Lecture 11

Theory of Combinators

- Combinators are an alternative theory of functions to the λ-calculus
- Originally introduced by logicians as a way of studying the process of substitution
- More recently, Turner has argued that combinators provide a good 'machine code' into which functional programs can be compiled
- Several experimental computers have been built based on Turner's ideas
- Combinators also provide a good intermediate code for conventional machines
  - several of the best compilers for functional languages are based on them

Formulations of theory of combinators

- **Two equivalent ways of formulating the theory of combinators:**

  (i) within the λ-calculus, or

  (ii) as a completely separate theory.

  - **approach (i) taken here**
  - **approach (ii) was the original one**

- **It will be shown that *any* λ-expression is equal to an expression built from variables and two particular expressions, `K` and `S`, using only function application**

- **This is done by mimicking λ-abstractions using combinations of `K` and `S`**

- **β-reductions can be simulated by simpler operations involving `K` and `S`**

  - **it is these simpler operations that combinator machines implement directly in hardware**

$\mathtt{S}$ and $\mathtt{K}$

- **The definitions of $\mathtt{K}$ and $\mathtt{S}$ are:**

$$\mathtt{LET}\ \mathtt{K} = \lambda x\ y.\ x$$

$$\mathtt{LET}\ \mathtt{S} = \lambda f\ g\ x.\ (f\ x)\ (g\ x)$$

- **By β-reduction, for all E_1, E_2 and E_3:**

$$\mathtt{K}\ E_1\ E_2 = E_1$$

$$\mathtt{S}\ E_1\ E_2\ E_3 = (E_1\ E_3)\ (E_2\ E_3)$$

Combinators

- **Any expression built by application (i.e. combination) from `K` and `S` is called a** *combinator*

  - **`K` and `S` are the** *primitive combinators*

- **Combinators have the following syntax:**

  <combinator> ::= `K` | `S` | (*<combinator>* *<combinator>*)

- **A** *combinatory expression* **is an expression built from `K`, `S` and zero or more variables**

  - **a combinator is a combinatory expression not containing variables**

- **Syntax of combinatory expressions:**

  <combinatory expression>
  ::= `K` | `S`
  | *<variable>*
  | (*<combinatory expression>* *<combinatory expression>*)

The identity combinator $\mathtt{I}$

- **The identity function $\mathtt{I}$ is often taken as a primitive combinator, but this is not necessary as it can be defined from $\mathtt{K}$ and $\mathtt{S}$**

- **Define $\mathtt{I}$ by:**

$$\mathtt{LET}\ \ \mathtt{I}\ =\ \lambda x.\ x$$

- **Then $\mathtt{I} = \mathtt{S}\ \mathtt{K}\ \mathtt{K}$**
  - **exercise!**

Combinator reduction

- **If E and E' are combinatory expressions then $E \underset{c}{\longrightarrow} E'$ means:**
  - $E \equiv E'$
  - **or E' can be got from E by a sequence of rewritings of the form:**

    (i) $\mathtt{K}\ E_1\ E_2 \underset{c}{\longrightarrow} E_1$

    (ii) $\mathtt{S}\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} (E_1\ E_3)\ (E_2\ E_3)$

    (iii) $\mathtt{I}\ E \underset{c}{\longrightarrow} E$

- **Example: for any E**

$$\begin{aligned} \mathtt{S}\ \mathtt{K}\ \mathtt{K}\ E &\underset{c}{\longrightarrow} \mathtt{K}\ E\ (\mathtt{K}\ E) && \textbf{by (ii)} \\ &\underset{c}{\longrightarrow} E && \textbf{by (i)} \end{aligned}$$

  - **thus (iii) is derivable from (i) and (ii)**

- **Any sequence of combinatory reductions can be expanded into a sequence of β-conversions**
  - $\mathtt{K}\ E_1\ E_2 \longrightarrow E_1$
  - $\mathtt{S}\ E_1\ E_2\ E_3 \longrightarrow (E_1\ E_3)\ (E_2\ E_3)$

Functional completeness

- **Every λ-expression is equal to some combinatory expression**
  - **called the functional completeness of combinators**
  - **basis for compilers for functional languages to the machine code of combinator machines**
- **Key idea:**
  - **for variable V and combinatory expression E a combinatory expression $\lambda^* V.\ E$ will be defined**
  - $\lambda^* V.\ E$ **uses** `K` **and** `S` **to simulate adding 'λV' to an expression**
  - $\lambda^* V.\ E \ = \ \lambda V.\ E$

Bracket abstraction $\lambda^* V.\ E$

- **If V a variable and E a combinatory expression, then $\lambda^* V.\ E$ is defined inductively on the structure of E as follows:**

  (i) $\lambda^* V.\ V = \mathtt{I}$

  (ii) $\lambda^* V.\ V' = \mathtt{K}\ V'$ **(if $V \neq V'$)**

  (iii) $\lambda^* V.\ C = \mathtt{K}\ C$ **(if C is a combinator)**

  (iv) $\lambda^* V.\ (E_1\ E_2) = \mathtt{S}\ (\lambda^* V.\ E_1)\ (\lambda^* V.\ E_2)$

- **Note that $\lambda^* V.\ E$ is a combinatory expression not containing V**

- **Example: if f and x are variables and $f \neq x$, then:**

$$\begin{aligned} \lambda^* x.\ \ f\ x &= \mathtt{S}\ (\lambda^* x.\ f)\ (\lambda^* x.\ x) \\ &= \mathtt{S}\ (\mathtt{K}\ f)\ \mathtt{I} \end{aligned}$$

Proof of functional completeness

- **THEOREM:**
  - $(\lambda^* V.\ E) = \lambda V.\ E$

- **PROOF:**
  - **show** $(\lambda^* V.\ E)\ V = E$
  - **follows immediately that** $\lambda V.\ (\lambda^* V.\ E)\ V = \lambda V.E$
  - **and hence by** η**-reduction that** $\lambda^* V.\ E = \lambda V.\ E$

Proof that $(\lambda^* V.\ E)\ V\ =\ E$

- **Mathematical induction on the 'size' of E:**

  (i) if $E = V$ then:

  $$(\lambda^* V.\ E)\ V\ =\ \mathtt{I}\ V\ =\ (\lambda x.\ x)\ V\ =\ V\ =\ E$$

  (ii) if $E = V'$ where $V' \neq V$ then:

  $$(\lambda^* V.\ E)\ V\ =\ \mathtt{K}\ V'\ V\ =\ (\lambda x\ y.\ x)\ V'\ V\ =\ V'\ =\ E$$

  (iii) if $E = C$ where C is a combinator, then:

  $$(\lambda^* V.\ E)\ V\ =\ \mathtt{K}\ C\ =\ (\lambda x\ y.\ x)\ C\ V\ =\ C\ =\ E$$

  (iv) if $E = (E_1\ E_2)$ then we can assume by induction that:

  $$(\lambda^* V.\ E_1)\ V = E_1$$
  $$(\lambda^* V.\ E_2)\ V = E_2$$

  and hence

  $$\begin{aligned}(\lambda^* V.\ E)\ V &= (\lambda^* V.\ (E_1\ E_2))\ V\\ &= (\mathtt{S}\ (\lambda^* V.\ E_1)\ (\lambda^* V.\ E_2))\ V\\ &= (\lambda f\ g\ x.\ f\ x\ (g\ x))\ (\lambda^* V.\ E_1)\ (\lambda^* V.\ E_2)\ V\\ &= (\lambda^* V.\ E_1)\ V\ ((\lambda^* V.\ E_2)\ V)\\ &= E_1\ E_2 \quad \textbf{(by induction assumption)}\\ &= E\end{aligned}$$

Translation to combinators

- **The notation**

$$\lambda^* V_1 \ V_2 \ \cdots \ V_n.\ E$$

is used to mean

$$\lambda^* V_1.\ \lambda^* V_2. \quad \cdots \ \lambda^* V_n.\ E$$

- **Define the translation of λ-expression E to a combinatory expression $(E)_{\mathtt{C}}$:**

(i) $(V)_{\mathtt{C}} = V$

(ii) $(E_1 \ E_2)_{\mathtt{C}} = (E_1)_{\mathtt{C}} \ (E_2)_{\mathtt{C}}$

(iii) $(\lambda V.\ E)_{\mathtt{C}} = \lambda^* V.\ (E)_{\mathtt{C}}$

$$\boxed{E = (E)_{\mathtt{C}}}$$

- **THEOREM:**
  - **for every λ-expression E we have:** $E = (E)_{\mathtt{C}}$

- **PROOF: induction on the size of E**

  (i) If $E = V$ then $(E)_{\mathtt{C}} = (V)_{\mathtt{C}} = V$

  (ii) If $E = (E_1\ E_2)$ we can assume by induction that

$$E_1 = (E_1)_{\mathtt{C}}$$
$$E_2 = (E_2)_{\mathtt{C}}$$

  hence

$$(E)_{\mathtt{C}} = (E_1\ E_2)_{\mathtt{C}} = (E_1)_{\mathtt{C}}\ (E_2)_{\mathtt{C}} = E_1\ E_2 = E$$

  (iii) If $E = \lambda V.\ E'$ then we can assume by induction that

$$(E')_{\mathtt{C}} = E'$$

  hence

$$\begin{aligned}(E)_{\mathtt{C}} &= (\lambda V.\ E')_{\mathtt{C}} \\ &= \lambda^* V.\ (E')_{\mathtt{C}} && \textbf{(by translation rules)} \\ &= \lambda^* V.\ E' && \textbf{(by induction assumption)} \\ &= \lambda V.\ E' && \textbf{(by previous theorem)} \\ &= E\end{aligned}$$

Consequences of last theorem

- **Every λ-expression is equal to a λ-expression built up from $\mathtt{K}$ and $\mathtt{S}$ and variables by application**

  - the class of λ-expressions E defined by:

    $$E \ ::= \ V \ \mid \ \mathtt{K} \ \mid \ \mathtt{S} \ \mid \ E_1 \ E_2$$

    is equivalent to the full λ-calculus

- **A collection of n combinators C_1, ... , C_n is called an n-element *basis***

  - if every λ-expression E is equal to an expression built from C_is and variables by function applications

  - theorem above shows $\mathtt{K}$ and $\mathtt{S}$ form a 2-element basis

- **There exists a 1-element basis!**

  Exercise
  Find a combinator, $\mathtt{X}$ say, such that any λ-expression is equal to an expression built from $\mathtt{X}$ and variables by application.
  Hint: Let $\langle E_1, E_2, E_3\rangle = \lambda p.\ p\ E_1\ E_2\ E_3$ and consider $\langle \mathtt{K}, \mathtt{S}, \mathtt{K}\rangle\ \langle \mathtt{K}, \mathtt{S}, \mathtt{K}\rangle\ \langle \mathtt{K}, \mathtt{S}, \mathtt{K}\rangle$ and $\langle \mathtt{K}, \mathtt{S}, \mathtt{K}\rangle\ \langle\langle \mathtt{K}, \mathtt{S}, \mathtt{K}\rangle\ \langle \mathtt{K}, \mathtt{S}, \mathtt{K}\rangle\rangle$

Examples

- **Part of $\mathtt{Y}$:**

$$
\begin{aligned}
&\lambda^* f.\ \lambda^* x.\ f\ (x\ x)\\
&\quad = \lambda^* f.\ (\lambda^* x.\ f\ (x\ x))\\
&\quad = \lambda^* f.\ (\mathtt{S}\ (\lambda^* x.\ f)\ (\lambda^* x.\ x\ x))\\
&\quad = \lambda^* f.\ (\mathtt{S}\ (\mathtt{K} f)\ (\mathtt{S}(\lambda^* x.\ x)\ (\lambda^* x.\ x)))\\
&\quad = \lambda^* f.\ (\mathtt{S}\ (\mathtt{K} f)\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))\\
&\quad = \mathtt{S}\ (\lambda^* f.\ \mathtt{S}\ (\mathtt{K} f))\ (\lambda^* f.\ \mathtt{S}\ \mathtt{I}\ \mathtt{I})\\
&\quad = \mathtt{S}\ (\mathtt{S}\ (\lambda^* f.\ \mathtt{S})\ (\lambda^* f.\ \mathtt{K}\ f))\ (\mathtt{K}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))\\
&\quad = \mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{S})\ (\mathtt{S}\ (\lambda^* f.\ \mathtt{K})\ (\ \lambda^* f.\ f)))\ (\mathtt{K}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))\\
&\quad = \mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{S})\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{K})\ \mathtt{I}))\ (\mathtt{K}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))
\end{aligned}
$$

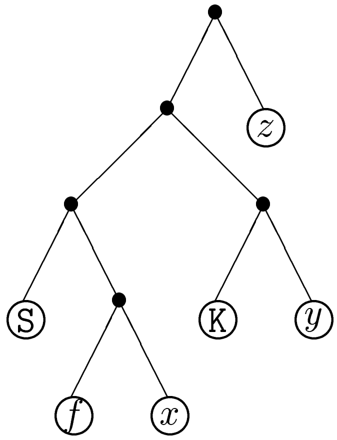
- $\mathtt{Y}$:

$$
\begin{aligned}
(\mathtt{Y})_{\mathtt{C}} &= (\lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))_{\mathtt{C}}\\
&= \lambda^* f.\ ((\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))_{\mathtt{C}}\\
&= \lambda^* f.\ ((\lambda x.\ f(x\ x))_{\mathtt{C}}\ (\lambda x.\ f(x\ x))_{\mathtt{C}})\\
&= \lambda^* f.\ (\lambda^* x.\ (f(x\ x))_{\mathtt{C}})\ (\lambda^* x.\ (f(x\ x))_{\mathtt{C}})\\
&= \lambda^* f.\ (\lambda^* x.\ f(x\ x))\ (\lambda^* x.\ f(x\ x))\\
&= \mathtt{S}\ (\lambda^* f.\ \lambda^* x.\ f(x\ x))\ (\lambda^* f.\ \lambda^* x.\ f(x\ x))\\
&= \mathtt{S(S(S(KS)(S(KK)I))(K(SII)))(S(S(KS)(S(KK)I))(K(SII)))}
\end{aligned}
$$

Reduction machines

- **Represent combinatory expressions by trees**

- **Example:** $\mathtt{S}\ (f\ x)\ (\mathtt{K}\ y)\ z$ **represented by:**

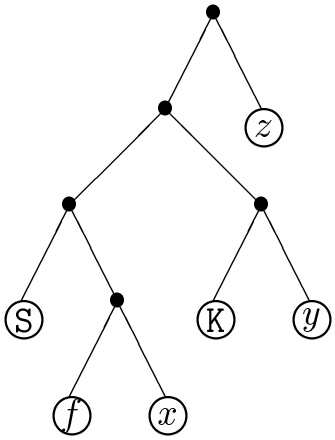


- **Such trees are represented as pointer structures in memory**

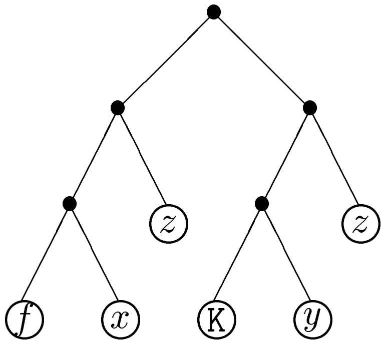
  - **special hardware or firmware can then be implemented to transform such trees according to the rules of combinator reduction defining** $\underset{c}{\longrightarrow}$

Examples of tree reduction

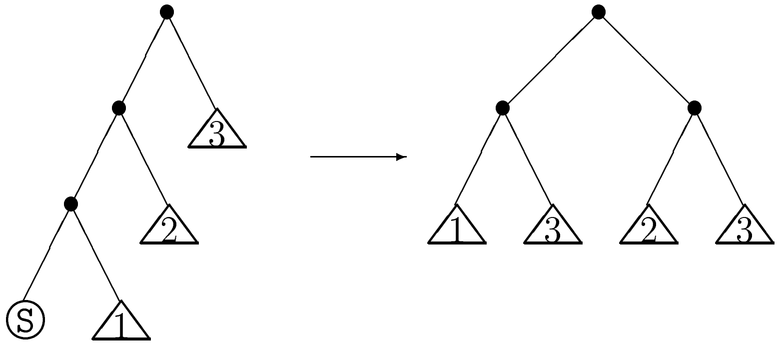
- **The tree:**



Could be transformed to:



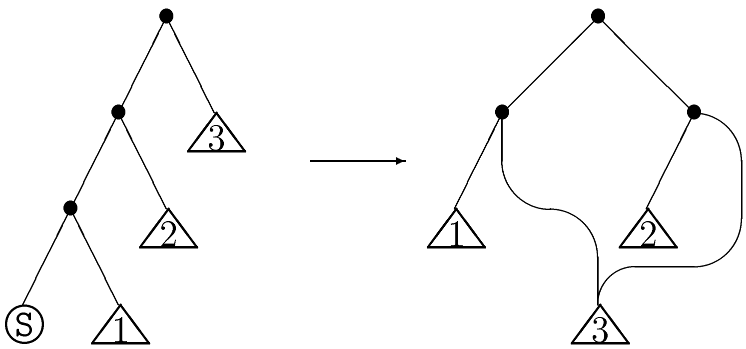
Using the transformation:



- **Implements:** $\mathtt{S}\ E_1\ E_2\ E_3 \underset{\mathrm{c}}{\longrightarrow} (E_1\ E_3)\ (E_2\ E_3)$

Graph reduction

- **Tree transformation for `S` just given duplicates a subtree**
  - **wastes space**
  - **a better transformation would be to generate one subtree with two pointers to it:**



- **Generates a *graph* rather than a tree**

Using combinators for evaluation

- **Valid way of reducing λ-expressions is:**

  (i) translating to combinators

  - **i.e.** $E \mapsto (E)_{\mathtt{C}}$

  (ii) applying the rewrites

$$\begin{array}{l} \mathtt{K}\ E_1\ E_2 \underset{c}{\longrightarrow} E_1 \\ \mathtt{S}\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} (E_1\ E_3)\ (E_2\ E_3) \end{array}$$

  until no more rewriting is possible

- **If $E_1 \longrightarrow E_2$ in the λ-calculus**

  - **then** *not* **necessarily** $(E_1)_{\mathtt{C}} \underset{c}{\longrightarrow} (E_2)_{\mathtt{C}}$

  - **for example, take**

$$E_1 = \lambda y.\ (\lambda z.\ y)\ (x\ y)$$

$$E_2 = \lambda y.\ y$$

Combinatory normal form

- **A combinatory expression is in** *combinatory normal form* **if it contains no subexpressions of the form** $\mathtt{K}\ E_1\ E_2$ **or** $\mathtt{S}\ E_1\ E_2\ E_3$

- **Normalization theorem holds for combinatory expressions**

    - **i.e. always reducing the leftmost combinatory redex will find a combinatory normal form if it exists**

- **If** E **is in combinatory normal form, then it does not necessarily follow that it is a** λ**-expression in normal form**

    - $\mathtt{S}\ \mathtt{K}$ **is in combinatory normal form, but it contains a** β**-redex, namely:**

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

Improving translation to combinators

- **Simple λ-expressions can translate to complex combinatory expressions**

- **To make the 'code' executed by reduction machines more compact, various optimizations have been devised**

- **Let E be a combinatory expression and x a variable not occurring in E**

  - **then:**

$$\mathtt{S}\ (\mathtt{K}\ E)\ \mathtt{I}\ x \underset{c}{\longrightarrow} (\mathtt{K}\ E\ x)\ (\mathtt{I}\ x) \underset{c}{\longrightarrow} E\ x$$

  - **hence** $\mathtt{S}\ (\mathtt{K}E)\ \mathtt{I}\ x = E\ x$ **(because** $E_1 \underset{c}{\longrightarrow} E_2$ **implies** $E_1 \longrightarrow E_2$**)**

  - **so by extensionality:**

$$\mathtt{S}\ (\mathtt{K}\ E)\ \mathtt{I} = E$$

- **Whenever $\mathtt{S}\ (\mathtt{K}\ E)\ \mathtt{I}$ is generated**

  - **it can be 'peephole optimized' to just E**

Another optimisation

- **Let E_1, E_2 be combinatory expressions and x a variable not occurring in either of them**

  - **then:**

$$\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2)\ x \underset{\mathrm{c}}{\longrightarrow} \mathtt{K}\ E_1\ x\ (\mathtt{K}\ E_2)\ x \underset{\mathrm{c}}{\longrightarrow} E_1\ E_2$$

  - **thus**

$$\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2)\ x \ = \ E_1\ E_2$$

  - **now**

$$\mathtt{K}\ (E_1\ E_2)\ x \ \underset{\mathrm{c}}{\longrightarrow}\ E_1\ E_2$$

  - **hence** $\mathtt{K}\ (E_1\ E_2)\ x = E_1\ E_2$

  - **thus**

$$\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2)\ x \ = \ E_1\ E_2 \ = \ \mathtt{K}\ (E_1\ E_2)\ x$$

  - **it follows by extensionality that:**

$$\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2) \ = \ \mathtt{K}\ (E_1\ E_2)$$

- **Whenever $\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2)$ is generated**

  - **it can be optimized to** $\mathtt{K}\ (E_1\ E_2)$

Example optimisation

- **Example: showed earlier that:**

  $\lambda^* f.\ \lambda^* x.\ f(x\ x) = \mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{S})\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{K})\ \mathtt{I}))\ (\mathtt{K}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))$

- **Using the optimization**

$$\mathtt{S}\ (\mathtt{K}\ E)\ \mathtt{I} = E$$

- **This simplifies to:**

$$\lambda^* f.\ \lambda^* x.\ f(x\ x) = \mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{S})\ \mathtt{K})\ (\mathtt{K}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))$$

More combinators

- **Easy to recognize applicability of optimization $\mathtt{S}\ (\mathtt{K}\ E)\ \mathtt{I} = E$ if $\mathtt{I}$ has not been expanded to $\mathtt{S}\ \mathtt{K}\ \mathtt{K}$**

  - **i.e. if $\mathtt{I}$ is taken as a primitive combinator**

- **Other combinators similarly useful**

- **Define $\mathtt{B}$ and $\mathtt{C}$ by:**

$$\mathtt{LET}\ \mathtt{B} = \lambda f\ g\ x.\ f\ (g\ x)$$

$$\mathtt{LET}\ \mathtt{C} = \lambda f\ g\ x.\ f\ x\ g$$

- **These have the following reduction rules:**

$$\begin{array}{l} \mathtt{B}\ E_1\ E_2\ E_3 \underset{\text{c}}{\longrightarrow} E_1\ (E_2\ E_3) \\ \mathtt{C}\ E_1\ E_2\ E_3 \underset{\text{c}}{\longrightarrow} E_1\ E_3\ E_2 \end{array}$$

- **It follows that:**

$$\begin{array}{l} \mathtt{S}\ (\mathtt{K}\ E_1)\ E_2 = \mathtt{B}\ E_1\ E_2 \\ \mathtt{S}\ E_1\ (\mathtt{K}\ E_2) = \mathtt{C}\ E_1\ E_2 \end{array}$$

(E_1, E_2 are any two combinatory expressions)

Curry's algorithm

- **Combining the various optimizations yields** *Curry's algorithm* **for translating λ-expressions to combinatory expressions**

- **Use the definition of** $(E)_{\mathtt{C}}$

- **Whenever an expression of the form** $\mathtt{S}\ E_1\ E_2$ **is generated, try to apply the following rewrite rules:**

  1. $\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2) \longrightarrow \mathtt{K}\ (E_1\ E_2)$

  2. $\mathtt{S}\ (\mathtt{K}\ E)\ \mathtt{I} \longrightarrow E$

  3. $\mathtt{S}\ (\mathtt{K}\ E_1)\ E_2 \longrightarrow \mathtt{B}\ E_1\ E_2$

  4. $\mathtt{S}\ E_1\ (\mathtt{K}\ E_2) \longrightarrow \mathtt{C}\ E_1\ E_2$

- **Always use earliest applicable rule**

- $\mathtt{S}\ (\mathtt{K}\ E_1)\ (\mathtt{K}\ E_2)$ **is translated to** $\mathtt{K}\ (E_1\ E_2)$

- $\mathtt{Y}$ **is translated to** $\mathtt{S}\ (\mathtt{C}\ \mathtt{B}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))\ (\mathtt{C}\ \mathtt{B}\ (\mathtt{S}\ \mathtt{I}\ \mathtt{I}))$