ENE4014: Programming Languages

Lecture 17 — Lambda Calculus (Origin of Programming Languages)

Woosuk Lee

2024 Spring

Questions

- Programming languages look very different
  - C, C++, Java, OCaml, Haskell, Scala, JavaScript, etc
- Are they different fundamentally?
- Is there core mechanism underlying all programming languages?

Syntactic Sugar

- Syntactic sugar is syntax that makes a language "sweet": it does not add expressiveness but makes programs easier to read and write.
- For example, we can "desugar" the `let` expression:

$$\texttt{let } x = E_1 \texttt{ in } E_2 \overset{desugar}{\Longrightarrow} (\texttt{proc } x\ E_2)\ E_1$$

- Exercise) Desugar the program:

```
let x = 1 in
  let y = 2 in
    x + y
```

Syntactic Sugar

Q) Identify all syntactic sugars of the language:

$$
\begin{array}{rcl}
\boldsymbol{E} & \rightarrow & \boldsymbol{n} \\
& | & \boldsymbol{x} \\
& | & \boldsymbol{E} + \boldsymbol{E} \\
& | & \boldsymbol{E} - \boldsymbol{E} \\
& | & \texttt{iszero } \boldsymbol{E} \\
& | & \texttt{if } \boldsymbol{E} \texttt{ then } \boldsymbol{E} \texttt{ else } \boldsymbol{E} \\
& | & \texttt{let } \boldsymbol{x} = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
& | & \texttt{letrec } \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
& | & \texttt{proc } \boldsymbol{x}\ \boldsymbol{E} \\
& | & \boldsymbol{E}\ \boldsymbol{E}
\end{array}
$$

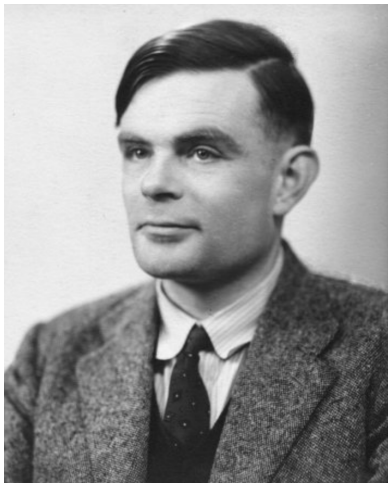
Lambda Calculus (λ-Calculus)

- By removing all syntactic sugars from the language, we obtain a minimal language, called *lambda calculus*:

$$
\begin{array}{rcll}
e & \to & x & \text{variables} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & e\ e & \text{application}
\end{array}
$$

Programming language = Lambda calculus + Syntactic sugars

Origins of Programming Languages and Computer



- In 1935, Church developed λ-calculus as a formal system for mathematical logic and argued that any computable function on natural numbers can be computed with λ-calculus (the model of programming languages).
- In 1936, Turing independently developed Turing machine and argued that any computable function on natural numbers can be computed with the machine (the model of computers)[1].
- Turing machine and lambda calculus appeared as byproducts of a mathematicians dream.

[1] `http://ropas.snu.ac.kr/~kwang/4190.310/15/book-ch2.pdf` (in Korean)

What Mathematicians Dreamed About

Leibniz (1646-1716)

Hillbert (1862-1943)

Hillberts Entscheidungsproblem (1928 @ICM):

- "Is there an algorithm to decide whether a given first-order statement is provable from the axioms using the inference rules? "

cf) Peano Arithmetic

Vocabulary:

$$\Sigma = \{0, 1, +, \cdot, =\}$$

Axioms:

- $\forall x.\ \neg(x + 1 = 0)$
- $\forall x, y.\ x + 1 = y + 1 \implies x = y$
- $\forall x.\ x + 0 = x$
- $\forall x, y.\ x + (y + 1) = (x + y) + 1$
- $\forall x.\ x \cdot 0 = 0$
- $\forall x, y.\ x \cdot (y + 1) = x \cdot y + x$
- $\vdots$

cf) Peano Arithmetic

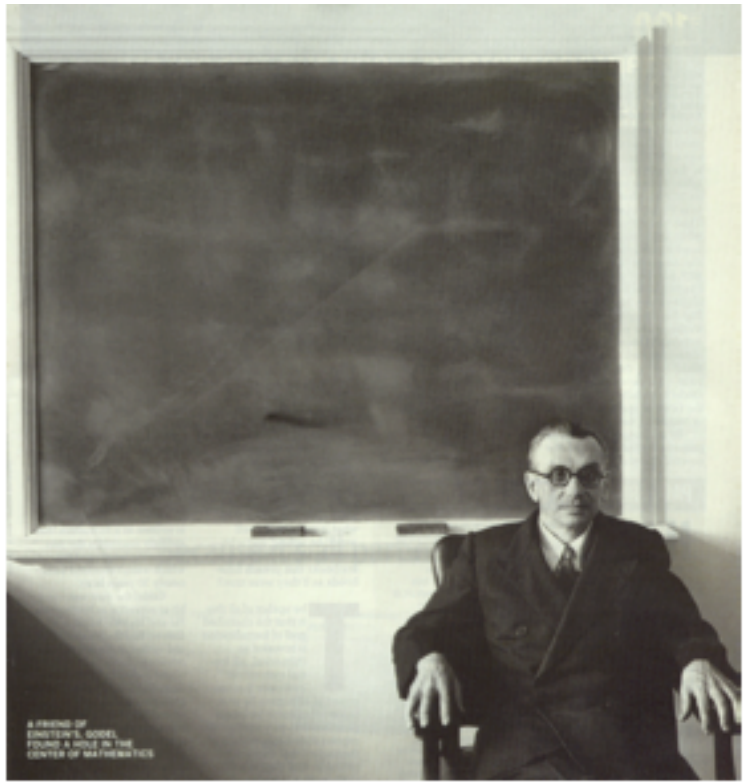
Theorem (Fermats Last Theorem)

$$\forall x, y, z, n.\ x \neq 0 \land y \neq 0 \land z \neq 0 \land n > 2 \implies x^n + y^n \neq z^n$$

- Proposed by Fermat in 1637.
- Completely proved by Wiles in 1995.
- Can we automate the proof search? (Hillberts problem)

Godels Incompleteness Theorems (1931)

A complete and consistent set of axioms for all mathematics is impossible.



Kurt Gödel (1906-1978)

Direct Proofs by Turing and Church

In 1936, Alonzo Church and Alan Turing directly showed that a general solution to the decision problem is impossible.

230 A. M. TURING [NOV. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

1. **Introduction.** There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of a positive integers, such that $f(x_1, x_2, \cdots, x_a) = 2$[2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_a$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly the condition that the function f be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional simplicial manifolds under homeomorphisms. This problem can be interpreted as a problem of elementary number theory in view of the fact that topological complexes are representable by matrices of incidence. In fact, as is well known, the property of a set of incidence matrices that it represent a closed three-dimensional manifold, and the property of two sets of incidence matrices that they represent homeomorphic complexes, can both be described in purely number-theoretic terms. If we enumerate, in a straightforward way, the sets of incidence matrices which represent closed three-dimensional manifolds, it will then be immediately provable that the problem under consideration (to find a complete set of effectively calculable invariants of closed three-dimensional manifolds) is equivalent to the problem, to find an effectively calculable function f of positive integers, such that $f(m, n)$ is equal to 2 if and only if the m-th set of incidence matrices and the n-th set of incidence matrices in the enumeration represent homeomorphic complexes.

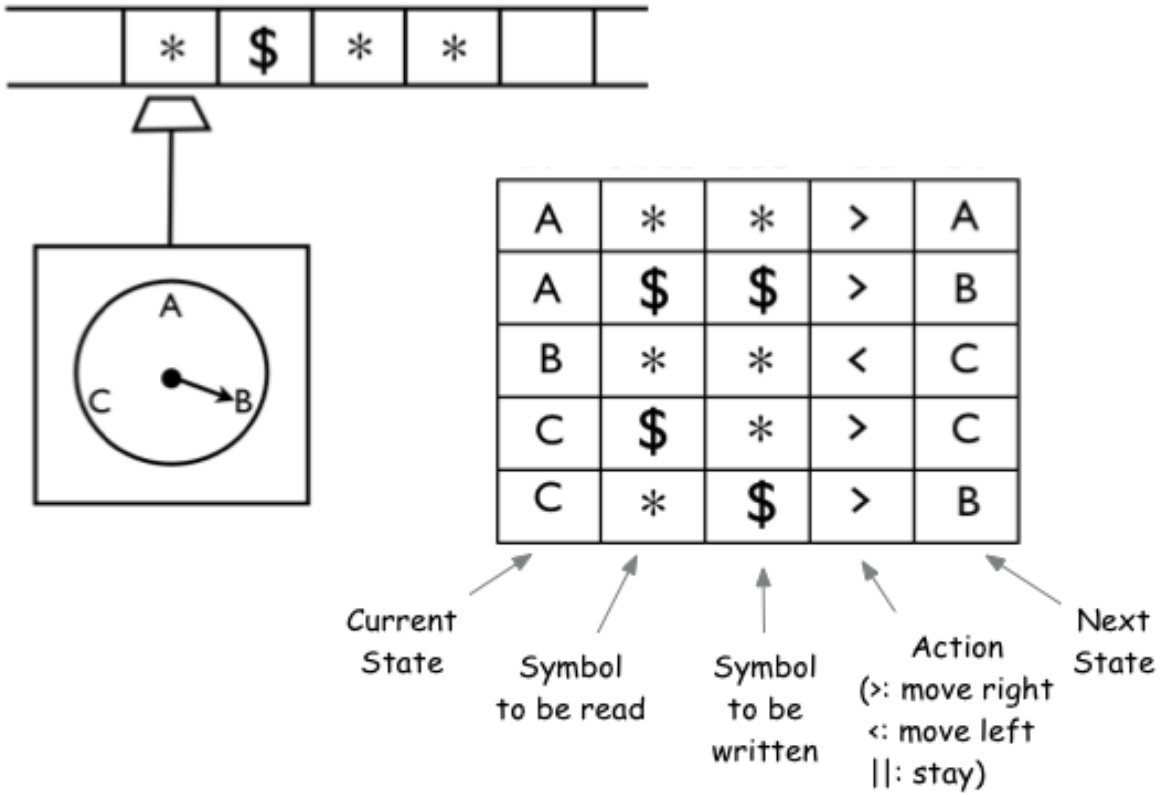
Other examples will readily occur to the reader.

[1] Presented to the American Mathematical Society, April 19, 1935.

[2] The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.

345

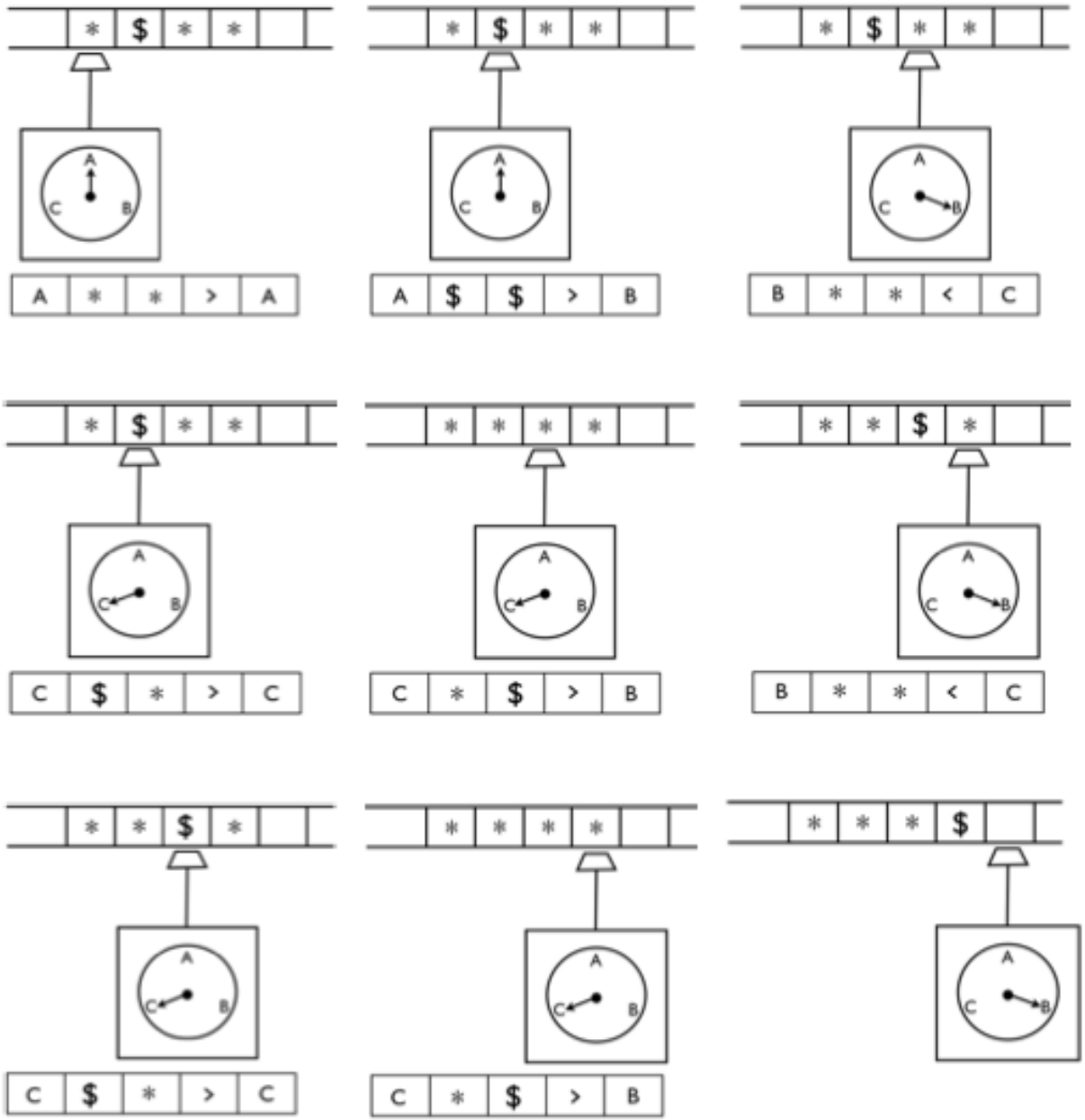
Overview of Turing Machines (Turings Definition of Computation)



Examples:

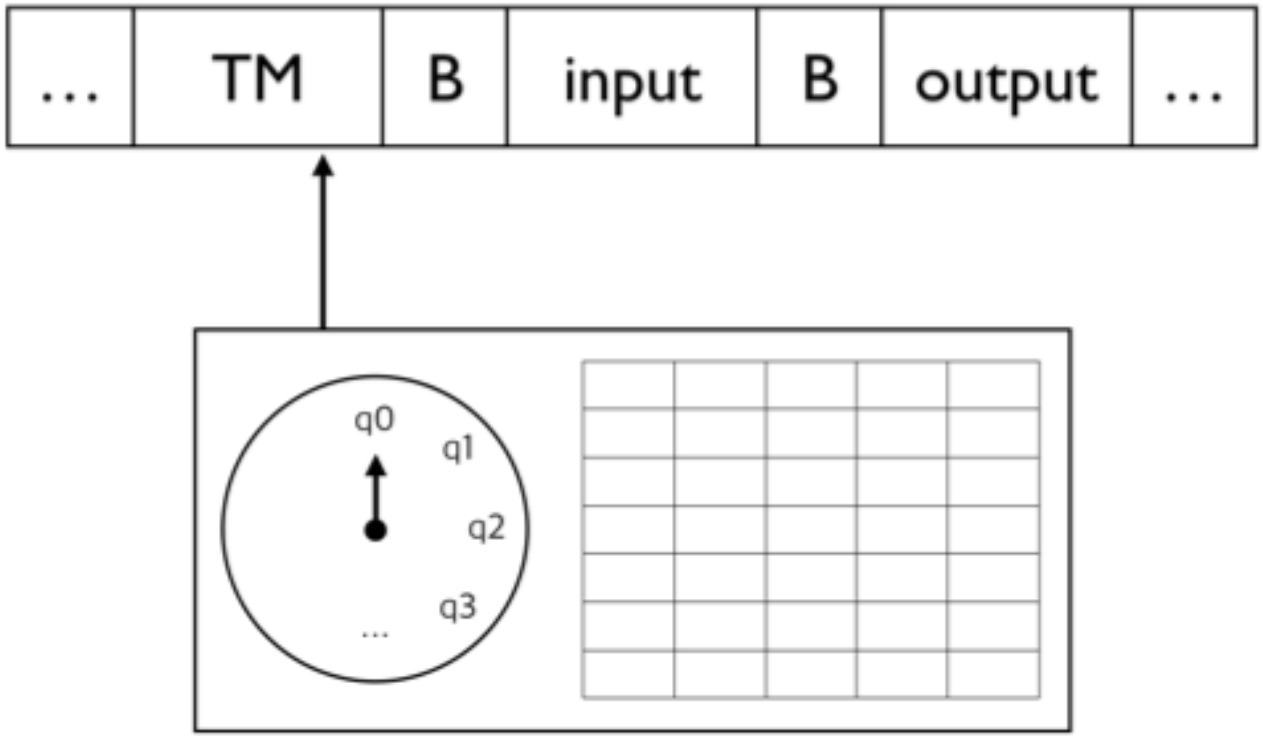
- A machine to write the sequence $\mathbf{0010110111011111011111\ldots}$
- A machine to add 2 and 3: given a tape "*11*111*", writes "*11*111*11111*".

Overview of Turing Machines



Universal Turing Machine

The culmination of Turings work.



UTM is a Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.

Turing's Proof

- Turing reduced the halting problem for Turing machines to the decision problem.
- $\mathbf{H}$: the Turing machine that solves the halting problem
- $\mathbf{A}$: the Turing machine that solves the decision problem
- $\mathbf{A} \implies \mathbf{H}$
- $\mathbf{H}$ is logically impossible.

	I_1	I_2	I_3	$\cdots$
M_1	1	1	0	$\cdots$
M_2	1	0	1	$\cdots$
M_3	1	0	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

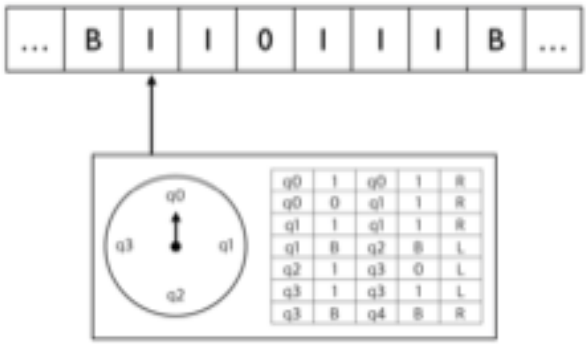
Church's Proof

Church proved that there is no function which decides for two given lambda calculus expressions whether they are equivalent or not.

Church-Turing Thesis

- A surprising fact is that the classes of λ-calculus and Turing machines can compute coincide even though they were developed independently.
- Church and Turing proved that the classes of computable functions defined by λ-calculus and Turing machine are equivalent.
- In other words, Turing machine and lambda calculus are equally powerful.

$$\begin{aligned} E &\rightarrow x \\ &\mid \lambda x.E \\ &\mid E\ E \end{aligned} \quad =$$



A function is λ-computable if and only if Turing computable.

- This equivalence has led mathematicians and computer scientists to believe that these models are "universal": A function is computable if and only if λ-computable if and only if Turing computable.

Impact of λ-Calculus

λ-calculus had immense impacts on programming languages.

- It has been the core of functional programming languages (e.g., Lisp, ML, Haskell, Scala, etc).
- Lambdas in other languages:
  - Java8

```
(int n, int m) -> n + m
```

  - C++11

```
[](int x, int y) { return x + y; }
```

  - Python

```
(lambda x, y:  x + y)
```

  - JavaScript

```
function (a, b) { return a + b }
```

Syntax of Lambda Calculus

$$
\begin{array}{rcll}
e & \rightarrow & x & \text{variables} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & e\ e & \text{application}
\end{array}
$$

- Examples:

$$
\begin{array}{cccc}
 & x & y & z \\
\lambda x.x & \lambda x.y & \lambda x.\lambda y.x & \\
x\ y & (\lambda x.x)\ z & x\ \lambda y.z & ((\lambda x.x)\ \lambda x.x)
\end{array}
$$

- Conventions when writing λ-expressions:
  1. Application associates to the left, e.g., $s\ t\ u = (s\ t)\ u$
  2. The body of an abstraction extends as far to the right as possible, e.g., $\lambda x.\lambda y.x\ y\ x = \lambda x.(\lambda y.((x\ y)\ x))$

Bound and Free Variables

- An occurrence of variable x is said to be *bound* when it occurs inside λx, otherwise said to be *free*.
  - $\lambda y.(x\ y)$
  - $\lambda x.x$
  - $\lambda z.\lambda x.\lambda x.(y\ z)$
  - $(\lambda x.x)\ x$
- Expressions without free variables is said to be *closed expressions* or *combinators*.

Evaluation

To evaluate λ-expression e,

1. Find a sub-expression of the form:

$$(\lambda x.e_1)\ e_2$$

   Expressions of this form are called "redex" (reducible expression).

2. Rewrite the expression by substituting the e_2 for every free occurrence of x in e_1:

$$(\lambda x.e_1)\ e_2 \rightarrow [x \mapsto e_2]e_1$$

   This rewriting is called β-reduction

Repeat the above two steps until there are no redexes.

Evaluation

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

Substitution

The definition of $[x \mapsto e_1]e_2$:

$$
\begin{array}{rcl}
[x \mapsto e_1]x & = & e_1 \\
[x \mapsto e_1]y & = & y \\
[x \mapsto e_1](\lambda y.e_2) & = & \lambda z.[x \mapsto e_1]([y \mapsto z]e_2) \quad (\text{new } z) \\
[x \mapsto e_1](e_2\ e_3) & = & ([x \mapsto e_1]e_2\ [x \mapsto e_1]e_3)
\end{array}
$$

Evaluation Strategy

- In a lambda expression, multiple redexes may exist. Which redex to reduce next?

$$\lambda x.x\ (\lambda x.x\ (\lambda z.(\lambda x.x)\ z)) = id\ (id\ (\lambda z.id\ z))$$

redexes:

$$\underline{id\ (id\ (\lambda z.id\ z))}$$
$$id\ \underline{(id\ (\lambda z.id\ z))}$$
$$id\ (id\ (\lambda z.\underline{id\ z}))$$

- Evaluation strategies:
  - Normal order
  - Call-by-name
  - Call-by-value

Normal order strategy

Reduce the leftmost, outermost redex first:

$$
\begin{aligned}
& \underline{\boldsymbol{id}\ (\boldsymbol{id}\ (\boldsymbol{\lambda z.id}\ z))} \\
\rightarrow\ & \underline{\boldsymbol{id}\ (\boldsymbol{\lambda z.id}\ z))} \\
\rightarrow\ & \boldsymbol{\lambda z.}\underline{\boldsymbol{id}\ z} \\
\rightarrow\ & \boldsymbol{\lambda z.z} \\
\not\rightarrow\ &
\end{aligned}
$$

The evaluation is deterministic.

Call-by-name strategy

Follow the normal order reduction, not allowing reductions inside abstractions:

$$\begin{array}{rl} & \underline{\boldsymbol{id}\ (\boldsymbol{id}\ (\boldsymbol{\lambda z.id}\ \boldsymbol{z}))} \\ \rightarrow & \underline{\boldsymbol{id}\ (\boldsymbol{\lambda z.id}\ \boldsymbol{z})} \\ \rightarrow & \boldsymbol{\lambda z.id}\ \boldsymbol{z} \\ \not\rightarrow & \end{array}$$

The call-by-name strategy is *non-strict* (or *lazy*) in that it evaluates arguments that are actually used.

Call-by-value strategy

Reduce the outermost redex whose right-hand side has a *value* (a term that cannot be reduced any further):

$$
\begin{array}{ll}
 & \boldsymbol{id\ (\underline{id\ (\lambda z.id\ z)})} \\
\rightarrow & \boldsymbol{\underline{id\ (\lambda z.id\ z)}} \\
\rightarrow & \boldsymbol{\lambda z.id\ z} \\
\not\rightarrow &
\end{array}
$$

The call-by-value strategy is *strict* in that it always evaluates arguments, whether or not they are used in the body.

Compiling to Lambda Calculus (with Normal Order Strategy)

Consider the source language:

$$
\begin{array}{rcl}
\boldsymbol{E} & \rightarrow & \boldsymbol{true} \\
& | & \boldsymbol{false} \\
& | & \boldsymbol{n} \\
& | & \boldsymbol{x} \\
& | & \boldsymbol{E} + \boldsymbol{E} \\
& | & \texttt{iszero } \boldsymbol{E} \\
& | & \texttt{if } \boldsymbol{E} \texttt{ then } \boldsymbol{E} \texttt{ else } \boldsymbol{E} \\
& | & \texttt{let } \boldsymbol{x} = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
& | & \texttt{letrec } \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
& | & \texttt{proc } \boldsymbol{x}\ \boldsymbol{E} \\
& | & \boldsymbol{E}\ \boldsymbol{E}
\end{array}
$$

Define the translation procedure from $\boldsymbol{E}$ to $\boldsymbol{\lambda}$-calculus.

Compiling to Lambda Calculus (with Normal Order Strategy)

$\underline{E}$: the translation result of E in λ-calculus

$$
\begin{array}{rcl}
\underline{true} & = & \lambda t.\lambda f.t \\
\underline{false} & = & \lambda t.\lambda f.f \\
\underline{0} & = & \lambda s.\lambda z.z \\
\underline{1} & = & \lambda s.\lambda z.(s\ z) \\
\underline{n} & = & \lambda s.\lambda z.(s^n\ z) \\
\underline{x} & = & x \\
\underline{E_1 + E_2} & = & (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{E_1}\ \underline{E_2} \\
\underline{\texttt{iszero}\ E} & = & (\lambda m.m\ (\lambda x.\underline{false})\ \underline{true})\ \underline{E} \\
\underline{\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3} & = & \underline{E_1}\ \underline{E_2}\ \underline{E_3} \\
\underline{\texttt{let}\ x = E_1\ \texttt{in}\ E_2} & = & (\lambda x.\underline{E_2})\ \underline{E_1} \\
\underline{\texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2} & = & \underline{\texttt{let}\ f = Y\ (\lambda f.\lambda x.E_1)\ \texttt{in}\ E_2} \\
\underline{\texttt{proc}\ x\ E} & = & \lambda x.\underline{E} \\
\underline{E_1\ E_2} & = & \underline{E_1}\ \underline{E_2}
\end{array}
$$

Correctness of Compilation

Theorem

For any expression $\boldsymbol{E}$,

$$[\![\boldsymbol{E}]\!] = \underline{[\![\boldsymbol{E}]\!]}$$

where $[\![\boldsymbol{E}]\!]$ *denotes the value that results from evaluating* $\boldsymbol{E}$.

Examples: Booleans

$$
\begin{aligned}
\underline{\texttt{if } \mathit{true} \texttt{ then } \mathbf{0} \texttt{ else } \mathbf{1}} &= \underline{\mathit{true}}\ \underline{\mathbf{0}}\ \underline{\mathbf{1}} \\
&= (\lambda t.\lambda f.t)\ \underline{\mathbf{0}}\ \underline{\mathbf{1}} \\
&= \underline{\mathbf{0}} \\
&= \lambda s.\lambda z.z
\end{aligned}
$$

Note that

$$
[\![\underline{\texttt{if } \mathit{true} \texttt{ then } \mathbf{0} \texttt{ else } \mathbf{1}}]\!] = \underline{[\![\texttt{if } \mathit{true} \texttt{ then } \mathbf{0} \texttt{ else } \mathbf{1}]\!]}
$$

Church Booleans

- Logical "and":

$$\text{and} = \boldsymbol{\lambda b.\lambda c.(b\ c}\ \text{false})$$

$$\begin{aligned} \text{and true true} &= \text{true} \\ \text{and true false} &= \text{false} \\ \text{and false true} &= \text{false} \\ \text{and false false} &= \text{false} \end{aligned}$$

- Logical "or":

$$\text{or} = \boldsymbol{\lambda b.\lambda c.(b}\ \text{true}\ \boldsymbol{c})$$

$$\begin{aligned} \text{or true true} &= \text{true} \\ \text{or true false} &= \text{true} \\ \text{or false true} &= \text{true} \\ \text{or false false} &= \text{false} \end{aligned}$$

- Logical "not":

$$\text{not} = \boldsymbol{\lambda b.(b}\ \text{false true})$$

$$\begin{aligned} \text{not true} &= \text{false} \\ \text{not false} &= \text{true} \end{aligned}$$

Pairs

Using booleans, we can encode pairs of values.

$$\begin{array}{rcl} \text{pair } v\ w & : & \text{create a pair of } v \text{ and } w \\ \text{fst } p & : & \text{select the first component of } p \\ \text{snd } p & : & \text{select the second component of } p \end{array}$$

- Definition:

$$\begin{array}{rcl} \text{pair} & = & \boldsymbol{\lambda f.\lambda s.\lambda b.b\ f\ s} \\ \text{fst} & = & \boldsymbol{\lambda p.p}\ \text{true} \\ \text{snd} & = & \boldsymbol{\lambda p.p}\ \text{false} \end{array}$$

- Example:

$$\begin{array}{rcl} \text{fst (pair } v\ w) & = & \text{fst } (\boldsymbol{(\lambda f.\lambda s.\lambda b.b\ f\ s)\ v\ w}) \\ & = & \text{fst } \boldsymbol{(\lambda b.b\ v\ w)} \\ & = & \boldsymbol{(\lambda p.p}\ \text{true)}\ \boldsymbol{(\lambda b.b\ v\ w)} \\ & = & \boldsymbol{(\lambda b.b\ v\ w)}\ \text{true} \\ & = & \text{true}\ \boldsymbol{v\ w} \\ & = & \boldsymbol{v} \end{array}$$

Church Numerals

$$
\begin{aligned}
\underline{\mathbf{1}+\mathbf{2}} &= (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{\mathbf{1}}\ \underline{\mathbf{2}} \\
&= \lambda s.\lambda z.(\underline{\mathbf{2}}\ s\ (\underline{\mathbf{1}}\ s\ z)) \\
&= \lambda s.\lambda z.(\underline{\mathbf{2}}\ s\ (\lambda s.\lambda z.(s\ z)\ s\ z)) \\
&= \lambda s.\lambda z.(\underline{\mathbf{2}}\ s\ (s\ z)) \\
&= \lambda s.\lambda z.((\lambda s.\lambda z.(s\ (s\ z)))\ s\ (s\ z)) \\
&= \lambda s.\lambda z.(s\ (s\ (s\ z))) \\
&= \underline{\mathbf{3}}
\end{aligned}
$$

Church Numerals

- Multiplication:

$$\underline{E_1 \times E_2} = (\lambda m.\lambda n.m\ (\underline{+}\ n)\ \underline{0}\ \underline{E_1}\ \underline{E_2}$$

Example:

$$\begin{aligned}
\underline{1 \times 2} &= (\lambda m.\lambda n.m\ (\underline{+}\ n)\ \underline{0})\ \underline{1}\ \underline{2} \\
&= \underline{1}\ (\underline{+}\ \underline{2})\ \underline{0} \\
&= (\lambda s.\lambda z.\ s\ z)\ (\underline{+}\ \underline{2})\ \underline{0} \\
&= (\underline{+}\ \underline{2})\ \underline{0} \\
&= (\lambda m.\lambda s.\lambda z.m\ s\ (\underline{2}\ s\ z))\ \underline{0} \\
&= \lambda s.\lambda z.(\underline{0}\ s\ (\underline{2}\ s\ z)) \\
&= \lambda s.\lambda z.((\lambda s.\lambda z.z)\ s\ (\underline{2}\ s\ z)) \\
&= \lambda s.\lambda z.\underline{2}\ s\ z \\
&= \lambda s.\lambda z.((\lambda s.\lambda z.s\ (s\ z))\ s\ z) \\
&= \lambda s.\lambda z.s\ (s\ z) = \underline{2}
\end{aligned}$$

- Power (n^m):

$$\lambda m.\lambda n.m\ (\underline{\times}\ n)\ \underline{1}$$

Recursion

- For example, the factorial function

$$f(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

  is encoded by

$$\text{fact} = Y(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1))$$

  where Y is the Y-combinator (or fixed point combinator):

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

- Then, fact n computes $n!$.
- Recursive functions can be encoded by composing non-recursive functions!

Recursion

Let $F = \lambda f.\lambda n.$if $n = 0$ then 1 else $n * f(n-1)$ and $G = \lambda x.F(x\ x)$.

$$
\begin{aligned}
&\text{fact } 1 \\
&= (Y\ F)\ 1 \\
&= (\lambda f.((\lambda x.f(x\ x))(\lambda x.f(x\ x)))\ F)\ 1 \\
&= ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 1 \\
&= (G\ G)\ 1 \\
&= (F\ (G\ G))\ 1 \\
&= (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))\ 1 \\
&= \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (G\ G)(1-1)) \\
&= \text{if false then } 1 \text{ else } 1 * (G\ G)(1-1)) \\
&= 1 * (G\ G)(1-1) \\
&= 1 * (F\ (G\ G))(1-1) \\
&= 1 * (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))(1-1) \\
&= 1 * \text{if } (1-1) = 0 \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1) \\
&= 1 * 1
\end{aligned}
$$

Summary

Programming language = Lambda calculus + Syntactic sugars

- λ-calculus is a minimal programming language.
  - Syntax: $e \rightarrow x \mid \lambda x.e \mid e\ e$
  - Semantics: β-reduction
- Yet, λ-calculus is Turing-complete.

$$\begin{array}{rcl} E & \rightarrow & x \\ & \mid & \lambda x.E \\ & \mid & E\ E \end{array} \quad = \quad$$