History

- Formal mathematical system
- Simplest programming language
- Intended for studying functions, recursion
- Invented in 1936 by Alonzo Church (1903-1995)
  - Same year as Turing’s paper
Warning

- May seem trivial and/or irrelevant now
- May remind you of brainf*^&
- Had a tremendous influence in PLs
  - λ-calculus → Lisp → everything

- Context in the early 60s:
  - Assembly languages
  - Cobol
  - Unstructured programming
What is Calculus?

Calculus is a branch of mathematics that deals with limits and the differentiation and integration of functions of one or more variables.
Real Definition

- A *calculus* is just a bunch of rules for manipulating symbols.
- People can give meaning to those symbols, but that’s not part of the calculus.
- Differential calculus is a bunch of rules for manipulating symbols. There is an interpretation of those symbols corresponds with physics, slopes, etc.
Syntax

\[ M ::= x \quad \text{(variable)} \]
\[ \quad | \lambda x.M \quad \text{(abstraction)} \]
\[ \quad | MM \quad \text{(application)} \]

Nothing else!
- No numbers
- No arithmetic operations
- No loops
- No etc.

Symbolic computation
Syntax reminder

$\lambda x. M \rightarrow \text{function}(x) \{ M \}$

$L M$, e.g. $\lambda x. N \ y \rightarrow \text{apply } L \text{ to } M$
The binding operator \( \lambda \) binds the variable \( x \) in the \( \lambda \)-term \( x.M \)

- \( M \) is called the scope of \( x \)
- \( x \) is said to be a bound variable
Free variables are all symbols that aren’t bound (duh)

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(MN) &= FV(M) \cup FV(N) \\
FV(x.M) &= FV(M) - x
\end{align*}
\]
Renaming of bound variables

\[ \lambda x. M = \lambda y. ([y/x]M) \quad \text{if } y \text{ not in } \text{FV}(M) \]

\[ \alpha\text{-conversion} \]

i.e. you can replace \( x \) with \( y \) aka “renaming”
Evaluating function application: \((\lambda x. e_1) \, e_2\)
- Replace every \(x\) in \(e_1\) with \(e_2\)
- Evaluate the resulting term
- Return the result of the evaluation

Formally: “\(\beta\)-reduction” (aka “substitution”)
- \((\lambda x. e_1) \, e_2 \rightarrow_{\beta} e_1[e_2/x]\)
- A term that can be \(\beta\)-reduced is a redex (reducible expression)
- We omit \(\beta\) when obvious
Note again

- Computation $\equiv$ pure symbolic manipulation
  - Replace some symbols with other symbols
Scoping etc.

- Scope of \( \lambda \) extends as far to the right as possible
  - \( \lambda x.\lambda y.xy \) is \( \lambda x.(\lambda y.(x y)) \)

- Function application is left-associative
  - \( xyz \) means \((xy)z\)

- Possible syntactic sugar for declarations
  - \((\lambda x.N)M\) is \( \text{let } x = M \text{ in } N \)
  - \((\lambda x.(x + 1))10\) is \( \text{let } x=10 \text{ in } (x+1) \)
Multiple arguments

- $\lambda(x,y).e$ ???
  - Doesn’t exist

- Solution: $\lambda x.\lambda y.e$ [remember, $(\lambda x.(\lambda y.e))$]
  - A function that takes $x$ and returns another function that takes $y$ and returns $e$
  - $(\lambda x.\lambda y.e) a b \rightarrow (\lambda y.e[a/x]) b \rightarrow e[a/x][b/y]$
  - “Currying” after Curry: transformation of multi-arg functions into higher-order functions

- Multiple argument functions are nothing but syntactic sugar
Boolean Values and Conditionals

- True = λx.λy.x
- False = λx.λy.y
- If-then-else = λa.λb.λc. a b c = a b c

For example:
- If-then-else true b c
  → (λx.λy.x) b c → (λy.b) c → b
- If-then-else false b c
  → (λx.λy.y) b c → (λy.y) c → c
If True M N = (λa.λb.λc.abc) True M N

→ (λb.λc. True b c) M N
→ (λc. True M c) N
→ True M N

= (λx.λy.x) M N
→ (λy. M) N
→ M
Numbers…

- Numbers are counts of things, any things. Like function applications!

- $0 = \lambda f. \lambda x. x$
- $1 = \lambda f. \lambda x. (f \ x)$
- $2 = \lambda f. \lambda x. (f (f \ x))$
- $3 = \lambda f. \lambda x. (f (f (f \ x)))$
- $\ldots$
- $N = \lambda f. \lambda x. (f^N \ x)$

Church numerals
Successor

\[ \text{succ} = \lambda n. \lambda f. \lambda x. f (n f x) \]

Want to try it on succ(1)?

\[ \lambda n. \lambda f. \lambda x. f (n f x) (\lambda f. \lambda x. (f x)) \]

\[ \to \lambda f. \lambda x. f ((\lambda f. \lambda x. (f x)) f x) \]

\[ \to \lambda f. \lambda x. f (f x) \]
There’s more

- Reading materials
Recursion

\[(\lambda n. \right)
  \left(\text{if (zero? } n) \right)
  1 \\
  (* n (f (sub1 n))))\]

Free variable
Recursion – The Y Combinator

\[ Y = \lambda t. (\lambda x. t (x x)) (\lambda x. t (x x)) \]

\[
Y a = \lambda t. (\lambda x. t (x x)) (\lambda x. t (x x)) a \\
= (\lambda x. a (x x)) (\lambda x. a (x x)) \\
= a ((\lambda x. a (x x)) (\lambda x. a (x x))) \\
= a (Y a)
\]

\[ Y a = a \text{ applied to itself!} \]

\[ Y a = a (Y a) = a (a (Y a)) = a (a (a (Y a))) = ... \]
Factorial again

\( \lambda n. \) 
\( (\text{if } (\text{zero? } n) \) 
\( 1 \) 
\( (* n (f (\text{sub1 } n)))) \) 

\( \lambda f. \lambda n. \) 
\( (\text{if } (\text{zero? } n) \) 
\( 1 \) 
\( (* n (f (\text{sub1 } n)))) \) 

Now it’s bound!
Does it work?

F takes one function and one number as arguments

\[(Y \ F) \ 2 = F \ (Y \ F) \ 2\]
\[= \lambda f. \lambda n. (\text{if } (\text{zero? } n) \ 1 \ (* \ n \ (f \ (\text{sub1 } n))))\]
\[\quad ((\lambda t. (\lambda x. t \ (x \ x)) \ (\lambda x. t \ (x \ x)))\]
\[\quad \quad (\lambda f. \lambda n. (\text{if } (\text{zero? } n) \ 1 \ (* \ n \ (f \ (\text{sub1 } n))))))\]
\[2\]
\[= \text{if } (\text{zero? } 2) \ 1 \ (* \ 2 \ (Y \ F \ (\text{sub1 } 2)))\]
\[= (* \ 2 \ (Y \ F \ (\text{sub1 } 2)))\]
\[= (* \ 2 \ (Y \ F \ 1))\]
\[= \ldots\]
\[= (* \ 2 \ 1)\]
\[= 2\]
Points to take home

- Model of computation completely different from Turing Machine
  - pure functions, no commands
- Church-Turing thesis: the two models are equivalent
  - What you can compute with one can be computed with the other
- Inspiration behind Lisp (late 1950s)
- Foundation of all “functional programming” languages