work for numbers?

λf. λx. x	zero
λf. λx. f x	one
λf. λx. f (f x)	two
λf. λx. f (f (f x))	three
......	......

Church numerals

how many times **f** is applied to **x**

successor **succ** ≡ λn.λf.λx.(f ((n f) x))

succ zero ≡ λn.λf.λx. (f ((n f) x)) (λf.λx. x)

→ λf.λx. (f (λf.λx. x f) x)

→ λf.λx. (f (λg.λy. y g) x)

→ λf.λx. (f (λy. y) x)

→ λf.λx. (f x) → one

- **add** ≡ λm.λn.λf.λx.(((m succ) n) f) x) f ((n f) x))
  ≡ λm.λn.λf.λx.m f (n f x)

- **mult** ≡ λm.λn.λf.m f (n f)

- **exp** ≡ λm.λn. (m n)

... and for booleans?

IDENTITY	λx. x	I	I M ≡ M, I I ≡ I
True	λx. λy. x	K	K M N ≡ M
False	λx. λy. y	K*	K* M N ≡ N
NOT	λx. (x F) T		
AND	λx. λy ((x y)F)		
OR	λx. λy ((x T) y)		

if .... then ... else?

- if True B C → B (1)
  if False B C → C (2)

suppose we take λx. x for if

then (1) becomes
(λx. x) (λx. λy. x) B C → (λx. λy. x) B C → (λy. B) C → B

and (2) becomes
(λx. x) (λx. λy. y) B C → (λx. λy. y) B C → (λy. C) → C

... recursion?

- let

  Y ≡ λf. (λx. f (x x)) ((λx. f (x x)) recursion combinator)

  YR ≡ λf. (λx. f (x x)) ((λx. f (x x)) R

  → (λx. R (x x)) ((λx. R (x x))
  → R (λx. R (x x)) ((λx. R (x x))
  → ...

  keeps generating R's

  a function that calls (a function) **f** and regenerates itself

  YR ≡ R (YR)

Turing Machine, **μ-recursive functions** (Gödel), **λ-calculus** (Church), **formal grammars** (Post), **combinatory logic** (Schönfinkel, Curry)

are computationally equivalent

Church Thesis **every intuitively computable function is λ-definable**

λ-calculus is about
- **processing functions by manipulating their abstractions using application and formal conversion rules**

λ-calculus
everything in the computational process is represented via functions;
there are no other objects or types (bool, int, chars, strings, etc.) ;
if they are needed they must be represented using functions

λ-calculus let us to analyse the functions
- **without having to name them**
- **seeing their abstractions at all times**
- **being free from their intuitive properties**

normal order β-reduction models lazy evaluation functional languages, such as Haskell