

History

- **Alonzo Church and Stephen Cole Kleene in the 1930s**
- **Church and Turing published independent papers in 1936**
 - Impossible to write a program that for any statement in arithmetic decides whether it is true or false.
 - Thus, a general solution to the Entscheidungsproblem (decision problem) is impossible.
- **Used to define what a computable function is.**
 - There is no algorithm that can decide, for any two lambda calculus expressions, whether they are equivalent.
 - First problem to be shown to be undecidable. (Even before the halting problem.)

Why interesting?

- **Theory of computation: Equivalent alternative to a Turing machine**
- **Basis of functional programming**

Overview

- **Three constructs:**
 - symbols (identifiers), e.g., x
 - function definitions, e.g., $(\lambda x . x + 1)$
 - function applications, e.g., $((\lambda x . x + 1) 1)$
 - syntax: apply f to x is written as " $(f x)$ " or " $f x$ " instead of " $f(x)$ "
- **Example**
 - Definition of function that adds 1 to its argument: $(\lambda x . x + 1)$
 - Apply it to the number 3: $((\lambda x . x + 1) 3)$
 - Rewrite by substituting 3 for x : $((\lambda x . x + 1) 3) = 3 + 1 = 4$
- **Can leave out parentheses in many cases.**
 - Left associative
 - $f x y = (f x) y$
 - Function application has higher precedence than function definition
 - $\lambda x . y z = \lambda x . (y z)$
 - Example: $(\lambda x . x + 1) 3$
- **Functions have only one argument, but we can fake multiple arguments**
 - "Currying" (named after Haskell Curry)
 - Example:
 - Subtract b from a
 - Want to find the ... in $(... a b)$ so that it means subtract b from a
 - What does the following function do?
 - $(\lambda x . (\lambda y . x - y))$
 - To see what it does, apply it to a :
 - $((\lambda x . (\lambda y . x - y)) a) = (\lambda y . a - y)$
 - Returns a function that subtracts its argument from a
 - Try to apply the function to a and b
 - $((\lambda x . (\lambda y . x - y)) a b) = ((\lambda y . a - y) b) = a - b$

- Two noteworthy things:
 - We can get away with a language that only has single arguments
 - Of course we would like multiple arguments in a "real" language, but it's nice to know that the semantics of multiple arguments can be defined in terms of single arguments.
 - Functions in Haskell can be curried, i.e., applied to one argument at a time. Useful in some cases.
 - Functions can return functions!

More formally...

- **Why?**
 - We have been applying functions, etc., in a hand-waving way.
 - Before we get confused, let's step back and be a little more precise.
- **Grammar**
 - expression := identifier
 - expression := '(' 'λ' identifier '.' expression ')'
 - expression := '(' expression expression ')'
- **Bound and free names**
 - Will need this in a moment to get rewrite and substitution rules right.
 - Consider the expression $E1 = (\lambda x . x + y)$
 - The expression $x + y$ has two names.
 - One of them is bound in $E1$ —which one?
 - x is bound: We know what it refers to (the parameter)
 - y is free: It refers to something outside $E1$
 - Depends on whatever is around $(\lambda x . x + y)$
 - Example: $E2 = (\lambda y . (\lambda x . x + y))$
 - Both x and y are bound in $E2$
- **Semantics of lambda calculus defined in terms of three simple rewrite rules**
 - R1 (α -conversion): Renaming
 - Rule: $\lambda y . C = \lambda z . [z/y]C$ where z is not free in C
 - The notation $[z/y]C$ means substitute y with z in C .
 - Will be defined formally later.
 - Intuition: We can change the name of the argument as long as we change all the places it's referenced
 - R2 (β -conversion): Function application
 - Rule: $(\lambda y . C) B = [B/y]C$
 - Intuition: Applying a function to an argument means replacing the formal parameter with the actual parameter
 - R3 (η -conversion): Redundant function elimination
 - Rule: $(\lambda y . (E y)) = E$ if y is not free in E
 - Intuition: If a function just applies its argument to another function, it's useless, and we can just use the other function directly.