CS 162 Programming languages

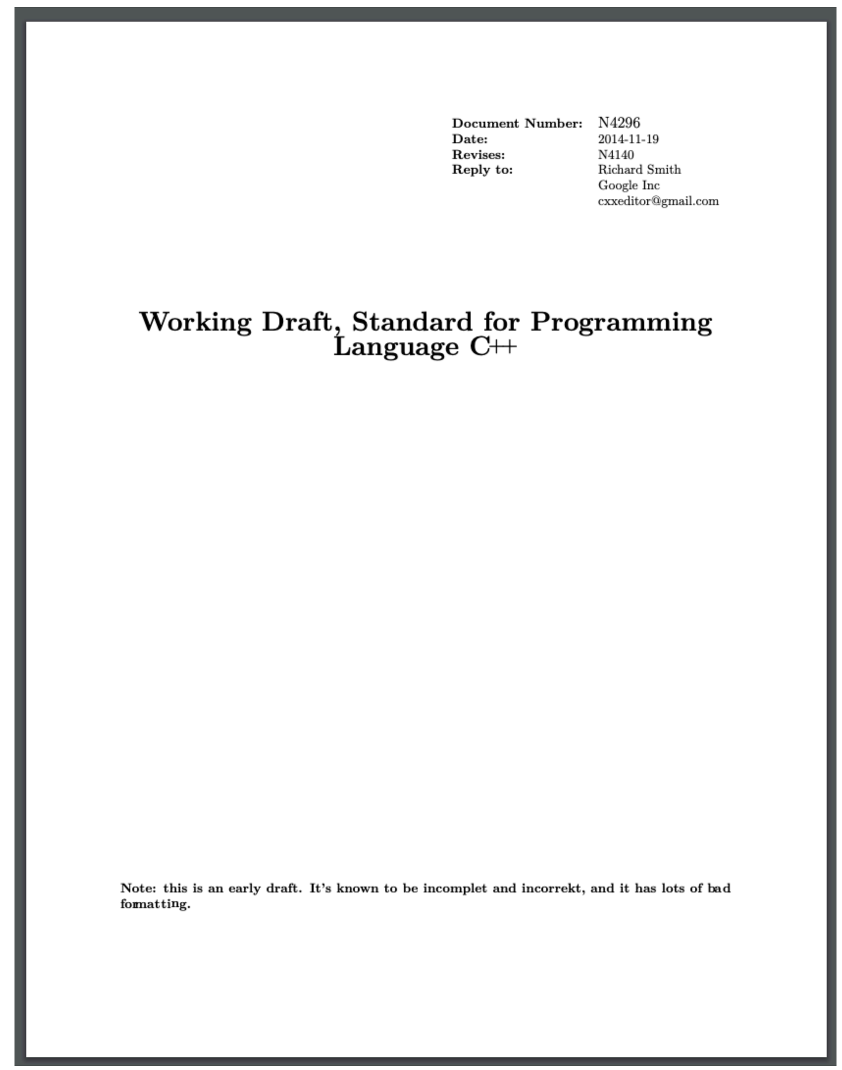
Lecture 2: λ-calculus

Yu Feng

Winter 2021

Your favorite language?

- Assignment
- Booleans, integers, characters, strings, …
- Conditionals
- Loops
- Functions
- Recursion
- References / pointers
- Objects and classes
- Inheritance



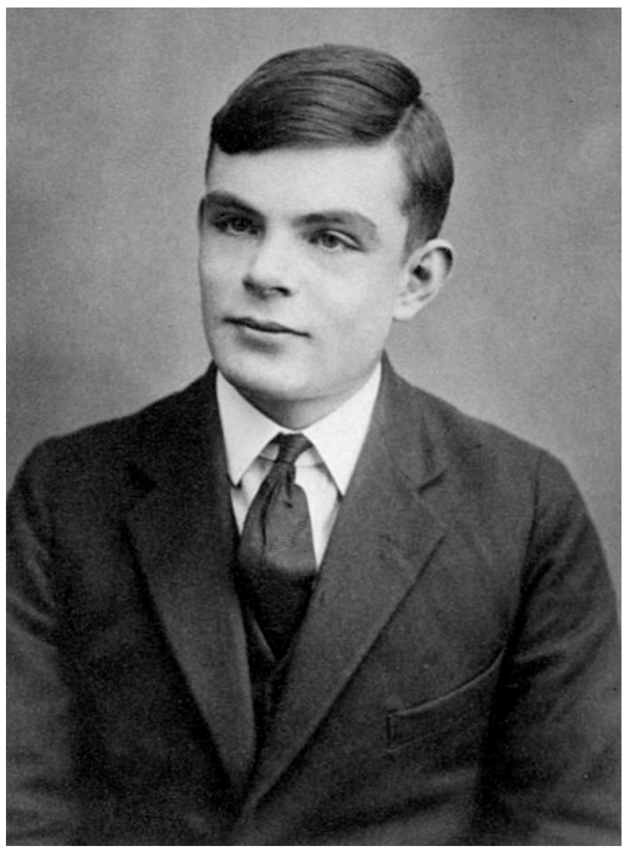
Document Number:	N4296
Date:	2014-11-19
Revises:	N4140
Reply to:	Richard Smith Google Inc cxxeditor@gmail.com

Working Draft, Standard for Programming Language C++

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

1368 pages in 2014!

The smallest universal language



Alan Turing

Alonzo Church

The Calculi of Lambda-Conversion, 1936
ENIAC, 1943

The next 700 languages

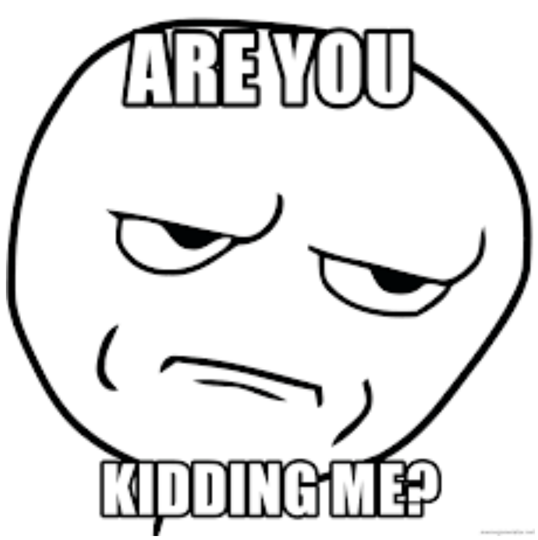
"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

Peter Landin 1966

The λ-Calculus

Has one and ONLY one feature

- Functions



- ~~Assignment~~
- ~~Booleans, integers, characters, strings, …~~
- ~~Conditionals~~
- ~~Loops~~
- Functions
- ~~Recursion~~
- ~~References / pointers~~
- ~~Objects and classes~~
- ~~Inheritance~~

The λ-Calculus

More precisely, the only things you can do are:

Define a function

Call a function

“Free your mind”

Design a programming language

- Syntax: what do programs look like?
  - Grammar: what programs are we allowed to write?
- Semantics: what do programs mean?
  - Operational semantics: how do programs execute step-by-step?

Syntax: what programs look like

$$
\begin{aligned}
e ::= &\ x \\
 \mid &\ \lambda x.\, e \\
 \mid &\ e_1\ e_2
\end{aligned}
$$

\x → e (Haskell)

fun x → e (OCaml)

lambda x. e (λ+)

- Programs are expressions e (also called λ-terms) of one of three kinds:
  - Variable x, y, z
  - Abstraction (i.e. nameless function definition)
    - $\lambda x.\, e$
    - x is the formal parameter, e is the function body
  - Application (i.e. function call)
    - $e_1\ e_2$
    - e_1 is the function, e_2 is the argument

Running examples

$\lambda x.\ x$ — The identity function

$\lambda x.\ (\lambda y.\ y)$ — A function that returns the identity function

$\lambda f.\ f\ (\lambda x.\ x)$ — A function that applies its argument to the identity

Semantics: what programs mean

- How do I execute a λ-term?
- "Execute": rewrite step-by-step following simple rules, until no more rules apply

Similar to simplifying (x+1) * (2x -2) using middle-school algebra

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & \lambda x.\, e \\
\mid \; & e_1 \; e_2
\end{aligned}
$$

What are the rewrite rules for λ-calculus?

Operational semantics

$$(\lambda x\,.\,t_1)\ t_2 \rightarrow [x \mapsto t_2]t_1$$

β-reduction
(function call)

$[x \mapsto t_2]t_1$ means "t_1 with all ***free occurrences*** of x replaced with t_2"

```
inc1(int x ) {
  return x+1
}

inc1(2);
```

$$(\lambda x\,.\,x + 1)\ 2 \rightarrow [x \mapsto 2]x + 1 = 3$$

$$[x \mapsto y]\lambda x\,.\,x = \lambda x\,.\,y$$

What does free occurrences mean?

Semantics: variable scope

The part of a program where a variable is visible

In the expression $\lambda x.\ e$

- x is the newly introduced variable
- e is the scope of x
- any occurrence of x in $\lambda x.\ e$ is bound (by the binder λx)

$\lambda x.\ x$

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

x is bounded

$x\ y$

$\lambda y.\ x\ y$

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

x is free

An occurrence of x in e is **free** if it's *not bound* by an enclosing abstraction

Semantics: free variables

An variable x is free in e if there exists a free occurrence of x in e

We use "FV" to represent the set of all free variables in a term:

$FV(x) = x$
$FV(\lambda x.\ e) = FV(e) - x$
$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

$FV(x\ y) = \{x, y\}$
$FV(\lambda y.\ x\ y) = \{x\}$
$FV((\lambda x.\ \lambda y.\ y)\ x) = \{x\}$

If e has no free variables it is said to be closed, or combinators

Semantics: β-reduction

$$(\lambda x\,.\,t_1)\ t_2 \rightarrow [x \mapsto t_2]t_1$$

β-reduction
(function call)

$[x \mapsto t_2]t_1$ means "t_1 with all ***free occurrences*** of x replaced with t_2"

$$[x \mapsto s]x = s$$
$$[x \mapsto s]y = y (x \neq y)$$
$$[x \mapsto s]\lambda y\,.\,t_1 = \lambda y\,.\,[x \mapsto s]t_1 (y \neq x \wedge y \notin FV(s))$$
$$[x \mapsto s]t_1\ t_2 = [x \mapsto s]t_1\ [x \mapsto s]t_2$$

Semantics: α-renaming

$$\lambda x\,.\,e =_\alpha \lambda y\,.\,[x \mapsto y]e$$

- Rename a formal parameter and replace all its occurrences in the body

$$\lambda x\,.\,x =_\alpha \lambda y\,.\,y =_\alpha \lambda z\,.\,z$$

$$[x \mapsto y]\lambda x\,.\,x = \lambda x\,.\,y$$

$$[x \mapsto y]\lambda x\,.\,x =_\alpha [x \mapsto y]\lambda z\,.\,z = \lambda z\,.\,z$$

Currying: multiple arguments

$$\lambda(x, y)\,.\,e = \lambda x\,.\,\lambda y\,.\,e$$

$$(\lambda(x, y)\,.\,x + y)\ 2\ 3 =$$

$$(\lambda x\,.\,\lambda y\,.\,x + y)\ 2\ 3 = (\lambda y.2 + y)\ 3 = [y \mapsto 3]2 + y = 5$$

Transformation of multi-arguments functions to higher-order functions is called currying (in the honor of Haskell Curry)

TODOs by next lecture

- Join Slack for CS162!
- Install OCaml/λ^+ on your laptop
- Finish Quiz 1 by the end of this Friday