Introduction to Lambda Calculus

By John Lång, 22 May 2018
“Yön maku,
kun linnut syöksyy mustina siipinä tähtiä päin;
vaan mikä on nimesi nimi,
tähtesi salainen luku ja numero?”

– “Arcana”, by CMX, from their album Discopolis
Map of the Universe

Mathematics

We are here

Computer Science
What Is a Function?

<table>
<thead>
<tr>
<th>Formula</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = 2x$</td>
<td>$f(x)$ yields $2x$</td>
</tr>
<tr>
<td>$g(x,y) = 5$</td>
<td>$g$ is a constant function</td>
</tr>
<tr>
<td>$h : X \rightarrow Y$</td>
<td>$h$ maps every $X$ to an $Y$</td>
</tr>
<tr>
<td>$x \mapsto 3$</td>
<td>anonymous constant map</td>
</tr>
<tr>
<td>$(a \circ b)(x) = a(b(x))$</td>
<td>composition of $a$ and $b$</td>
</tr>
</tbody>
</table>
Two Answers

Intensional

\[ x \leq y \]

GCD\((x, y)\)

\[ x = 0 \]

false

\((x, y) \leftarrow (y \% x, x)\)

true

return \(y\)

Extensional

\(f(x)\)

\(y\)

\(x\)

\(f[x]\)
Two Answers

Intensional

\[ x \leq y \]

\[ \text{GCD}(x,y) \]

false

\[ x = 0 \]

true

\[ (x, y) \leftarrow (y \% x, x) \]

return y

Extensional

\[ f(x) \]
My Answer

• According to Wiktionary:
  – The word “function” comes from a Latin word that means “performance” or “execution”.
  – The Latin word “calculus” means “a pebble or stone used for counting”.

• Rhetorical question: Does modern set theoretical mathematics interpret these concepts appropriately?
  – This presentation tries to honour the traditional interpretations.
Pros and Cons of Constructive Approach

Pros

● Compatible with empirical science and skepticism.
● Theory may also work in practice without extra work.
● Involves less ambiguity.
● Subset of classical toolset.

Cons

● Lack of certain symmetries, such as $p \lor \neg p$.
● Our culturally conditioned, Euclidean world view becomes challenged.
● Proofs may become lengthy.
Pros and Cons of Constructive Approach

Pros

- Compatible with empirical science and skepticism.
- Theory may also work in practice without extra work.
- Involves less ambiguity.
- Subset of classical toolset.

Cons

- Lack of certain symmetries, such as $p \lor \neg p$.
- Our culturally conditioned, Euclidean world view becomes challenged.
- Proofs may become lengthy.
Terminology

- **Functions** are abstract dependencies between objects.
- **Algorithms** are representations of functions in a formal system. A function may have multiple algorithms.
- **Strings** can represent functions as well as their arguments.
- **Computation** is the art of algorithmic string manipulation.
- A **model of computation** is a formal system that defines effectively computable functions in terms of algorithms.
An Appetizer: The MIU-System

- Douglas Hofstadter presented the following puzzle in his famous book “Gödel, Escher, Bach: an Eternal Golden Braid”:
- The MIU-system has the letters $M$, $I$, and $U$ in its alphabet.
- It has four production rules:
  1) $aI \rightarrow aIU$;  
  2) $M\alpha \rightarrow M\alpha\alpha$;  
  3) $aIII\beta \rightarrow aU\beta$; and  
  4) $aUU\beta \rightarrow a\beta$;  
- The question is: Is $MU$ derivable from $MI$?
Some Models of Computation

Legend:

CoC = Calculus of Constructions
CA = Cellular Automata
CL = Combinatory Logic
FC = Flow Charts
FT = FRACTRAN
LC = Lambda Calculus
μRF = Mu Recursive Functions
PA = Peano Arithmetic
PN = Petri Nets
PCS = Post Canonical Systems
TM = Turing Machines
URM = Unlimited Register Machines
Lambda Calculus?

- *Lambda Calculus* (LC) is the model of computation used in this presentation.
  - It is a system that expresses functions as strings of symbols.
- A couple of common misconceptions need to be addressed:
  - It’s ‘Lambda” (the Greek letter Λ, λ), not ‘lambada”. This is not a dance lesson!
  - ‘Calculus” refers to a logical proof calculus. These slides are not going to discuss integral or differential calculus.
Teaser: Building Functions

- As a teaser on LC, consider the expression $2x + c$.
  - It’s composed of $2x$ and $c$, glued together with the plus operation.
  - $2x$ can be seen as a function of $x$, or $(\lambda x.2x)$, and $c$ as a free variable.
  - In LC, every operator uses infix notation, so we get $+ (\lambda x.2x)\ c$.
  - We will plug in something to the left summand: $((\lambda x.2x)\ y)\ c$.

- A ‘$\lambda$’ creates an anonymous function with one named argument.
- $(\lambda x.2x)\ y$ means evaluating this function with the argument $y$. 
How to Define Algorithms?

- **Imperative** paradigm operates algorithms like cooking recipes, on step-by-step basis:

  0: IN R1, =KBD
  1: LOAD R2, =1
  2: JZER R1, 6
  3: MUL R2, R1
  4: SUB R1, =1
  5: JUMP 2
  6: OUT R2, =CRT
  7: SVC R0, =HALT

- **Declarative** paradigm defines algorithms as compositions of computational primitives:

  ```
  select name, email
  from students join staff
  where email like '%.fi'
  ```

- We will use a declarative model of computation.
Conventional Systems Create Challenges

- A “function” in C:

```
int c = 0;
int f(int i)
{
    return i + c++;
}
f(3); // Returns 3.
f(3); // Returns 4.
```

- Thus, $f(3) \neq f(3)$!

- Ambiguities in classical math:

  - $f(x) = y$
  - What does ‘=’ mean?
  - Definition or assertion?
  - Did we apply $f$ in $x$?
  - Perhaps we multiplied $f$ by $x$?

- Is $\circ$ in $(f \circ g)$ a function?
Lambda Calculus Provides Solutions

- Consider the pseudocode:
  
  ```c
  int f'(int c)(int i)
  {return i + (c + 1);}
  ```

- Let \( f'' \defeq f'(0) \), so

  ```c
  int f''(int i)
  {return i + (0 + 1);}
  ```

- Now, \( f''(3) = f''(3) \).

- The state of \( c \) is now explicit.

- In LC, \( f(x) = y \) means \( \text{“f applied to x equals y”} \).
  - It is an assertion (\( “==” \)), not a definition (\( “=” \)).
  - *No other interpretations.*
  - ‘\( = \)’ always has a direct proof.

- \( \circ \) is just another function:
  - \( f \circ g \equiv (\lambda vwx.v(wx))fg \).
The Black Box Analogy Revised

- Traditionally, functions are seen as black boxes
- In LC, functions are the stuff we’re playing with
Why Lambda Calculus?

- Lambda Calculus is capable of expressing dynamic behaviour using a referentially transparent formal string rewrite system.
- LC has an unambiguous syntax with very few special cases.
- The computer of LC doesn’t need soul, intuition, or magic.
- LC doesn’t burden us with boring and error-prone technical details, such as memory management or instruction sequencing.
- LC gives the best of both mathematics and computer science.
The Extent of Computational Universe

- Lambda Calculus, Combinatory Logic, Mu Recursive Functions, Turing machines, and others, represent the strongest class of models of computation. (Church-Turing Thesis)
- They’re powerful enough to feature undecidability, i.e. the incompleteness of computational universe. (Thanks, Gödel!)
  - There are strange loops between object and meta languages.
  - On the bright side, we can extend every formal system infinitely!
  - We’ll see later how undecidability emerges in LC.
A Very Small World Map

Logic → Sets → Types → Categories

Logic is defined by Complexity and extend Express by Sets Construct and Types are organized by compute.

Logic explains, equips, and proves Computability, which extends Complexity and exploit LC to create Programs that help us understand Algebra.

Sets models, organize, and populate Types, which builds Functions and inspire Categories.

LC composes, builds, and needs Functions, which helps Algebra and builds Programs.
Subway Map

- This slide was inspired by the West Metro, that finally connects Espoo and Helsinki. The stops are the concepts worth learning. Page number is written beneath the stop.
Before We Start

- Our **object language** is the language of lambda expressions.
  - The object language consists of variable and control symbols.

- Our **meta language** defines rules for working with lambda expressions.
  - Capital Latin letters (i.e. *meta variables*), indices, substitution notation, and various relational symbols belong to this language.

- Don’t get confused with the object and meta languages!
Zone 1: Strings

- \textit{Strings} are (finite) ordered sequences of symbols.
  - For example, “a string” is a string (of the English alphabet).
  - Delimiters ‘“’ and ‘”’ are notation and may (and usually will) be omitted, when there’s no risk of ambiguity.

- We refer to strings by identifying them with (meta) \textit{variables}.
  - $V \defeq \text{“value”}$ declares that $V$ refers to “value”; and
  - $V \defeq W$ declares that $V$ refers to the value that $W$ refers to.
  - A variable is \textit{interchangeable} with its value.
String Length and Equality

- The length of a string $V$, written as $|V|$, is the number of (possibly repeated) symbols in it.
  - For example, $|""| = 0$, $|"x"| = 1$, $|"xx"| = 2$, $|"string"| = 6$.

- Strings $V$ and $W$, are equal, written as $V \equiv W$, if and only if $|V| = |W|$ and they contain the same symbols in the same order.
  - For example, “cat” $\equiv$ “cat”, but “cat” $\not\equiv$ “tac” and “cat” $\not\equiv$ “catch”.
  - (String equality satisfies reflexivity, symmetry, transitivity and substitution property, so it can be seen as an equivalence relation.)
String Catenation and Substrings

- \( V \star W \) or \( VW \) is a string, called \((\text{con})\text{catenation}\) of \( V \) and \( W \). It contains every symbol of first \( V \), then \( W \).
  - \( (U \star V) \star W \equiv U \star (V \star W) \) and \( |U \star V| = |U| + |V| \).
  - E.g. “bat”\*“man” \( \equiv \) “batman”, but “bat”\*“man” \( \neq \) “man”\*“bat”.

- \( V \) is a substring of \( W \) if and only if \( |V| \leq |W| \) and \( V \) contains the same symbols as \( W \) in the same order, until the end of \( V \).
  - E.g. \( V \) and \( W \) are substrings of \( VW \) (\( \equiv \) \( V \star W \)); “at” and “cat” are substrings of “cat”; “Fat” is not a substring of “cat”.
Alphabet of Lambda Calculus

- The objects of study in LC are (non-empty) strings known as *lambda expressions* (\(\lambda\)-exprs, a.k.a. *lambda terms*). Their alphabet contains:

  1) ‘\(\lambda\)’ (lambda)
  2) ‘.’ (dot)
  3) ‘(’ (left parenthesis)
  4) ‘)’ (right parenthesis)
  5) \(x_0, x_1, x_2, \ldots\) (variable symbols)
Lambda Expressions

• Let \( x \) be a variable symbol and \( M, N \) \( \lambda \)-exprs. Then:
  1) \( x \) is a lambda expression (called \textit{variable});
  2) \( (\lambda x. M) \) is a lambda expression (called \textit{abstraction});
  3) \( (MN) \) is a lambda expression (called \textit{application}); and
  4) nothing else is a lambda expression.

• (Technically, we have “\( x \)”, “(\( \lambda x.\) \( \star M\)\)””, and “(”\( \star M\) \( \star N\)\)”.”)

• The definitions above may be applied \textit{recursively}.

Alternative Definition

- We can also use Backus-Naur Form. Let

\[
\begin{align*}
\text{<λ-expr>} & ::= \text{<λ-var>} \mid \text{<λ-abstr>} \mid \text{<λ-app>}
\text{<λ-var>} & ::= \text{“} ( \text{<λ-symbol>} \text{“})
\text{<λ-abstr>} & ::= \text{“} (λ \text{<λ-symbol>} \text{“}. \text{<λ-expr>} \text{“})
\text{<λ-app>} & ::= \text{“} (\text{<λ-expr>} \text{<λ-expr>} \text{“})
\end{align*}
\]

where <λ-symbol> can be replaced with an appropriate variable symbol from the alphabet.

- We can construct every (finitary) λ-expr be applying these rules finitely many times.
Syntactic Sugar

- It shall be declared that:
  1) parentheses may be omitted if there’s no risk of ambiguity;
  2) abstraction binds as far to the right as possible;
  3) $MNO \overset{\text{def}}{=} (MN)O$, so application is left-associative;
  4) $\lambda x.\lambda y.M \overset{\text{def}}{=} \lambda x.(\lambda y.M)$, so abstraction is right-associative; and
  5) $\lambda xy.M \overset{\text{def}}{=} \lambda x.\lambda y.M$ (in general, $\lambda x_0 x_1 x_2 \ldots x_n.M \overset{\text{def}}{=} \lambda x_0.\lambda x_1.\lambda x_2 \ldots \lambda x_n.M$).

- For example, $(\lambda x.x)y$ is different from $\lambda x.xy$ (i.e. $\lambda x.(xy)$).
Subexpressions

• In these slides, small Latin letters stand for lambda variables and capital letters for arbitrary expressions (including lambda variables), even when it’s not explicitly stated.

• A substring of a $\lambda$-expr, that is also a valid $\lambda$-expr, is a subexpression. For example:
  - $x$ is a subexpression of itself, $(\lambda x. x)$, and $(xy)$; and
  - $(\lambda x. z)$ is a not a subexpression of $x$, $(xy)$, or $(z(\lambda x. y)wv)$.
  - $\lambda x$ is not a $\lambda$-expr and thus not a subexpression of any $\lambda$-expr.
A Corner Case

- Consider $\lambda xz.xz$. One might think that it has $\lambda xz.x$ or $\lambda z.x$ as a subexpression...
- But it doesn’t! Let’s unroll the syntactic sugar:
  
  $\lambda xz.xz \equiv \lambda x.\lambda z.xz$
  
  $\equiv (\lambda x.(\lambda z.(xz)))$

- The subexpressions of $\lambda xz.xz$ really are $\lambda xz.xz$, $\lambda z.xz$, $xz$, $x$, and $z$.
- The lesson: *The definition applies to de-sugared expressions.*
Tree Analogue

- Lambda expressions can be combined into larger expressions, using the syntax rules given in p. 28 and 29.
- The structure of a lambda expression can be visualized with a parse tree.
How To Read Parse Trees

- Colours identify variables.
- Trees can be nested, from up to down.

\[\lambda x. M\]

\[MN\]

\[\alpha M\beta\]

\[y\]
The Syntax of LC Lacks Symmetry

- Remember, that
  - $(MN)O \neq M(NO)$
  - $MN \neq NM$ (unless $M \equiv N$)
  - $(\lambda x.x)y \neq \lambda x.(xy)$

- Cf. function composition:
  - Usually, $-(x^2) \neq (-x)^2$
  - hence, $-\circ^2 \neq 2\circ$ –

- No associativity, commutativity, or distributivity in LC!
The Other Calculus

- Here’s another tree representing the process of finding the derivative of a polynomial function.
- No limits needed!
- The term “Lambda Calculus” is no coincidence!
Did You Get The Syntax?

Are the following strings are λ-exprs? (Why?)

- \( \lambda x \)
- \( (\lambda x.x) \)
- \( zx(\lambda x.y) \)
- \( \lambda xy.z \)
- \( (\lambda x.x)(\lambda x.x) \)
- \( c \)
- \( (c) \)
Did You Get The Syntax?

Are the following strings are λ-exprs? (Why?)

- \( \lambda x \)  
  No. There’s no axiom for this.

- \((\lambda x.x)\)  
  No. The right parenthesis is missing.

- \(zx(\lambda x.y)\)  
  Yes. A stupid one, if you ask me...

- \(\lambda xy.z\)  
  Yes. Stands for \((\lambda x.\lambda y.z)\).

- \((\lambda x.x)(\lambda x.x)\)  
  Yes, it’s \((\lambda x.x)\) applied to itself.

- \(c\)  
  Of course. (A variable expression.)

- \((c)\)  
  Yes, it’s the same as \(c\) above.
Are These Expressions Identical?

- Are the following pairs of lambda expressions identical?

- \((x)\) and \(x\)

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

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

- \(wx(yz)\) and \((wx)(yz)\)

- \(\lambda x.(\lambda y.x)z\) and \(\lambda x.\lambda y.xz\)

- \(\lambda wx.(\lambda y.z)\) and \(\lambda wxy.z\)

- \(\lambda xy.z\) and \(\lambda x.yz\)
Are These Expressions Identical?

- Are the following pairs of lambda expressions identical?
  - \((x)\) and \(x\)  
    Yes. Outer parentheses omitted.
  - \(z(\lambda x.y)\) and \((\lambda x.y)z\)  
    No, there’s no commutativity.
  - \(x(\lambda x.y)z\) and \((x)((\lambda x.y)z)\)  
    No. Application is left-associative.
  - \(wx(yz)\) and \((wx)(yz)\)  
    Yes. Application is left-associative.
  - \(\lambda x.(\lambda y.x)z\) and \(\lambda x.\lambda y.xz\)  
    No. \((\lambda y.x)z \neq \lambda y.xz\).
  - \(\lambda wx.(\lambda y.z)\) and \(\lambda wxy.z\)  
    Yes.
  - \(\lambda xy.z\) and \(\lambda x.yz\)  
    No. \(\lambda xy.z \equiv \lambda x.\lambda y.z \neq \lambda x.yz\)
From Lambdas to Calculus

- So far, we are only familiar with strings and lambda expressions. We don’t have the “Calculus” part yet.
- The idea in LC is to show that the (extensional) behaviour of functions emerges from syntactic (intensional) foundations.
- At this point, the only notion of equivalence between lambda expressions we have, is string equality.
- In order to construct a more useful notion of equality, we’re going to need more definitions...
Zone 2 – Next Stop: Free Variables

- Now that we are familiar with the object language, it’s time to move on to Zone 2, where we learn how to relate \( \lambda \)-exprs to each other.
There are free, binding, and bound variables. Intuitively:

- **Bound variables** occur as free variables in subexpressions of lambda abstractions. For example, $x$ is bound in $\lambda x.xx$ but not in $\lambda x.y$.
- **Binding variables** are bound and prefixed with lambdas, like $x$ in $\lambda x.y$.
- **Free variables** are variables not bound, such as $x$ in $zxy$ or $(\lambda y.y)x$.
- The subexpressions of applications are handled separately.

**A variable can be both free and bound, or neither.**

- Depends on context. For example, $x(\lambda x.x)$ is free, binding and bound.
Free and Bound Variables, Formal Definition

• The free variables in a λ-expr, using the language of sets:
  1) \( \text{free}(x) \) \( \overset{\text{def}}{=} \{ x \} \)
  2) \( \text{free}(\lambda x. M) \) \( \overset{\text{def}}{=} \text{free}(M) \setminus \{ x \} \)
  3) \( \text{free}(MN) \) \( \overset{\text{def}}{=} \text{free}(M) \cup \text{free}(N) \)

• The bound variables, on the other hand, can be defined as:
  1) \( \text{bound}(x) \) \( \overset{\text{def}}{=} \{ \} \) (i.e. \( \emptyset \))
  2) \( \text{bound}(\lambda x. M) \) \( \overset{\text{def}}{=} \text{bound}(M) \cup \{ x \} \)
  3) \( \text{bound}(MN) \) \( \overset{\text{def}}{=} \text{bound}(M) \cup \text{bound}(N) \)
Example on Free Variables

\[
\text{free}(\lambda y. (\lambda x. zy) w) = \text{free}((\lambda x. zy) w) \setminus \{y\} \\
= (\text{free}(\lambda x. zy) \cup \text{free}(w)) \setminus \{y\} \\
= (((\text{free}(zy) \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
= (((\text{free}(z) \cup \text{free}(y)) \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
= (((\{z\} \cup \{y\}) \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
= ((\{z, y\}) \setminus \{x\}) \cup \{w\} \setminus \{y\} \\
= (\{z, y\} \cup \{w\}) \setminus \{y\} \\
= \{z, y, w\} \setminus \{y\} \\
= \{z, w\}.
\]
Real World Examples on Variable Binding

• Consider the following examples:
  - $\exists x. (\emptyset \in x) \land (\forall y. (y \in x \rightarrow \{ y, \{ y \} \} \in x))$
  - $z + (\sum_{k \in \mathbb{N} \setminus \{0\}} 6k^{-2}) = z + \pi^2$
  - `int f(int i) {return i + c++;}`

• Variables $x$, $y$, $k$, and $i$ are bound. They are fixed quantities.
• Variables $z$, $\pi$, and $c$ are not bound. Their meaning depends largely on the context in which they are interpreted in.
Free and Bound Variables Quiz

Which variables in the following expressions are free/bound/both/neither on outermost context?

- \( \lambda x.x \)
- \( \lambda x.xx \)
- \( \lambda x.xy \)
- \((\lambda x.y)(\lambda y.x)\)
- \(\lambda x.\lambda y.xy\)
- \(\lambda x.x(\lambda y.y)\)
- \(x(\lambda y.y)yz\)
Which variables in the following expressions are free/bound/both/neither on outermost context?

- $\lambda x. x$  
  $x$ is bound and not free. Easy, wasn’t it?

- $\lambda x. xx$  
  $x$ is bound, not free.

- $\lambda x. xy$  
  $x$ is bound, not free. $y$ is free and not bound.

- $(\lambda x. y)(\lambda y. x)$
  Both $x$ and $y$ have free and bound occurences.

- $\lambda x. \lambda y. xy$
  Both $x$ and $y$ are bound and not free.

- $\lambda x. x(\lambda y. y)$
  Both $x$ and $y$ are bound and not free.

- $x(\lambda y. y)yz$
  $x$ and $z$ are only free. $y$ is both free and bound.
Substitution of Expressions

- \([x=N]M\) denotes “the result of replacing every free occurrence of \(x\) with \(N\) in \(M\), provided it’s legal to do so; otherwise \(M\)”.

- The formal definition of substitution is technical. Intuitively:
  1) The result must be a legal lambda expression;
  2) free variables must remain free;
  3) bound variables must remain bound; and
  4) the differences/equalities between variables must be preserved.
Substitution Flowchart

- $\text{true}$: $\text{false}$
- $\text{false}$: $\text{true}$

$x \in \text{free}(M)$

$x \in \text{free}(N)$

$M \equiv x$

$M \equiv \lambda y.0$

$\text{return }\{x:=N\}M$

$\text{return }\{x:=N\}O(\{x:=N\}P)$

$\text{return }\{x:=N\}P$

$\text{return }\lambda y.\{x:=N\}O$

$\text{return }\{x:=N\}P$

$\text{return }\{x:=N\}O$

$\text{return }\{x:=N\}P$

$\text{return }\lambda y.\{x:=N\}O$
Perils of Careless Substitution

• Substitution is non-trivial business because of the risk of variable capture.

• *Variable capture* is a violation of the rules 2–4 in p. 49.

• For example, if \([x:=y](\lambda x.y)\) yielded \(\lambda y.y\), the free variable \(y\) would become bound.
  
  − In this kind of situation substitution yields \(\lambda x.y\).
  
  − Other examples include \([y:=x](\lambda x.y)\), \([x:=z](\lambda x.yz)\), and \([y:=z](\lambda x.yz)\).
Substitution Quiz

What is the result of these substitutions? (Why?)

- \([x=y]y\)
- \([x=y]x\)
- \([x=y](xy)\)
- \([x=y](\lambda y.x)\)
- \([x=y](\lambda y.y)\)
- \([x=(\lambda z.z)](\lambda y.x)\)
- \([x=(\lambda x.x)](\lambda y.x)\)
Substitution Quiz

What is the result of these substitutions? (Why?)

- $[x=y]y$  
  $y$. There’s no $x$ to be replaced.

- $[x=y]x$  
  $y$. This is a basic substitution.

- $[x=y](xy)$  
  $xy$. Their difference must be preserved.

- $[x=y](\lambda y.x)$  
  $\lambda y.x$. Variable identity must be preserved.

- $[x=y](\lambda y.y)$  
  $\lambda y.y$. No $x$ present.

- $[x=(\lambda z.z)](\lambda y.x)$  
  $\lambda y.\lambda z.z$. No variable capture here.

- $[x=(\lambda x.x)](\lambda y.x)$  
  $\lambda y.x$. Variable $x$ would become bound.
Conversion and Reduction

- **Conversion** is a two-way transformation (i.e. a function in the meta language) from one \( \lambda \)-expr to another.
  - Wrt. free and bound variables, we want to identify \( x \) with \( y \), as well as \( \lambda x.x \) with \( \lambda y.y \), \( \lambda x.y \) with \( \lambda y.x \), and \( \lambda x.MxN \) with \( \lambda y.MyN \), etc.

- **Reduction** is the concept used in LC to bring \( \lambda \)-exprs into life as substitution instructions.
  - \( \lambda \)-exprs also act as *data*.
  - Cf. memory words in a von Neumann Machine or sets in Set Theory.
Alpha Conversion

• A $\lambda$-expr $M \equiv \lambda x. N$ is alpha congruent with $M' \equiv \lambda y. [x:=y]N$ (i.e. $M$ with $x$ renamed to $y$), if and only if $y$ is not free in $N$.
  - The assertion of alpha congruence is denoted with $M \equiv_{\alpha} M'$.
  - We also consider expressions having congruent subexpressions to be congruent, i.e. if $M \equiv_{\alpha} M'$, then $aM\beta \equiv_{\alpha} aM'\beta$. ($a$ or $\beta$ may be empty.)

• It is customary to focus on $\lambda$-exprs modulo alpha convertibility (i.e. as representatives of equivalence classes of $\equiv_{\alpha}$), meaning that if $M \equiv_{\alpha} N$, then we overload ‘$\equiv$’ with $M \equiv N$. We do so.
“Modulo Convertibility”?

- Consider parse trees for $w(\lambda yz.x(yz))$ and $w(\lambda gx.f(gx))$.
- They’re essentially the same tree, with different labels.
- Thus, we will focus on their structure instead of concrete typography.
How to Avoid Variable Capture

- **Wrong** \([z := x](\lambda x. xyz)\):
  - Direct substitution.

- **Correct** \([z := x](\lambda x. xyz)\):
  - Rename \(x\); Then substitute.
Can You Convert These?

- Are the following pairs of expressions alpha congruent?
  - $\lambda x.x$ and $\lambda y.y$
  - $\lambda x.x$ and $\lambda x.y$
  - $\lambda x.x$ and $\lambda y.x$
  - $\lambda xy.x$ and $\lambda yx.x$
  - $\lambda xy.x$ and $\lambda yx.y$
  - $\lambda xy.xy$ and $\lambda yz.zy$
  - $\lambda xy.xy$ and $\lambda yx.yx$
Can You Convert These?

• Are the following pairs of expressions alpha congruent?
  
  - $$\lambda x.x$$ and $$\lambda y.y$$  
    Yes. $$\lambda y.y$$ is the same as $$\lambda x.[x=y]x.$$  
  
  - $$\lambda x.x$$ and $$\lambda x.y$$  
    No. Different variable bindings.  
  
  - $$\lambda x.x$$ and $$\lambda y.x$$  
    No. Different variable bindings.  
  
  - $$\lambda xy.x$$ and $$\lambda yx.x$$  
    No. $$\lambda xy.x \equiv \lambda x.\lambda y.x \neq \lambda y.\lambda x.x \equiv \lambda yx.x.$$  
  
  - $$\lambda xy.x$$ and $$\lambda yx.y$$  
    Yes. $$\lambda xy.x \equiv \lambda xz.x \equiv \lambda yz.y \equiv \lambda yx.y.$$  
  
  - $$\lambda xy.xy$$ and $$\lambda yz.zy$$  
    Yes. $$\lambda xy.xy \equiv \lambda x.\lambda y.xy \equiv \lambda z.\lambda y.zy \equiv \lambda yz.zy.$$  
  
  - $$\lambda xy.xy$$ and $$\lambda yx.yx$$  
    Yes. $$\lambda xy.xy \equiv \lambda xz.xz \equiv \lambda yz.yz \equiv \lambda yx.yx.$$  

Beta Reducible Expressions

- From now on, we’re going to be naïve. We will only work with expressions for which substitution makes sense.
- A lambda expression is *beta reducible expression* (β-redex) if (and only if) it’s of the following form:
  - \((\lambda x. M) N\) (where \(x\) is a variable symbol and \(M, N\) are \(\lambda\)-expr).
- A \(\lambda\)-expr is in *β-Normal Form* (β-NF) if and only if it doesn’t contain any β-redexes. (Remember the parse tree in p. 33?)
Beta Reduction

- Let $M \equiv (\lambda x. N) O$ and $M' \equiv [x := O] N$ be expressions. The rule of (beta) reduction says that:
  - redex $M$ reduces to reduct $M'$. This is denoted with $M \rightarrow M'$.
  - We also say that $aM\beta$ to reduces into $aM'\beta$. ($a$ or $\beta$ can be empty.)

- This means, that the computer has to:
  - Take $(\lambda x. N) O$. (A $\beta$-reducible expression.)
  - Give $[x := O] N$. (Drop “$\lambda x.$”; Substitute free ‘$x$’s with $O$s.)
Reduction Flowchart

\( \text{(1) } M \equiv x \)

- Reduction
- True: Return \( x \)
- False: Return \([x \equiv O]N\)

\( \text{(2) } M \equiv (\lambda x. N) O \)

- Reduction
- True: Return \( (\text{reduce } N) \ast (\text{reduce } O) \)
- False: Return \([x \equiv O]N\)

\( \text{(3) } M \equiv \lambda x. N \)

- Reduction
- True: Return \( (\text{reduce } N) \ast (\text{reduce } O) \)
- False: Return \( \lambda x. N \)

\( \text{(4) } M \equiv NO \)

- Reduction
- True: Return \( (\text{reduce } N) \ast (\text{reduce } O) \)
- False: Return \( NO \)
The Fun Is About to Begin

• Performing a single step of reduction is like performing one arithmetic operation, or one step of logical inference.
  – Usually, we need many steps.

• Reducing complex expressions comes down to repeatedly applying beta reduction until a β-NF is reached.

• If $M$ reduces to $N$ in zero or more steps, we write $M \rightarrow\!\!\!\!\!\!\!\!\!\rightarrow N$.
  – ($\rightarrow\!\!\!\!\!\!\!\!\!\rightarrow$ is the transitive-reflexive closure of $\rightarrow$.)


Graph(ical) Example

\[(\lambda x.\lambda y.x)(\lambda z.z) MN\]

- \(\lambda x.\lambda y.x\) is an abstraction.
- \(\lambda z.z\) is a variable.
- Leftmost application:
- Redex: \(\lambda z.z\)
“\( \lambda x. \)" was removed and \( \lambda z.z \) replaced \( x \).

In programming jargon, the **formal parameter** \( x \) was **evaluated** with the **actual parameter** \( \lambda z.z \).
Next, “λy.” was removed.

Because \( y \) was not free in the subexpression \( \lambda z. z \), the applicand \( M \) was thrown away during the reduction process.

One more step to go.
Graph(ical) Example

- So, $(\lambda x.\lambda y.x)(\lambda z.z) MN \rightarrow N.$
- The result didn’t depend on $M$ or $N$ in any way, because LC is referentially transparent.
  - Cf. the C language example in p. 16.
Formal Example

• Let’s consider $M \overset{\text{def}}{=} (\lambda w.v)xy$. It holds that $M \rightarrow x$:

\[
M \equiv (\lambda w.v)xy \\
\equiv ((\lambda v.\lambda w.v)x)y \\
\rightarrow ([v:=x]\lambda w.v)y \\
\equiv (\lambda w.x)y \\
\rightarrow [w:=y]x \\
\equiv x
\]

• Thus, $M \rightarrow x$ (in two steps).
Numerical Example

- We can define these things in a way that makes even machines able operate on them.
  - The trick is called recursion.

\[
(\lambda x.\lambda y.2 \cdot x + x \cdot y - 3) \quad 5 \quad 7
\]  
\[
\rightarrow (\lambda y.2 \cdot 5 + 5 \cdot y - 3) \quad 7
\]  
\[
\rightarrow (\lambda y.10 + 5 \cdot y - 3) \quad 7
\]  
\[
\rightarrow 10 + 5 \cdot 7 - 3
\]  
\[
\rightarrow 10 + 35 - 3
\]  
\[
\rightarrow 10 + 32
\]  
\[
\rightarrow 42
\]
• The graph to the right shows all the possible reducts of one particular \( \lambda \)-expr.
• Note that every path ends to the same normal form \( (\lambda v. v) \).
  - This is not a coincidence!
  - (It follows from the Church-Rosser Theorem.)
Algebraic Analogy

\[(1+2) \cdot (3-4)\]

\[\frac{1}{3-4} + 2 \cdot (3-4) \cdot (1+2) \cdot (-1) \cdot 3 \cdot (-1)\]

\[3-4+2 \cdot (3-4) \quad 1 \cdot (-1)+2 \cdot (3-4) \quad 1 \cdot (3-4)+2 \cdot (-1)\]

\[1 \cdot (3-4)+6-8 \quad 3 \cdot (-1)\]

\[(1+2) \cdot 3-(1+2)\cdot 4\]
Reduction Quiz, Part I

1) The β-NF of \((\lambda x.xx)y\) is...
   a) \(xx\)
   b) \(yy\)
   c) neither

2) The β-NF of \(w(\lambda x.xz)y\) is...
   a) the expression itself
   b) \(wyz\)
   c) \(wzy\)
Reduction Quiz, Part I

1) The β-NF of \((\lambda x.xx)y\) is...
   a) \(xx\)
   b) \(yy\)
   c) neither

By definition,

\[(\lambda x.xx)y \rightarrow [x:=y]xx \equiv yy,\]

so we throw away “\(\lambda x.\)” and substitute both \(xs\) with \(y\).

2) The β-NF of \(w(\lambda x.xz)y\) is...
   a) the expression itself
   b) \(wyz\)
   c) \(wzy\)

\(w(\lambda x.xz)y \equiv (w(\lambda x.xz))y\), so \(\lambda x.xz\) cannot be applied to \(y\). On the other hand, \(w\) is just a variable. Thus, the expression is in β-NF.
3) What is the β-NF of
\((\lambda w. w)(\lambda x. z)(((\lambda x. x) y)(\lambda x. x z)y w)\)?
   a) \((\lambda w. w) z\)
   b) \((\lambda x. z)(((\lambda x. x) y)(\lambda x. x z)y w)\)
   c) neither

4) The β-NF of \(v(\lambda x. z)\) is...
   a) the expression itself
   b) \(z\)
   c) neither
Reduction Quiz, Part II

3) What is the β-NF of

$$(\lambda w. w)(\lambda x. z)((\lambda x. x)y)(\lambda x. xz)yw)?$$

a) $$(\lambda w. w)z$$
b) $$(\lambda x. z)(((\lambda x. x)y)(\lambda x. xz)yw)$$
c) neither

The first two functions from the left are identity and a constant function, so we get $z$ in 2 steps.

4) The β-NF of $v(\lambda x. z)$ is...

a) the expression itself
b) $z$
c) neither

$v(\lambda x. z)$ is in normal form, so it cannot be reduced to anything else. N.B. (b) is wrong, because the operations don’t commute.
5) The $\beta$-NF of $w(x(\lambda y.wz))$ is...
   a) $w(x(\lambda x.wz))$
b) $w(x(\lambda w.wz))$
c) $w(x(\lambda z.wz))$

6) The $\beta$-NF of $(\lambda x.xx)(\lambda x.xx)$ is...
   a) the expression itself
   b) $(\lambda x.xx)$
c) neither
Reduction Quiz, Part III

5) The $\beta$-NF of $w(x(\lambda y. wz))$ is...
   a) $w(x(\lambda x. wz))$
   b) $w(x(\lambda w. wz))$
   c) $w(x(\lambda z. wz))$

   $w(x(\lambda y. wz)) \equiv_\alpha w(x(\lambda x. wz))$, so we consider them identical. There’s no variable capture, since $x, y$ are not free in $wz$.

6) The $\beta$-NF of $(\lambda x. xx)(\lambda x. xx)$ is...
   a) the expression itself
   b) $(\lambda x. xx)$
   c) neither

   $(\lambda x. xx)(\lambda x. xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$. Uh, oh! This expression reduces itself, so it’s a $\beta$-redex that cannot be reduced!
A Quick Recapitulation

- Let’s have a short recap on the (meta) notation:
  - $v \overset{\text{def}}{=} w$ defines $v$ to refer to the same string as $w$ does;
  - $v \overset{\text{def}}{=} \text{“wzq”}$ defines $v$ to be a reference to the string literal “wzq”;
  - $[x:=N]M$ the λ-expr obtained from $M$ by substituting $x$ with $N$;
  - $M \equiv N$ asserts that $M$ and $N$ refer to (alpha) congruent λ-exprs;
  - $M \rightarrow N$ asserts that $M$ reduces to $N$ in single step; and
  - $M \rightarrow^* N$ asserts that $M$ reduces to $N$ in any number of steps.
Equality of Lambda Expressions

- We’re finally ready to formulate what is perhaps the main question in LC: *Equality* of lambda expressions.

- Given any two λ-exprs $M$ and $N$, if
  1) $M \equiv N$,  
  2) $M \equiv^\alpha N$, or  
  3) $M \rightarrow N$ or $N \rightarrow M$,
then $M$ and $N$ are said to be *equal*, denoted with $M = N$. 

Example Equation

- Consider \((\lambda x y. y) z\) and \((\lambda x. x)(\lambda w. w)\).
  1) \((\lambda x y. y) z \rightarrow (\lambda y. y)\), so \((\lambda x y. y) z = (\lambda y. y)\)
  2) \((\lambda y. y) \equiv_\alpha (\lambda w. w)\), so \((\lambda y. y) = (\lambda w. w)\)
  3) \((\lambda x. x)(\lambda w. w) \rightarrow (\lambda w. w)\), so \((\lambda w. w) = (\lambda x. x)(\lambda w. w)\)
  4) By transitivity (twice), \((\lambda x y. y) z = (\lambda x. x)(\lambda w. w)\).
- However, \((\lambda x y. y) \neq (\lambda x. x)\) and \(z \neq (\lambda w. w)\).
- Thus, equal expressions might have non-equal subexpressions.
Zone 3 - Semantics

- Now that we are familiar with the basic tools used for working with λ-exprs, it’s time to see how they can be used.
Finding the β-NF is analogous to calculating the value of a function or a computer program:

1) If \( f \) is a (possibly constant) function (or program) of \( x \); and
2) its value can be interpreted as a λ-expr \( M \); then
3) \( f(x) \approx \lambda x. M \) (“take \( x \); give \( M \)”); and
4) \( f(c) \approx (\lambda x. M)c \) (“apply \( f \) to \( c \)”); so
5) \( \lambda x. M \) is like a program, which transforms each \( c \) into \([x:=c]M\); and
6) if \([x:=c]M\) is in β-NF, the program has halted with output \([x:=c]M\).
Referential Transparency

- A very important feature of LC is **compositionality**, a.k.a. **referential transparency** in computer science.
  - The meaning of a lambda expression is entirely determined by the meanings of its subexpressions.
  - “The whole is a sum of its parts.”
  - $M = N$, if and only if $aM\beta = aN\beta$. This is *always* true in LC.

- Compositionality is taken as granted in mathematics, but only the most elite functional programming languages can deliver it.
Confluence of Beta Reduction

- Church-Rosser Theorem is perhaps the central result in LC:
  - If $M \rightarrow N$ and $M \rightarrow N'$, then there is $O$ s.t. $N \rightarrow O$ and $N' \rightarrow O$.
- This means that if a normal form exists, it is unique and reachable through (iterated) reduction.
  - For example, consider the graphs in p. 70–71.
  - If the NF is reachable, then the order of reductions is irrelevant to the outcome.
  - Cf. $(1+2) \cdot (3-4) = 3 \cdot (3-4) = (1+2) \cdot (-1) = 3 \cdot (-1) = -3$. 
Confluence as Diagrams

\[
\begin{align*}
N & \quad M \\
N' & \quad N
\end{align*}
\implies
\left(\begin{array}{ccc}
N & M & N' \\
O & N & N'
\end{array}\right)
\land
\begin{align*}
M & \quad M \\
\iff & \quad \iff \\
N & \quad = & \quad = \\
O & \quad N' & \quad O
\end{align*}
\]
Side Note: Extensionality

- \((\lambda xy. xy) \neq (\lambda z. z)\), even though \((\lambda xy. xy) MN = MN = (\lambda z. z) MN\) for any \(M\) and \(N\).
- **Equal structure implies same behaviour, but not vice versa.**
- Another aspect of equality is called **extensionality**.
  - The rule of \(\eta\)-conversion is that \(\lambda x. Mx =_{\eta} M\) when \(x\) is not free in \(M\).
  - Using this rule, we see that \(\lambda xy. xy \equiv \lambda x. \lambda y. xy =_{\eta} \lambda x. x \equiv_{\alpha} \lambda z. z\).
- We don’t need extensionality for this presentation though.
Connection Between Computability And Logic

- LC can be seen as a functional programming language.
  - It can be used for developing and analyzing algorithms.
  - It can be used as the foundational basis for more sophisticated and practical programming languages (e.g. Haskell, Agda, Idris, etc.).

- LC can be also seen as a formal proof system.
  - The equivalence of computer programs and logical proofs is a deep mathematical fact, known as the Curry-Howard correspondence.
  - However, “truth” is not a concept in LC (but provability is).
Logical Example

• We’ll define if-then-else soon, but let’s use intuition for now. Our example is the famous Aristotelian syllogism.

• We know that every man is mortal, so
  \[ P \overset{\text{def}}{=} (\lambda x. \text{if } (\text{Man } x) \text{ then } (\text{Mortal } x) \text{ else } \bot). \]

• By assumption, Socrates is a man, i.e. \textbf{Man Socrates} holds.

• Thus, \( P(\text{Socrates}) \rightarrow (\text{Mortal Socrates}) \), so \textit{reduction is like deduction using the modus ponens rule}. 

Did That Even Make Sense?

- The key difference between LC and predicate logic is that in LC there’s no notion of truth.
  - Instead, LC investigates definability, provability, and solvability.
- Also, λ-exprs don’t quite seem like the same kind of functions than those encountered in logic or set theory.
- Actually, LC does have models that make the connection to set theory clear, but they require rather advanced mathematics that is beyond our scope. Domain theory studies these models.
The Notion of Consistency in LC

- For the logicians among the audience, here’s the idea of consistency in LC:
  - Two expressions, $M$ and $N$ are incompatible, denoted with $M \not= N$, if and only if it is possible to derive an arbitrary equation from $M = N$.
  - A theory, i.e. an assortment of equations is consistent if and only if it doesn’t contain an equation $M = N$ such that $M \not= N$.
  - Such an equation would collapse the universe into a singleton.
  - We’ll see an example soon.
Standard Combinators

- **Combinator** is a λ-expr without free variables. For example:
  - \( I \triangleq \lambda x.x. \) (Thus, \( IM \to M. \))
  - \( K \triangleq \lambda xy.x. \) (Thus, \( KMN \to M. \))
  - \( S \triangleq \lambda xyz.xz(yz). \) (Thus, \( SMNO \to MO(NO). \))
- Actually, \( S \) and \( K \) are sufficient for expressing all combinators.
  - There is even a single combinator \( X \) that can express both \( S \) and \( K \)!
  - Likewise, there is a single-instruction Turing-complete computer!
Using Combinators to Express Others

- **SKK** ≡ (λxyz.xz(yz))KK
  → (λyz.Kz(yz))K
  → (λz.Kz(Kz))
  → λz.z
  ≡ I

- Because **SKK** → I (in 4 steps), it holds that **SKK** = I. ■

- (Also, **SKK**M → M ← IM for any M, but deducing the equality from this fact would require extensionality. See p. 87.)
Observations On the Proof

• In the previous slide, $S$ was \textit{partially applied} (i.e. lacked some of its defined argument(s)), so we needed to recall its definition.

• The instance of $K$ that was reduced, was \textit{fully applied}, so we could treat it as a black box, using the fact $KMN \rightarrow M$.
  - The second instance of $(Kz)$ was discarded completely!

• These kind of situations are common in LC.
  - This has profound implications in functional programming!
An Inconsistent Theory

• Suppose that $K = S$. For an arbitrary $\lambda$-expr $M$ we have

$KI(KM)I = SI(KM)I \rightarrow II(KMI) \rightarrow M$, so $M = KI(KM)I$.

• On the other hand, $KI(KM)I = I$. By transitivity $M = I$.

• The previous steps can be repeated for another arbitrary $\lambda$-expr $M'$, yielding $M' = I$. By symmetry and transitivity, $M = M'$.

• Therefore, $K \# S$, so the theory $\{K = S\}$ is inconsistent. ■
Side Note: The Notion of Definedness

- A combinator $M$ is **solvable** if and only if there are expressions $N_0, N_1, \ldots, N_k$ such that $M \ N_0 \ N_1 \ldots \ N_k = \ I$

- Unsolvable expressions can be safely identified. The symbol `bottom`, ‘$\perp$’ is sometimes used to represent an unsolvable expression, or **undefined** value. E.g. $\Omega \triangleq (\lambda x.xx)(\lambda x.xx) = \perp$.

- Identifying a solvable term with an unsolvable term is inconsistent. E.g. the theory \{\lambda x.xI\Omega = \perp\} proves anything!
(Semi)Decidability

- Does a $\lambda$-expr have a $\beta$-nf?
  - If so, what is it?
- Do two functions have equal graphs?
- Do we know when two real numbers are equal?
- We’ll need more machinery!

- Does a Turing Machine halt and accept/reject an input?

\[ s_0 \xrightarrow{(\square, \square, \leftarrow)} s_1 \xrightarrow{(\uparrow, \uparrow, \rightarrow)} s_2 \xrightarrow{(\square, 1, \rightarrow)} s_3 \]

\[ s_0 \xrightarrow{(\uparrow, \uparrow, \rightarrow)} (0,0,\rightarrow) \]

\[ s_1 \xrightarrow{(0,\square, \leftarrow)} (1,\square, \leftarrow) \]

\[ s_2 \xrightarrow{(1,1,\rightarrow)} \]

\[ s_3 \]
Programming in LC

• Every computer program can be translated into LC.
• LC offers control structures for
  1) composition;
  2) decomposition (or branching); and
  3) recursion.
• All (partial) recursive functions, i.e. effectively computable programs can be expressed using these three operations.
Side Note: Function Composition

- \( \circ \stackrel{\text{def}}{=} S(KS)K \rightarrow \lambda f g x.f(g x) \), so \( \circ M N O \rightarrow M(NO) \)
- \( \lambda \)-exprs are closed under composition, as shown above.
- \( \circ (\circ M N) O = \circ M (\circ N O) \), so composition is associative.
  - (This statement would be \( (M \circ N) \circ O = M \circ (N \circ O) \) in infix notation.)
- \( I \) is the identity element of composition.
- (Thus, \( \lambda \)-exprs with the composition operation have the algebraic structure of a monoid.)
Currying

- Another powerful technique is called currying, after Haskell Curry, a pioneer in Combinatory Logic and LC.
- Currying is the transformation of a \( f : (X \times Y) \rightarrow Z \) (in a Closed Monoidal Category) to a \( f' : X \rightarrow (Y \rightarrow Z) \).
  - If \( f(x,y) \equiv \phi(x,y) \), then \( f'(x) \overset{\text{def}}{=} (y \mapsto \phi(x,y)) \) (\( x \) is constant in RHS).
  - Thus, \( f(a,b) \equiv f'(a)(b) \) (i.e. \( \phi(a,b) \)). Remember example on p. 17?
- We have been Currying our functions all along...
Truth Values And Branching

• Truth values and branching as programming concepts can be expressed in terms of combinators.
  
  - $T \overset{\text{def}}{=} K \rightarrow \lambda xy.x$. (Thus, $TMN \rightarrow M$.)
  
  - $F \overset{\text{def}}{=} KI \rightarrow \lambda xy.y$. (Thus, $FMN \rightarrow N$.)

• We define, that
  
  - if $M$ then $N$ else $O \overset{\text{def}}{=} MNO$;
  
  - provided $M \rightarrow T$ or $M \rightarrow F$. 
Side Note: Tuples and Projections

• The idea of truth values has a generalisation: An *n*-tuple.

• The constructor: \( \langle x_0, x_1, \ldots, x_{n-1} \rangle \overset{\text{def}}{=} \lambda x_0 x_1 \ldots x_{n-1}. \lambda z. zx_0 x_1 \ldots x_{n-1} \).

• Projections: \( P_i \overset{\text{def}}{=} \lambda w. w(\lambda x_0 x_1 \ldots x_{n-1}. x_i) \), for every \( i \) s.t. \( 0 \leq i < n \).

• For instance,
  - \( \langle M, N, O \rangle \equiv (\lambda x_0 x_1 x_2. \lambda z. zx_0 x_1 x_2)MNO \rightarrow \lambda z. zMNO \).
  - Thus, \( P_1 \langle M, N, O \rangle \equiv (\lambda w. w(\lambda x_0 x_1 x_2. x_1))(\lambda z. zMNO) \rightarrow N \).

• This construction is called the *Scott encoding*. 
Self-Application

- Consider $\Omega \overset{\text{def}}{=} (\lambda x.xx)(\lambda x.xx)$
  - It’s $\lambda x.xx$ applied to itself!
  - It’s not in normal form.
  - $\Omega \rightarrow \Omega!$

- Self-application is not possible in Set Theory or most programming languages (for good reasons).
Side Note: Evaluation Strategies

- Consider what happens when reducing $\text{KI}\Omega$.
  - If we start from $\text{K}$, we obtain $\text{I}$, which is in normal form.
  - If we start from $\Omega$, we get $\Omega$ back, which is not in normal form.

- An algorithm for choosing the redex to reduce, step by step, until a NF is reached, is called an evaluation strategy. E.g.:
  - “Always choose the leftmost redex” is guaranteed (by the Church-Rosser Theorem, see p. 85) to always find the NF, if it exists.
  - “Reduce arguments before functions” fails to reach NF with $\text{KI}\Omega$. 
A Fixed Point Combinator

- $\Theta \overset{\text{def}}{=} (\lambda xy.y(xxy))(\lambda xy.y(xxy))$ is another funny expression.

- It is called *Turing’s Theta Combinator* (after Alan Turing).

- For any lambda expression $\varphi$, it holds that:

\[
\Theta M \equiv ((\lambda xy.y(xxy))(\lambda xy.y(xxy)))M \\
\equiv ((\lambda x.\lambda y.y(xxy))(\lambda xy.y(xxy)))M \\
\rightarrow (\lambda y.((\lambda xy.y(xxy))(\lambda xy.y(xxy)))y)M \\
\rightarrow M((\lambda xy.y(xxy))(\lambda xy.y(xxy)))M \\
\equiv M(\Theta M)
\]
Let’s See a Replay

- $\Theta M \equiv (\lambda x y. y(x xy))(\lambda x y. y(x xy))M$
  $\equiv ((\lambda x. \lambda y. y(x xy))(\lambda x y. y(x xy)))M$
  $\rightarrow (\lambda y. y((\lambda x y. y(x xy))(\lambda x y. y(x xy))y))M$
  $\rightarrow M((\lambda x y. y(x xy))(\lambda x y. y(x xy))M)$
  $\equiv M(\Theta M)$

- Of course, proofs can be refactored. For example, we could assign a name for the green part or get $\Theta$ back earlier.
  - Also, the first two pairs of blue parentheses were redundant.
Recursive Function Definitions

- Consider factorial: \( 0! = 1, (n+1)! = (n+1) \cdot (n!), \forall n \in \mathbb{N}. \)
  - This kind of explicitly recursive definition is not possible in LC.
  - \( \lambda \)-exprs are anonymous, so they cannot refer to their own values.

- **Fixed point combinator** is a combinator \( F \), s.t. \( FM = M(FM) \) for an arbitrary \( \lambda \)-expr \( M \).
  - \( \Theta \) is a fixed point combinator.
  - Fixed point combinators, together with lambda abstraction, introduce a backdoor that enables recursion...
How To Hack The System To Get Recursion

- Let’s say that we want to define function $F$ recursively.
  - Firstly, let $F \overset{\text{def}}{=} \Theta \phi$, so $F \rightarrow \phi(\Theta \phi) \equiv \phi F$.

- In order to eventually stop the recursion, some condition $P$ is needed, and $\phi$ needs to be of the form:
  - $\lambda fx. \text{if } (P x) \text{ then } (G x) \text{ else } (H x(f x))$
  - In this case, we say that $F$ is defined by recursion over $G$ and $H$.

- Thus, $F \rightarrow \lambda x. \text{if } (P x) \text{ then } (G x) \text{ else } (H x(F x))$. 
Now, it’s time to exploit combinators!

Let \( Z^0 \) be a \( \lambda \)-expr that we designate as a natural number.

If \( n \) is a natural number, so is \( S^+ n \), by definition.

\[
\text{Zero } Z^0 \overset{\text{def}}{=} T \quad \text{and} \quad \text{Zero } (S^+ n) \overset{\text{def}}{=} F.
\]

\[
P^- Z^0 \overset{\text{def}}{=} Z^0, \quad P^- (S^+ n) \overset{\text{def}}{=} n.
\]

There are at least two different consistent definitions for \( Z^0 \), Zero, \( S^+ \) and \( P^- \).
Arithmetics

- Let $0 \approx Z^0$ and $n \approx (S^+ \circ S^+ \circ S^+ \circ \ldots \circ S^+) Z^0$ (with $n$ repetitions of $S^+$). (We used an infix ‘$\circ$’ for readability.)
  
  - $1 \equiv S^+ Z^0$, $2 \equiv S^+ (S^+ Z^0)$, $3 \equiv S^+ (S^+ (S^+ Z^0))$, etc.

- We can now proceed with:
  
  - $+ \overset{\text{def}}{=} \Theta X$;
  
  - $X \overset{\text{def}}{=} \lambda fnm. \text{if} (\text{Zero } m) \text{ then } n \text{ else } R$; and
  
  - $R \overset{\text{def}}{=} (f (S^+ n) (P^- m))$
One Plus One Equals Two

\[ + \ 1 \ 1 \equiv \ \Theta X \ 1 \ 1 \]
\[ \to \ X(\Theta X) \ 1 \ 1 \]
\[ \equiv (\lambda fnm. \text{if } (\text{Zero } m) \text{ then } n \text{ else } R) + 1 \ 1 \]
\[ \to \ \text{if } (\text{Zero } 1) \text{ then } 1 \text{ else } (+ (S^+ 1) (P^- 1)) \]
\[ \to \ F \ 1 \ (+ (S^+ 1) (P^- 1)) \]
\[ \to \ + (S^+ 1) Z^0 \]
\[ \to \ \text{if } (\text{Zero } Z^0) \text{ then } (S^+ 1) \text{ else } (+ (S^+ 1) Z^0) \]
\[ \to \ S^+ 1 \equiv 2. \]
Did That Look Complicated?

- Consider the following C-style programming example:

```c
int fact(int n)
{
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```

- The same idea can be expressed elegantly in LC:

```latex
\text{fix} (\lambda fn. \\
    \text{if } (\text{Zero } n) \text{ then } 1 \\
    \text{else } (S \cdot (\circ f P^-) n))
\text{ with } \text{fix} \overset{\text{def}}{=} \Theta.
```
The Limits of Lambda Calculus

• Using the techniques presented so far, it is possible to define each and every recursive function in LC.

• Now that we’ve seen what can be done in LC, it’s time to ask: what can’t be done in LC?

• It turns out to be impossible to predict whether or not an arbitrary λ-expr has a normal form.

• We’re going to prove this using combinators.
A Little Bit More Machinery

• Let’s define two more standard combinators:
  - \( B \overset{\text{def}}{=} K(SK)K \), or equivalently \( B \overset{\text{def}}{=} \lambda xyz.x(yz) \).
  - It follows that \( BMNO \to M(NO) \).
  - \( C \overset{\text{def}}{=} S(S(KS)(S(KK)S))(KK) \), or equivalently \( C \overset{\text{def}}{=} \lambda xzy.xzy \).
  - It follows that \( CMNO \to MON \).
### Why These Are the Standard Combinators?

<table>
<thead>
<tr>
<th>Combinator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Ix = x )</td>
<td>The identity function</td>
</tr>
<tr>
<td>( Kx = \lambda y.x )</td>
<td>Constant function</td>
</tr>
<tr>
<td>( Sfg = \lambda x.fx(gx) )</td>
<td>Can be used for recursion</td>
</tr>
<tr>
<td>( Bfg = \lambda x.f(gx) )</td>
<td>Function composition</td>
</tr>
<tr>
<td>( Cfgg = fhg )</td>
<td>Inverts if-then-else</td>
</tr>
</tbody>
</table>
## Combinator Cheatsheet

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Redex*</th>
<th>Reduct*</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>$\lambda x.x$</td>
<td><strong>I</strong></td>
<td>$M$</td>
</tr>
<tr>
<td>K</td>
<td>$\lambda xy.x$</td>
<td><strong>K</strong></td>
<td>$MN$</td>
</tr>
<tr>
<td>T</td>
<td>$\lambda xy.x$</td>
<td><strong>T</strong></td>
<td>$M$</td>
</tr>
<tr>
<td>F</td>
<td>$\lambda xy.y$</td>
<td><strong>F</strong></td>
<td>$N$</td>
</tr>
<tr>
<td>S</td>
<td>$\lambda xyz.xz(yz)$</td>
<td><strong>S</strong></td>
<td>$MO(NO)$</td>
</tr>
<tr>
<td>B</td>
<td>$\lambda xyz.x(yz)$</td>
<td><strong>B</strong></td>
<td>$M(NO)$</td>
</tr>
<tr>
<td>C</td>
<td>$\lambda xyz.xzy$</td>
<td><strong>C</strong></td>
<td>$MON$</td>
</tr>
<tr>
<td>Θ</td>
<td>$(\lambda xy.y(xxy))(\lambda xy.y(xxy))$</td>
<td><strong>Θ</strong></td>
<td>$M(ΘM)$</td>
</tr>
<tr>
<td>Ω</td>
<td>$(\lambda x.xx)(\lambda x.xx)$</td>
<td><strong>Ω</strong></td>
<td>$Ω$</td>
</tr>
</tbody>
</table>

* = When fully applied
Getting Ready For The Big Surprise

- We’re about to reach the climax of these slides, the famous *Halting Problem*, expressed in the language of combinators.
  - The problem is to decide whether an arbitrary $\lambda$-expr has a $\beta$-nf.
- Let’s assume that there is such a $\lambda$-expr $N$ that for any $F$:
  - $NF \rightarrow \mathbf{T}$, if $F$ has a normal form; and
  - $NF \rightarrow \mathbf{F}$, if $F$ doesn’t have a normal form.
- Let $Z \overset{\text{def}}{=} C(C(BN(\text{SII}))\Omega)I$. 
The Halting Problem

- $ZZ \equiv (C(C(BN(SII))Ω)I)Z$
  $\rightarrow C(BN(SII))ΩZI$
  $\rightarrow BN(SII)ZΩI$
  $\rightarrow N((SII)Z)ΩI$
  $\rightarrow N(IZ(IZ))ΩI$
  $\rightarrow N(ZZ)ΩI$.

- $ZZ$ asks $N$ whether $ZZ$ itself has a normal form or not.
  - A rather strange loop, isn’t it?
The Halting Problem, Continued

- If $N(ZZ) \rightarrow T$, then $ZZ \rightarrow N(ZZ)ΩI \rightarrow TΩI \rightarrow Ω$, which does not have a normal form.
  - This is a contradiction. $N$ didn’t see that coming!
- If $N(ZZ) \rightarrow F$, then $ZZ \rightarrow N(ZZ)ΩI \rightarrow FΩI \rightarrow I$, which does have a normal form.
  - Again, we arrive at a contradiction.
- Therefore, we must conclude that $N$ cannot exist. ■
Summary on LC

- The fundamental concept of LC is lambda expressions.
  - Lambda expressions behave like anonymous functions.
- Every λ-expr can be constructed using three rules.
- Reducing lambda expressions is like performing arithmetics.
- LC is the functional analogue to machine language.
  - It provides only the minimal set of building blocks.
  - Recursion emerges in LC through self-application.
One More Thing About Combinators

- Abstraction is not necessary if we axiomatize the reducts for fully applied combinators (see the table on p. 116).
- Combinators can simulate other λ-exprs.
  - (We know the converse already.)
- If we restrict the rules to only allow the use of combinators and free variables, we don’t need substitution at all.
  - Such restriction is called Combinatory Logic (CL).
Removing Lambda Abstractions

1) \([x] \overset{\text{def}}{=} x\);
2) \([\lambda x.x] \overset{\text{def}}{=} I\);
3) \([\lambda x.M] \overset{\text{def}}{=} (K[M]), \text{ if } x \text{ is not free in } M;\)
4) \([\lambda x.\lambda y.M] \overset{\text{def}}{=} [\lambda x.[\lambda y.M]], \text{ if } x \text{ is free in } M;\)
5) \([\lambda x.MN] \overset{\text{def}}{=} (C \ [\lambda x.M] \ [N]), \text{ if } x \text{ is free only in } M;\)
6) \([\lambda x.MN] \overset{\text{def}}{=} (B \ [M] \ [\lambda x.N]), \text{ if } x \text{ is free only in } N;\)
7) \([\lambda x.MN] \overset{\text{def}}{=} (S \ [\lambda x.M] \ [\lambda x.N]), \text{ if } x \text{ is free in } M \text{ and } N;\)
8) \([MN] \overset{\text{def}}{=} ([M] \ [N])\)
Example Abstraction Elimination

\[(\lambda x. \lambda z. zcy) d] = \left( [\lambda x. \lambda z. zcy] \left[ [d] \right] \right) \quad (8)
= ([\lambda x. \lambda z. zcy]) d \quad (1)
= (K[\lambda z. zcy]) d \quad (3)
= (K(C[\lambda z. zc][y])) d \quad (5)
= (K(C(C[\lambda z. z][c])[y])) d \quad (5)
= (K(C(CI[c])[y])) d \quad (2)
= (K(C(CIc)[y])) d \quad (1)
= K(C(CIc)y) d \quad (1)\]
The Last Trees

\[(\lambda x. \lambda z. z c y) d \equiv \text{K(C(CIc)y)} d\]
Conclusion

- We now (hopefully) understand the concept of a function from a computational (LC/CL) perspective.
  - Extensional equality of functions is undecidable in general case.
  - Halting problem is the computational analogue to the Gödel’s incompleteness theorems.
- Lambda Calculus is a deep field of mathematics with connections to many other disciplines.
  - This presentation barely scratched the surface.
Related Topics

- So, what next? We can extend LC into many directions. E.g.:
  - Q: When does an application make sense? A: Type Theory
  - Q: What kind of algebra is involved in LC? A: Category Theory
  - Q: What kind of calculus is involved in LC? A: Domain Theory
  - Q: How can we profit from LC in logic? A: Proof Theory
  - Q: What kind of programming is LC? A: Functional Programming
  - Q: Why is LC future-proof? A: Non-Classical Computing
# Programming Terminology Inspired by LC

<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-Order Function</td>
<td>A function that returns (or takes) another function.</td>
</tr>
<tr>
<td>Function Composition</td>
<td>A higher-order function that composes two functions.</td>
</tr>
<tr>
<td>Partial Application</td>
<td>Applying a single argument to a higher-order function.</td>
</tr>
<tr>
<td>Currying</td>
<td>Defining a higher-order function.</td>
</tr>
<tr>
<td>Recursion</td>
<td>Defining a function in terms of its own values.</td>
</tr>
<tr>
<td>Evaluation Strategy</td>
<td>Determines the order of reduction steps for an expression.</td>
</tr>
<tr>
<td>Pattern Matching</td>
<td>A generalization of the <code>if-then-else</code> construct.</td>
</tr>
<tr>
<td>Type</td>
<td>A proposition about the structure of an object.</td>
</tr>
</tbody>
</table>
Thank You!

https://xkcd.com/1270/
Warm-Up Exercises

1) Draw a parse tree (see p. 33 and 34) for \((\lambda x.\lambda y.\lambda z. zxw) cd\).  
2) Reduce this tree, step by step, to a normal form. (Cf. p. 64–67.)  
3) Prove the claims on p. 92.  
4) Draw your own World/Local Map of LC/CL. (See p. 21, 81.)  
5) Convert \((K(C(Clc)y))d\) back to LC and find its normal form.  
6) \(Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))\) is another fixed point combinator.  
   Can you tell the subtle behavioral difference between \(Y\) and \(Θ\)?
Advanced Exercises

1) Develop an example that uses all the rules given in p. 122.
2) Consider the Euclidean algorithm flowchart in p. 5 and 6. Describe it using the techniques using provided in these slides.
   - Hint: You may take elementary arithmetics as granted in LC.
3) Can you encode the algorithm shown in p. 122 using LC/CL?
4) Can you analyze the MIU-system (see p. 11) in LC/CL?
5) Construct a Turing Machine or other interpreter for LC/CL.
Further Reading

- Nederpelt, Rob; Geuvers, Herman: Type Theory and Formal Proof: An Introduction