CS 162 Programming languages

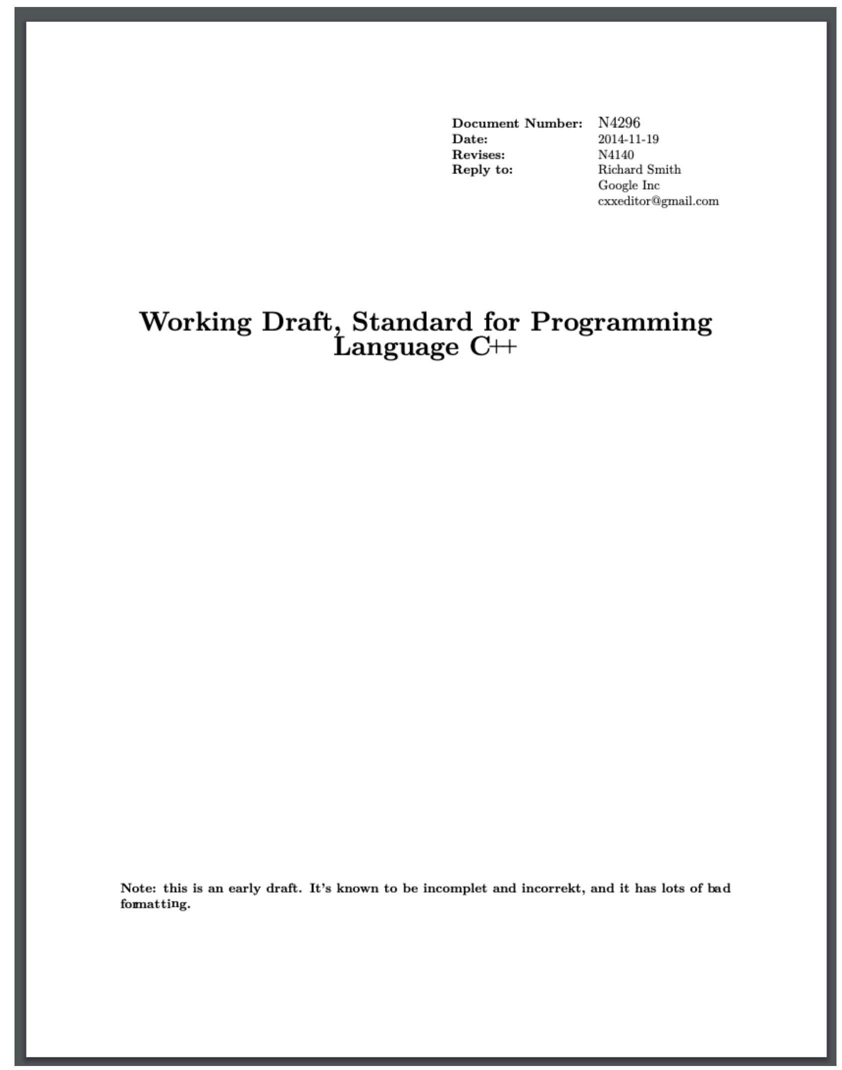
Lecture 2: λ-calculus

Yu Feng

Winter 2020

Your favorite language?

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



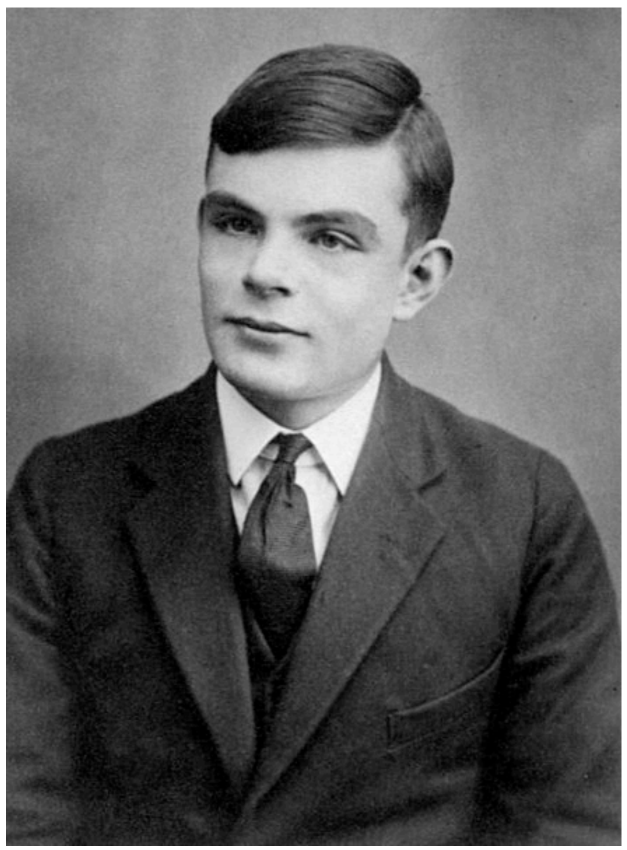
Document Number:	N4296
Date:	2014-11-19
Revises:	N4140
Reply to:	Richard Smith Google Inc cxxeditor@gmail.com

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

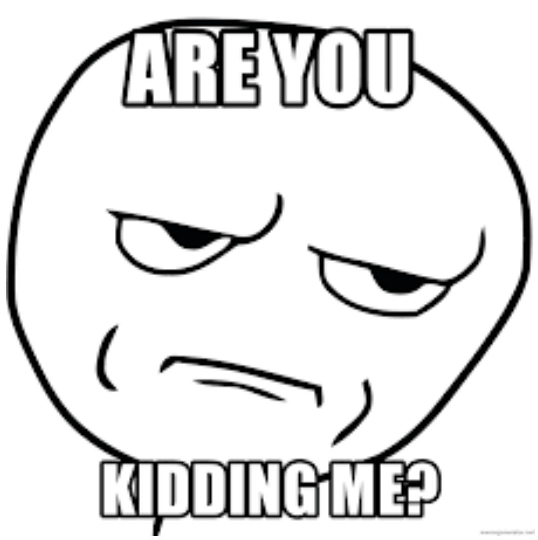
"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

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

$$
\begin{aligned}
e ::=\ & x \\
\mid\ & \backslash x \to e \\
\mid\ & e_1\ e_2
\end{aligned}
$$

$\backslash x \to e$ (Haskell)

fun $x \to e$ (OCaml)

- Programs are expressions e (also called λ-terms) of one of three kinds:
  - Variable x, y, z
  - Abstraction (i.e. nameless function definition)
    - $\backslash x \to e$
    - 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

\x → x

The identity function

\x →(\y → y)

A function that returns the identity function

\f → f (\x → x)

A function that applies its argument to the identity

How to define a function with two arguments?

Syntactic Sugar

instead of	we write
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`

\x → (\y → y)

A function that returns the identity function OR a function that takes two arguments and returns the second one

(((\x →(\y → y)) apple) banana)

=

\x y → y apple banana

Or

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

```
e ::= x
    | \x → e
    | e1 e2
```

What are the rewrite rules for λ-calculus?

Rewrite rules of λ-calculus

- α-renaming

renaming formals

- β-reduction

function call

Let us take a detour and talk about scope

Semantics: variable scope

The part of a program where a variable is visible

In the expression \x → e

- x is the newly introduced variable
- e is the scope of x
- any occurrence of x in \x → e is bound (by the binder \x)

\x → x

\x →(\y → x)

x is bounded

x y

\y → x y

(\x → \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:

$FV(x) = x$
$FV(\backslash x \rightarrow e) = vars(e) - x$
$FV(e_1\ e_2) = FV(e_1) + FV(e_2)$

$FV(x\ y) = \{x, y\}$
$FV(\backslash y \rightarrow x\ y) = \{x\}$
$FV((\backslash x \rightarrow \backslash y \rightarrow y)\ x) = \{x\}$

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

Semantics: β-Reduction

β-Reduction: $(\backslash x \rightarrow e_1)\ e_2 =_\beta e_1[x := e_2]$

where $e_1[x := e_2]$ means "e_1 with all ***free occurrences*** of x replaced with e_2"
In other words, If you see an *abstraction* applied to an *argument*, take the *body* of the abstraction and replace all free occurrences of the *formal* by that *argument*

$(\backslash x \rightarrow x)$ apple $=_\beta$ 

$(\backslash x \rightarrow (\backslash y \rightarrow y))$ apple $=_\beta$???

```
y[x := e]              = y               -- assuming x /= y
(e1 e2)[x := e]        = (e1[x := e]) (e2[x := e])
(\x → e1)[x := e]    = \x → e1
(\y → e1)[x := e]
  | not (y in FV(e)) = \y → e1[x := e]
  | otherwise          = undefined
```

Operational semantics of β-Reduction

https://crypto.stanford.edu/~blynn/lambda/

Semantics: α-renaming

$$\backslash x \rightarrow e =_\alpha \backslash y \rightarrow e[x := y]$$

- Rename a formal parameter and replace all its occurrences in the body
- \x -> e **α-equivalent** to \y -> e[x := y]

$$\backslash x \rightarrow x \quad =_\alpha \quad \backslash y \rightarrow y \quad =_\alpha \quad \backslash z \rightarrow z$$

$$\backslash f \rightarrow f\ x \quad =_\alpha \quad \backslash x \rightarrow x\ x$$

$$(\backslash x \text{ -> } \backslash y \rightarrow y)\ y \quad =_\alpha \quad (\backslash x \rightarrow \backslash z \rightarrow z)\ z$$

$$\backslash x \rightarrow \backslash y \rightarrow x\ y \quad =_\alpha \quad \backslash apple \rightarrow \backslash orange \rightarrow apple\ orange$$

What about the others?

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

TODOs by next lecture

- Join Piazza for CS162!
- Install OCaml on your laptop
- Come to the discussion session if you have questions
- 1st homework will be out by next Wed