Introduction to the Lambda Calculus

Overview:
- What is Computability? – Church’s Thesis
- The Lambda Calculus
- Scope and lexical address
- The Church-Rosser Property
- Recursion

References:
- Carl A. Gunter, "Semantics of Programming Languages", MIT Press, 1992
- David A. Schmidt, "The Structure of Typed Programming Languages", MIT Press, 1994

What is Computable?
- Computation is usually modeled as a mapping from inputs to outputs, carried out by a "formal machine", or program, which processes its input in a sequence of steps.
- An "effectively computable" function is one that can be computed in a finite amount of time using finite resources.

Turing Machine
- A Turing machine is an abstract representation of a computing device. It consists of a read/write head that scans (possibly infinitely) one-dimensional (bi-directional) tape divided into squares, each of which is inscribed with a 0 or 1.
- Computation begins with the machine in a given "state", scanning a square. It erases what it finds there, prints a 0 or 1, moves to an adjacent square, and goes into a new state.
- This behavior is completely determined by three parameters:
  - the state the machine is in,
  - the number on the square it is scanning, and
  - a table of instructions.
Both specifications describe the same Turing machine.

**Example**

Turing machine is more like a computer program (software) than a computer (hardware):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(1,0)</td>
<td>(3,1)</td>
</tr>
<tr>
<td>1</td>
<td>(2,1)</td>
<td>(3,0)</td>
</tr>
<tr>
<td>2</td>
<td>(3,0)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>3</td>
<td>(3,0)</td>
<td>(3,3)</td>
</tr>
</tbody>
</table>

Church’s Thesis

- Effectively computable functions [from positive integers to positive integers] are just those definable in the lambda calculus.
  
Or, equivalently:

- It is not possible to build a machine that is more powerful than a Turing machine.

Church Thesis Is Not Computable

- Church’s thesis cannot be proven because "effectively computable" is an intuitive notion, not a mathematical one.

- It can only be refuted by given a counter-example – a machine that can solve a problem not computable by a Turing machine.

- So far, all models of effectively computable functions have shown to be equivalent to Turing machines (or the lambda calculus).
Halting Problem

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called uncomputable.

Assuming Church’s thesis is true, an uncomputable problem cannot be solved by any real computer.

The Halting Problem:
Given an arbitrary Turing machine and its input tape, will the machine eventually halt?

The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

Ackermann Function

The Ackermann function is the simplest example of a well-defined total function which is computable but not primitive recursive.

The function \( A(x, y) \), while Turing computable, grows much faster than polynomials or exponentials. The definition is:

\[
\begin{align*}
A(0, n) &= n + 1 \\
A(m+1, 0) &= A(m, 1) \\
A(m+1, n+1) &= A(m, A(m+1, n))
\end{align*}
\]

Examples of Ackermann

<table>
<thead>
<tr>
<th>( m \times n )</th>
<th>( m = 0 )</th>
<th>( m = 1 )</th>
<th>( m = 2 )</th>
<th>( m = 3 )</th>
<th>( m = 4 )</th>
<th>( m = 5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 0 )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>( n = 1 )</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>( n = 2 )</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>( n = 3 )</td>
<td>5</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>125</td>
<td>253</td>
</tr>
<tr>
<td>( n = 4 )</td>
<td>13</td>
<td>65533</td>
<td>2^{65533}</td>
<td>2^{2^{65533}}</td>
<td>2^{2^{2^{65533}}}</td>
<td>2^{2^{2^{2^{65533}}}}</td>
</tr>
<tr>
<td>( n = 5 )</td>
<td>65533</td>
<td>4 \times 65533</td>
<td>4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 4 \times 4 \times 65533</td>
</tr>
<tr>
<td>( n = 6 )</td>
<td>4 \times 65533</td>
<td>4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 4 \times 4 \times 65533</td>
<td>4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 65533</td>
</tr>
</tbody>
</table>
The Lambda Calculus

- Lambda calculus is a language with clear operational and denotational semantics capable of expressing algorithms. It forms also a compact language to denote mathematical proofs.
- Logic provides a formal language in which mathematical statements can be formulated and provides deductive power to derive these. Type theory is a formal system, based on lambda calculus and logic, in which statements, computable functions, and proofs all can be naturally represented.
- The lambda calculus is a good medium to represent mathematics on a computer with the aim to exchange and store reliable mathematical knowledge.

The Definition of the Lambda Calculus

- The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.
- The lambda calculus can be viewed as the simplest possible pure functional programming language.

Syntax

\[ e ::= x \quad \text{variable} \]
\[ \lambda x . e \quad \text{abstraction (function)} \]
\[ e_1 e_2 \quad \text{(function) application} \]

We use parentheses to enhance readability and to build or retain sub-term structures.
Operational Semantics

\(\alpha\)-conversion (renaming):
\[ \lambda x . e \leftrightarrow \lambda y . [y/x]e \quad \text{where } y \text{ is fresh in } e \]

\(\beta\)-reduction (application):
\[ (\lambda x . e_1) e_2 \rightarrow [e_2/x]e_1 \quad \text{avoiding name capture} \]

\(\eta\)-reduction:
\[ \lambda x . (e x) \rightarrow e \quad \text{if } x \text{ is not free in } e \]

Examples

\(\text{id} = \lambda x . x \)
\(\Omega = (\lambda x . x x) (\lambda x . x x) \)
\(\text{pair}(x, y) = \lambda x . \lambda y . \lambda z . z x y \)

Currying

- Since a lambda abstraction only binds a single variable, functions with multiple parameters must be modeled as curried higher-order functions. This method is named after the logician H. B. Curry, who popularized the approach.
- To improve readability, multiple lambdas can be suppressed as follows:
  \[ \lambda x y . x = \lambda x . \lambda y . x \]
  \[ \lambda b x y . (b x) y = \lambda b . \lambda x . \lambda y . (b x) y \]
**Reference or Declaration**

- In a program, variables can appear in two different ways:
  - as declarations: \((\lambda x . e)\)
    - The occurrence of \(x\) in the lambda-abstraction introduces the variable \(x\) as a name for some value.
  - as references: \((f \, x \, y)\)
    - Here all variables, \(f, x, y\), appear as references, whose meanings are defined by an enclosing declaration.

**Binding**

- A value named by a variable is also called denotation (meaning). The denotation must come from some declaration, we say the variable is bound by that declaration, or it refers to that declaration.

- Declarations in most programming languages have limited scope (the area, where the variable is applicable). Therefore, the same variable name may occur multiple times in the program text, but being used for different purposes. We use binding rules to determine the declaration to which a concrete variable use refers.

**Scoping Rules**

- We call a language **statically scoped**, if we can determine the declaration of a variable by analyzing the program text alone.

- We call a language **dynamically scoped**, if we cannot determine the declaration of a variable until the program is executed.
Binding Rules in Lambda Calculus

- In $\lambda x . e$, the occurrence of $x$ is a declaration that binds all occurrences of that variable in $e$ unless some intervening declaration of the same variable occurs.

Examples:
- $\lambda x . \lambda y . y x$
- $\lambda x . \lambda y . (\lambda x . (x y)) x y$

Occurs Free, Occurs Bound

- A variable $x$ occurs free in $e$ if and only if there is some use of $x$ in $e$ that is not bound by any declaration of $x$ in $e$.
- A variable $x$ occurs bound in an expression $e$ if and only if there is some use of $x$ in $e$ that is bound by a declaration of $x$ in $e$.

Examples:
- $\lambda x . x y$: $x$ occurs bound, but $y$ occurs free
- $\lambda f . \lambda x . f x$: both $f$ and $x$ occur bound

Free and Bound Variables

- In $\lambda x . e$, the variable $x$ is bound by the enclosing $\lambda$. A variable that is not bound is free.

$\text{fv}(x) = \{x\}$
$\text{bv}(x) = \emptyset$
$\text{fv}(\lambda x . e) = \text{fv}(e) \setminus \{x\}$
$\text{bv}(\lambda x . e) = \text{bv}(e) \cup \{x\}$

- An expression with no free variables is closed (otherwise it is open). For example, $y$ is bound and $x$ is free in the open expression $\lambda y . x y$. 
Combinators

- Lambda expressions with no free variables are called **combinators**.
- Every procedure, when applied to all its necessary arguments (i.e., a procedure call), is a combinator.

The Scope of a Variable

- Problem: For each variable reference, find the corresponding declaration to which it refers.
- This problem is easier to solve, when we ask: Given a declaration, which variable references refer to it?
- In the definition of programming languages, binding rules for variables typically associate with each declaration of a variable a region of the program within the declaration is effective.

Blocks

- In lambda calculus as in many modern programming languages, regions can be **nested** within each other.
- We call these languages **block-structured**, and regions are also called **blocks**.
Visibility

- The scope of a variable, say $x$, can include inner regions that hide the variable $x$. Within these inner regions the outer declaration of the variable $x$ is hidden, that is, the scope of $x$ has a hole.

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

Contour Diagrams

- We use contour diagrams to picture the borders of a region:

![Contour Diagram]

- The lexical (or static) depth of a variable reference is the number of contours crossed to find the associated declaration.
- The lexical depth is used in compilers to tell how many static links to traverse to find a variable (see frame pointers).

Lexical Address

- The declarations associated with a region may be numbered in the order of their appearance in the text. Each variable reference may then be associated with two numbers:
  - its lexical depth and
  - its position.
- To illustrate lexical addresses, we replace every variable reference $x$ with an expression $(x : d p)$, where $d$ is the lexical depth and $p$ is the declaration position of $x$.

[$(\lambda x y . (\lambda a . (x : 1 0) (a : 0 0) (y : 1 1) (x : 0 0))$]
Alpha Conversion

- Alpha conversion allows one to rename bound variables.
- A bound name x in the lambda abstraction (λx. e) may be substituted by any other name y, as long as there are no free occurrences of y in e.

Consider:
(λx. λy. x y) y → [(λx. λy. x y)/x] y
β-reduction
= (λy. y y)
η-reduction

Beta Reduction

- Beta reduction is the computational engine of the lambda calculus.
- Define: I  x x
- Now consider:
1 I = (λx. x)(λx. x) → [(λx. x)/x] x
β-reduction
= I
substitution

Name Capture

- Syntactic substitution will not always work:
(λx. λy. x y) y → [(λy. x y)/x] y
β-reduction
= (λy. y y)
incorrect!
- Since y is already bound in λy. x y, we cannot directly substitute y for x, because the y inside term λy. x y is different from the variable y applied as argument to (λx. λy. x y).
Substitution

- We must define substitution carefully to avoid name capture:

\[ [e/x]x = e \]
\[ [e/x]y = y \] if \( x \neq y \)
\[ [e/x](e_1 e_2) = ([e/x]e_1)[e/x]e_2 \]
\[ [e/x](\lambda x . e_1) = \lambda x . [e/x]e_1 \]
\[ [e/x](\lambda y . e_1) = \lambda y . [e/x]e_1 \] if \( x \neq y \) and \( y \notin \text{fv}(e) \)
\[ [e/x](\lambda y . e_1) = \lambda z . [e/x][z/y]e_1 \] if \( x \neq y \) and \( z \notin (\text{fv}(e) \cup \text{fv}(e_1)) \)

Consider:

\( (\lambda x . (\lambda y . x) (\lambda y . x)) x ) y \)
\( \rightarrow [y/x](\lambda y . x) (\lambda y . x) x \)
\( = (\lambda y . y) (\lambda y . x) x \)
\( \beta\text{-reduction} \)

\( \rightarrow (\lambda z . z) (\lambda y . x) x \)
\( \beta\text{-reduction} \)
\( \rightarrow (\lambda z . z) (\lambda y . x) x \)
\( \beta\text{-reduction} \)
\( \rightarrow \ldots \)

Eta Reduction

- \( \eta \)-reductions allow one to remove "redundant lambdas."

- Suppose that \( f \) is a closed expression (i.e., \( x \) does not occur free in \( f \)). Then:

\( (\lambda x . f x) y \rightarrow [y/x]f \rightarrow f y \) \( \beta\text{-reduction} \)

More generally, this will hold whenever \( x \) does not occur free in \( f \). In such cases, we can always rewrite \((\lambda x . f x)\) as \( f \).

Normal Forms

- A lambda expression is in normal form if it can no longer be reduced by the \( \beta \) or \( \eta \)-reduction rules.

- But not all lambda expressions have normal forms:

\( \Omega = (\lambda x . x x) (\lambda x . x x) \rightarrow (\lambda x . x x)(\lambda x . x x) \)
\( \rightarrow (\lambda x . x x)(\lambda x . x x) \) \( \beta\text{-reduction} \)
\( \rightarrow (\lambda x . x x)(\lambda x . x x) \) \( \beta\text{-reduction} \)
\( \rightarrow \ldots \)

- The reduction of a lambda term to a normal form is analogous to a "Turing machine halts" or a "program terminates."
Evaluation Order

- Most programming languages are strict, that is, all expressions passed to a function call are evaluated before control is passed to the function (e.g. Java).
- Most modern functional languages (e.g. Haskell), on the other hand, use lazy evaluation, that is, expressions are only evaluated when they are needed.

\[
\text{square } n = n \times n
\]

Applicative-order reduction:
- \(\text{square } (2 + 5)\) → \(\text{square } 7\) → \(7 \times 7\) → \(49\)

Normal-order reduction:
- \(\text{square } (2 + 5)\) → \((2 + 5) \times (2 + 5)\) → \(7 \times (2 + 5)\) → \(7 \times 7\) → \(49\)

Applicative-Order Reduction

Motivation:
- Modeling call-by-value in programming languages
- In function calls, evaluate arguments then invoke function
- In the lambda-calculus, this means:
  - If \(x\) (\(\lambda e\) e), reduce \(x\) to normal form using applicative order reduction
  - Then reduce \(e\) to normal form using applicative order reduction
  - If \(e\) is a lambda abstraction, do beta reduction, and reduce the result to normal form using applicative order reduction
Syntax makes it easy

Write expression using fully parenthesized notation.
Always perform rightmost-innermost beta reduction by repeatedly scanning for rightmost innermost (left parenthesis) occurrence of \((\text{ld x . e} \ q e)\) terms.
This includes reduction of primitives, e.g. \((\text{add} \ 1 \ 2)\).

Applicative-Order Example

Consider:
\[
((\lambda x . ((\lambda y . \text{add} \ y \ y) (\text{mul} \ x \ x))) (\text{sub} \ 3 \ 1))
\]
Applicative order reduction gives:
\[
((\lambda x . ((\lambda y . \text{add} \ y \ y) (\text{mul} \ x \ x))) (\text{sub} \ 3 \ 1))
((\lambda x . ((\lambda y . \text{add} \ y \ y) (\text{mul} \ x \ x))) \ 2)
((\lambda x . (\text{add} (\text{mul} \ x \ x) (\text{mul} \ x \ x))) \ 2)
(\text{add} (\text{mul} \ 2 \ 2) (\text{mul} \ 2 \ 2))
(\text{add} (\text{mul} \ 2 \ 2) \ 4)
(\text{add} \ 4 \ 4)
8
\]

Applicative-Order - Head Normal Form

Consider:
\[
((\lambda x . ((\lambda y . \text{add} \ y \ y) (\text{mul} \ x \ x))) (\text{sub} \ 3 \ 1))
\]
Applicative order reduction gives:
\[
((\lambda x . ((\lambda y . \text{add} \ y \ y) (\text{mul} \ x \ x))) (\text{sub} \ 3 \ 1))
((\lambda y . \text{add} \ y \ y) (\text{mul} \ 2 \ 2))
((\lambda y . \text{add} \ y \ y) \ 4)
(\text{add} \ 4 \ 4)
8
\]
Normal-Order Example

Consider:

\[ ((\lambda x . (\lambda y . \text{add } y y) (\text{mul } x x)) (\text{sub } 3 1)) \]

Normal-order reduction gives:

\[ ((\lambda x . (\lambda y . \text{add } y y) (\text{mul } x x)) (\text{sub } 3 1)) \]
\[ ((\lambda y . \text{add } y y) (\text{mul } (\text{sub } 3 1) (\text{sub } 3 1))) \]
\[ (\text{add} (\text{mul } 2 2) (\text{mul } 2 2)) \]
\[ 8 \]

The Church-Rosser Property

The diamond property - confluence:

"If an expression can be evaluated at all, then it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative-order reduction), then all of these evaluation orders yield the same result."

Confluence - Implication

Evaluation order "does not matter" in the lambda calculus. However, applicative order reduction may not terminate, even if a normal form exists!

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

Applicative-order reduction
\[ \Rightarrow (\lambda x . y) (\lambda x . x x) (\lambda x . x x) \]
\[ \Rightarrow (\lambda x . y) (\lambda x . x x) (\lambda x . x x) \]

Normal-order reduction
\[ \Rightarrow y \]
\[ \Rightarrow y \]
Recursion

- Most programming languages support the definition of recursive abstractions:
  - Records (to construct linked lists)
  - Procedure (to implement inductively specified data types)
  - Mutual dependent data structures like classes
- Recursion is a challenging mechanism and may often lead to complications in program understanding.

A Recursive Problem

- Suppose we want to define the operation plus using only the operators increment and decrement. We may write:

  \[
  \text{plus} = \lambda n \ m . \ \text{if} \ n \ \text{then} \ (\text{plus} \ (\text{dec} \ n) \ (\text{inc} \ m)) \ \text{else} \ m
  \]

- Unfortunately, this is not a definition, since we are trying to use "plus" before it is defined.

- Although recursion is fundamental to programming, it is not a primitive. Therefore, we must find a way to "program" it!

Recursive Functions As Fixed Points

- However, we can obtain a closed expression by abstracting over plus:

  \[
  \text{rplus} = \lambda n \ m . \ \text{if} \ n \ \text{then} \ (\text{plus} \ (\text{dec} \ n) \ (\text{inc} \ m)) \ \text{else} \ m
  \]

- Now, let "fplus" be the actual addition function we want. We must pass it to "rplus" as a parameter before we can perform any additions.

- (rplus fplus) is the function we want. In other words, we are looking for an fplus such that:

  \[
  \text{rplus} \ fplus \leftrightarrow fplus
  \]

That is, we are searching for the fixed-point of "rplus".
Fixed-Points

- In general, a fixed-point of a function is a value in the function’s domain, which is mapped to itself by the function. Therefore, a fixed-point of a function \( f \) is a value \( p \) such that \( f(p) = p \).

- Examples:
  - \( (\text{factorial } 1) = 1 \)
  - \( (\text{factorial } 2) = 2 \)
  - \( (\text{fibonacci } 0) = 0 \)
  - \( (\text{fibonacci } 1) = 1 \)

- However, not all functions have exactly one fixed point:
  - \( \text{inc} = \lambda x . \text{add } x \) has none.

Fixed-Point Theorem

- Fixed-Point Theorem:
  - For every \( F \) there exists a fixed-point \( X \) such that \( F X = X \).

  Proof:
  - Let
    \[ Y = \lambda F . (\lambda x . F (x x)) (\lambda x . F (x x)) \]
  - Now consider:
    \[ X = Y F \]
    \[ \rightarrow (\lambda x . F (x x)) (\lambda x . F (x x)) \]
    \[ \rightarrow F (\lambda x . F (x x)) (\lambda x . F (x x)) \]
    \[ \rightarrow F X \]
  - The “\( Y \) combinator” can always be used to find a fixed-point of an arbitrary lambda expression, if such a fixed-point exists.

plus = Y rplus

- \( \text{plus} = \lambda n m . \text{if } n \text{ then } (\text{plus} (\text{dec } n) (\text{inc } m)) \text{ else } m \)

  Plus is a fixed-point of \( rplus \). By the fixed point theorem, we can take:
  
  \[ \text{plus} = Y \ rplus \]
Unfolding Recursive Lambda Expressions

\[
\begin{align*}
\text{plus} &\ 1 \\
\Rightarrow &\ (Y \text{plus}) \ 1 \\
\Rightarrow &\ \text{if} \ 1 \ \text{then} \ (\text{plus} \ (\text{pred} \ 1)) \ \text{else} \ 1 \\
\Rightarrow &\ (\text{plus} \ (\text{pred} \ 1)) \\
\Rightarrow &\ \text{if} \ (\text{pred} \ 1) \ \text{then} \ (\text{plus} \ (\text{pred} \ (\text{pred} \ 1))) \ \text{else} \ (\text{succ} \ 1) \\
\Rightarrow &\ (\text{if} \ 0 \ \text{then} \ (\text{pred} \ (\text{pred} \ 1))) \ \text{else} \ (\text{succ} \ 1) \\
\Rightarrow &\ (\text{succ} \ 1) \\
\Rightarrow &\ 2
\end{align*}
\]

Strict Fixed-Point Operator

The fixed-point operator \( Y \) is useless in a call-by-value setting, since the expression \( Y \ g \) diverges for any \( g \). In call-by-value settings we use, therefore, the operator \( \text{fix} \):

\[
\text{fix} \equiv \lambda f . (\lambda x . f (\lambda y . x x y)) (\lambda x . f (\lambda y . x x y))
\]

Unfolding Recursive Lambda Expressions II

\[
\begin{align*}
\text{plus} &\ 1 \\
\Rightarrow &\ (\lambda x . \text{plus} \ (\lambda y . x y)) \ 1 \\
\Rightarrow &\ (h \ h) \ 1 \\
\Rightarrow &\ \text{if} \ 0 \ \text{then} \ (\text{pred} \ (\text{pred} \ 0)) \ \text{else} \ 2 \\
\Rightarrow &\ 2
\end{align*}
\]
**SKI Combinator Reduction**

- SKI combinator reduction is an implementation technique that yields normal-order (lazy) evaluation in the most natural way.
- A lambda calculus expression (that denotes a program) can be transformed into an equivalent combinator expression that contains only constants and applications. Moreover, this combinator expression will contain neither any lambda abstractions nor any variables.
- The reduction of combinator expressions is based on a combinator calculus that does not have a beta reduction, hence term rewriting does not need to manipulate variables and environments explicitly.

<table>
<thead>
<tr>
<th>Combinator</th>
<th>Name</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>I x y z = x</td>
<td>Identity</td>
<td>I x z y</td>
</tr>
<tr>
<td>K x y = x</td>
<td>Constant</td>
<td>K x y z</td>
</tr>
<tr>
<td>S x y z = x (y z)</td>
<td>Distribution</td>
<td>S x z y z (y z)</td>
</tr>
<tr>
<td>B x y z = x (y z)</td>
<td>Composition</td>
<td>B x z y z (y z)</td>
</tr>
<tr>
<td>C x y z = x (y z)</td>
<td>Swap</td>
<td>C x z y z</td>
</tr>
</tbody>
</table>

The first three combinators I, K, and S are sufficient to transform every lambda expression into an equivalent combinator expression.

**A Combinator Language**

Syntax:

```
<expressions> ::= k  : constant
   | S
   | K
   | I
   | ( <expression> <expression> )
```

18
Translation Function

Let \( e \) be a lambda calculus expression. Then the function \( U(e) \) translates \( e \) into an equivalent combinator expression:

\[
U(e) = e \\
U(\lambda x . e) = [x](U(e)) \\
U(e_1 e_2) = U(e_1) U(e_2)
\]

The function \([x](e)\) is defined as follows:

\[
[x](k) = Kk \\
[x](x) = I \\
[x](y) = Ky \\
[x](e_1 e_2) = S[x](e_1)[x](e_2)
\]

Building a Combinator Expression

\[
U(\lambda x : \lambda y . x y) = [x](U(y x)) \\
= [x][y](U(x y)) \\
= [x][y][x](y x) \\
= [x][S(K x) I] \\
= S[x](S(K x)) [x](I) \\
= S(S[S(K S) S(K x)]) (K I) \\
= S(S(K S)(S[S(K x)])(K I)) (K I)
\]
Reduction of a Combinator Expression

\[(\lambda x. \lambda y. x y) \ A \ B \rightarrow A \ B\]

<table>
<thead>
<tr>
<th>Step</th>
<th>Expression</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(S (S \ (K \ S) \ (S \ (K \ K) \ I)) \ (K \ I))</td>
<td>(S \ x \ y \ z \ = \ x \ z \ (y \ z))</td>
</tr>
<tr>
<td>2</td>
<td>(S \ (K \ S) \ (S \ (K \ K) \ I) \ A \ (K \ I) \ B)</td>
<td>(S \ x \ y \ z \ = \ x \ z \ (y \ z))</td>
</tr>
<tr>
<td>3</td>
<td>(K \ S \ (S \ (K \ K) \ I) \ A \ (K \ I) \ B)</td>
<td>(K \ x \ y \ = \ x)</td>
</tr>
<tr>
<td>4</td>
<td>(S \ (K \ S) \ A \ (S \ (K \ K) \ I) \ (K \ I) \ B)</td>
<td>(S \ x \ y \ z \ = \ x \ z \ (y \ z))</td>
</tr>
<tr>
<td>5</td>
<td>(K \ (K \ S) \ A \ (S \ (K \ K) \ I) \ (K \ I) \ B)</td>
<td>(K \ x \ y \ = \ x)</td>
</tr>
<tr>
<td>6</td>
<td>(S \ (K \ K) \ I \ A \ B)</td>
<td>(S \ x \ y \ z \ = \ x \ z \ (y \ z))</td>
</tr>
<tr>
<td>7</td>
<td>(K \ (K \ K) \ A \ (I \ B))</td>
<td>(K \ x \ y \ = \ x)</td>
</tr>
<tr>
<td>8</td>
<td>(A \ (K \ I) \ B)</td>
<td>(K \ x \ y \ = \ x)</td>
</tr>
<tr>
<td>9</td>
<td>(A \ (I \ B))</td>
<td>(I \ x \ = \ x)</td>
</tr>
<tr>
<td>10</td>
<td>(A \ B)</td>
<td>(A \ B)</td>
</tr>
</tbody>
</table>