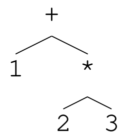
Chapter 1

λ-Calculus

This chapter presents the λ-*calculus*, a core calculus for functional languages (including SML of course). It captures the essential mechanism of computation in functional languages, and thus serves as an excellent framework for investigating basic concepts in functional languages. According to the Church-Turing thesis, the λ-calculus is equally expressive as Turing machines, but its syntax is deceptively simple. We first discuss the syntax and semantics of the λ-calculus and then show how to write programs in the λ-calculus.

Before we proceed, we briefly discuss the difference between *concrete syntax* and *abstract syntax*. Concrete syntax specifies which string of characters is accepted as a valid program (causing no syntax errors) or rejected as an invalid program (causing syntax errors). For example, according to the concrete syntax of SML, a string `~1` is interpreted as an integer -1, but a string `-1` is interpreted as an infix operator `-` applied to an integer argument `1` (which later causes a type error). A parser implementing concrete syntax usually translates source programs into tree structures. For example, a source program `1 + 2 * 3` is translated into



after taking into account operator precedence rules. Such tree structures are called *abstract syntax trees* which abstract away from details of parsing (such as operator precedence/associativity rules) and focus on the structure of source programs; abstract syntax is just the syntax for such tree structures.

While concrete syntax is an integral part of designing a programming language, we will not discuss it in this course. Instead we will work with abstract syntax to concentrate on computational aspects of programming languages. For example, we do not discuss why `1 + 2 * 3` and `1 + (2 * 3)`, both written in concrete syntax, are translated by the parser into the same abstract syntax tree shown above. For the purpose of understanding how their computation proceeds, the abstract syntax tree alone suffices.

1.1 Abstract syntax for the λ-calculus

The abstract syntax for the λ-calculus is given as follows:

$$\text{expression} \qquad e \quad ::= \quad x \mid \lambda x.\, e \mid e\; e$$

- An expression x is called a *variable*. We may use other names for variables (*e.g.*, z, s, t, f, arg, $accum$, and so on). Strictly speaking, therefore, x itself in the inductive definition of expression is a metavariable.
- An expression $\lambda x.\, e$ is called a λ-*abstraction*, or just a function, which denotes a mathematical function whose formal argument is x and whose body is e. We may think of $\lambda x.\, e$ as an internal representation of a nameless SML function `fn` x `=>` e in abstract syntax.

We say that a variable x is *bound* in the λ-abstraction $\lambda x.\, e$ (just like a variable x is bound in an SML function `fn x => e`). Alternatively we can say that x is a bound variable in the body e.

- An expression $e_1\ e_2$ is called an *application* which denotes a function application (if e_1 is shown to be equivalent to a λ-abstraction somehow). We may think of $e_1\ e_2$ as an internal representation of an SML function application in abstract syntax. As in SML, applications are left-associative: $e_1\ e_2\ e_3$ means $(e_1\ e_2)\ e_3$ instead of $e_1\ (e_2\ e_3)$.

As it turns out, every expression in the λ-calculus denotes a mathematical function. That is, the denotation of every expression in the λ-calculus is a mathematical function. Section 1.2 discusses how to determine unique mathematical functions corresponding to expressions in the λ-calculus, and in the present section, we develop the intuition behind the λ-calculus by considering a few examples of λ-abstractions.

Our first example is an identity function:

$$\mathsf{id} \quad = \quad \lambda x.\, x$$

id is an identity function because when given an argument x, it returns x without any further computation. Like higher-order functions in SML, a λ-abstraction may return another λ-abstraction as its result. For example, tt belows takes t to return another λ-abstraction $\lambda f.\, t$ which ignores its argument; ff below ignores its argument t to return a λ-abstraction $\lambda f.\, f$:

$$\begin{array}{lllll} \mathsf{tt} & = & \lambda t.\, \lambda f.\, t & = & \lambda t.\, (\lambda f.\, t) \\ \mathsf{ff} & = & \lambda t.\, \lambda f.\, f & = & \lambda t.\, (\lambda f.\, f) \end{array}$$

Similarly a λ-abstraction may expect another λ-abstraction as its argument. For example, the λ-abstraction below expects another λ-abstraction s which is later applied to z:

$$\lambda s.\, \lambda z.\, s\ z \quad = \quad \lambda s.\, (\lambda z.\, (s\ z))$$

1.2 Operational semantics of the λ-calculus

The semantics of a programming language answers the question of "what is the *meaning* of a given program?" This is an important question in the design of programming languages because lack of formal semantics implies potential ambiguities in interpreting programs. Put another way, lack of formal semantics makes it impossible to determine the meaning of certain programs. Surprisingly not every programming language has its semantics. For example (and perhaps to your surprise), the C language has no formal semantics — the same C program may exhibit different behavior depending on the state of the machine on which the program is executed.

There are three approaches to formulating the semantics of programming languages: *denotational semantics*, *axiomatic semantics*, and *operational semantics*. Throughout this course, we will use exclusively the operational semantics approach for its close connection with judgments and inference rules. The operational semantics approach is also attractive because it directly reflects the implementation of programming languages (*e.g.*, interpreters or compilers).

In general, the operational semantics of a programming language specifies how to transform a program into a *value* via a sequence of "operations." In the case of the λ-calculus, values consist of λ-abstractions and "operations" are called *reductions*. Thus the operational semantics of the λ-calculus specifies how to reduce an expression e into a value v where v is defined as follows:

$$\mathsf{value} \qquad v \quad ::= \quad \lambda x.\, e$$

Then we take v as the meaning of e. Since a λ-abstraction denotes a mathematical function, it follows that *every* expression in the λ-calculus denotes a mathematical function.

With this idea in mind, let us formally define reductions of expressions. We introduce a *reduction judgment* of the form $e \mapsto e'$:[1]

[1] After all, the notion of judgment that we learned in Chapter **??** is not really useless!

$$e \mapsto e' \qquad \Leftrightarrow \qquad e \textit{ reduces to } e'$$

We write $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$. That is, $e \mapsto^* e'$ holds if $e \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ where $n \geq 0$. We say that e *evaluates* to v if $e \mapsto^* v$ holds.

Before we provide inference rules to complete the definition of the judgment $e \mapsto e'$, let us see what kind of expression can be reduced to another expression. Clearly variables and λ-abstractions cannot be further reduced:

$$\begin{array}{rl} x & \not\mapsto \cdot \\ \lambda x.\, e & \not\mapsto \cdot \end{array}$$

($e \not\mapsto \cdot$ means that e does not reduce to another expression.) Then when can we reduce an application $e_1\ e_2$? If we think of it as an internal representation of an SML function application, we can reduce it only if e_1 represents an SML function. Thus the only candidate for reduction is an application of the form $(\lambda x.\, e_1')\ e_2$.

If we think of $\lambda x.\, e_1'$ as a mathematical function whose formal argument is x and whose body is e_1', the most natural way to reduce $(\lambda x.\, e_1')\ e_2$ is by substituting e_2 for every occurrence of x in e_1', or equivalently, by replacing every occurrence of x in e_1' by e_2. (For now, we do not consider the issue of whether e_2 is a value or not.) To this end, we introduce a *substitution* $[e'/x]e$:

$[e'/x]e$ *is defined as an expression obtained by substituting e' for every occurrence of x in e.*

$[e'/x]e$ may also be read as "applying a substitution $[e'/x]$ to e." Then the following reduction is justified

$$(\lambda x.\, e)\ e' \mapsto [e'/x]e$$

where the expression being reduced, namely $(\lambda x.\, e)\ e'$, is called a *redex* (*red*ucible *ex*pression). For historical reasons, the above reduction is called a *β-reduction*.

Simple as it may seem, the precise definition of $[e'/x]e$ is remarkably subtle (see Section 1.3). For now, we just avoid examples whose reduction would require the precise definition of substitution. Here are a few examples of β-reductions; the redex in each step is underlined:

$$\begin{array}{rclclcl}
\underline{(\lambda x.\, x)\ (\lambda y.\, y)} & \mapsto & \lambda y.\, y & & & & \\
\underline{(\lambda t.\, \lambda f.\, t)\ (\lambda x.\, x)}\ (\lambda y.\, y) & \mapsto & \underline{(\lambda f.\, \lambda x.\, x)\ (\lambda y.\, y)} & \mapsto & \lambda x.\, x & & \\
\underline{(\lambda t.\, \lambda f.\, f)\ (\lambda x.\, x)}\ (\lambda y.\, y) & \mapsto & \underline{(\lambda f.\, f)\ (\lambda y.\, y)} & \mapsto & \lambda y.\, y & & \\
\underline{(\lambda s.\, \lambda z.\, s\ z)\ (\lambda x.\, x)}\ (\lambda y.\, y) & \mapsto & \underline{(\lambda z.\, (\lambda x.\, x)\ z)\ (\lambda y.\, y)} & \mapsto & \underline{(\lambda x.\, x)\ (\lambda y.\, y)} & \mapsto & \lambda y.\, y
\end{array}$$

The β-reduction is the basic principle for reducing expressions, but it does not yield unique inference rules for the judgment $e \mapsto e'$. That is, there can be more than one way to apply the β-reduction to an expression, or equivalently, an expression may contain multiple redexes in it. For example, $(\lambda x.\, x)\ ((\lambda y.\, y)\ (\lambda z.\, z))$ contains two redexes in it:

$$\begin{array}{rclcl}
\underline{(\lambda x.\, x)\ ((\lambda y.\, y)\ (\lambda z.\, z))} & \mapsto & \underline{(\lambda y.\, y)\ (\lambda z.\, z)} & \mapsto & \lambda z.\, z \\
(\lambda x.\, x)\ (\underline{(\lambda y.\, y)\ (\lambda z.\, z)}) & \mapsto & \underline{(\lambda x.\, x)\ (\lambda z.\, z)} & \mapsto & \lambda z.\, z
\end{array}$$

In the first case, the expression being reduced has the form $(\lambda x.\, e)\ e'$ and we immediately apply the β-reduction to the whole expression to obtain $[e'/x]e$. In the second case, we apply the β-reduction to e' which happens to be a redex; if e' was not a redex (*e.g.*, $e' = \lambda t.\, t$), the second case would be impossible. In the course of reducing an expression to a value, we may have to apply the β-reduction many times. As we do not want to apply the β-reduction in an arbitrary way, we need a certain *reduction strategy* so as to apply the β-reduction in a systematic way.

In this course, we consider two reduction strategies: *call-by-name* and *call-by-value*. The call-by-name strategy always reduces the leftmost and outermost redex. To be specific, given an expression $e_1\ e_2$, it checks if e_1 is a λ-abstraction $\lambda x.\, e_1'$. If so, it applies the β-reduction to the whole expression to obtain $[e_2/x]e_1'$. Otherwise it attempts to reduce e_1 using the same reduction strategy without considering e_2; when e_1 later reduces to a value (which must be a λ-abstraction), it applies the β-reduction to the whole expression. Consequently the second subexpression in an application (*e.g.*, e_2 in $e_1\ e_2$) is never reduced.

The call-by-value strategy is similar to the call-by-name strategy, but it reduces the second subexpression in an application to a value v after reducing the first subexpression. Hence the call-by-value strategy applies the β-reduction to an application of the form $(\lambda x.\, e)\ v$ only. Note that neither strategy reduces expressions inside a λ-abstraction, which implies that values are not further reduced.

As an example, let us consider an expression $(\mathsf{id}_1\ \mathsf{id}_2)\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z))$ which reduces in different ways under the two reduction strategies; id_i is an abbreviation of an identity function $\lambda x_i.\, x_i$:

$$
\begin{array}{ll}
 & \text{call-by-name} \\
 & (\mathsf{id}_1\ \mathsf{id}_2)\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)) \\
\mapsto & \mathsf{id}_2\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)) \\
\mapsto & \mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z) \\
\mapsto & \lambda z.\, \mathsf{id}_4\ z
\end{array}
\qquad
\begin{array}{ll}
 & \text{call-by-value} \\
 & (\mathsf{id}_1\ \mathsf{id}_2)\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)) \\
\mapsto & \mathsf{id}_2\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)) \\
\mapsto & \mathsf{id}_2\ (\lambda z.\, \mathsf{id}_4\ z) \\
\mapsto & (\lambda z.\, \mathsf{id}_4\ z)
\end{array}
$$

The reduction diverges in the second step: the call-by-name strategy applies the β-reduction to the whole expression because it does not need to inspect the second subexpression $\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)$ whereas the call-by-value strategy chooses to reduce the second subexpression which is not a value yet.

Now we are ready to provide inference rules for the judgment $e \mapsto e'$, which we refer to as *reduction rules*. The call-by-name strategy uses two reduction rules:

$$
\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \qquad \frac{}{(\lambda x.\, e)\ e' \mapsto [e'/x]e}\ \mathit{App}
$$

The call-by-value strategy uses an additional rule to reduce second subexpression in applications; we reuse the reduction rule names from the call-by-name strategy:

$$
\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \quad \frac{e_2 \mapsto e_2'}{(\lambda x.\, e)\ e_2 \mapsto (\lambda x.\, e)\ e_2'}\ \mathit{Arg} \quad \frac{}{(\lambda x.\, e)\ v \mapsto [v/x]e}\ \mathit{App}
$$

We say that an expression is in *normal form* if no reduction rule is applicable. Clearly every value (which is a λ-abstraction in the case of the λ-calculus) is in normal form. There are, however, expressions in normal form that are not values. For example, $x\ \lambda y.\, y$ is in normal form because x cannot be further reduced, but it is not a value either. We say that such expression is *stuck* or its reduction gets *stuck*. A stuck expression may be thought of as an ill-formed program, and ideally should not arise during an evaluation. Chapter **??** presents an extension of the λ-calculus which statically (*i.e.*, at compile time) guarantees that a program satisfying a certain criterion never gets stuck.

A drawback of the call-by-name strategy is that the same expression may be evaluated multiple times. For example, $(\lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda z.\, z))$ evaluates $(\lambda y.\, y)\ (\lambda z.\, z)$ to $\lambda z.\, z$ eventually twice:

$$
\begin{array}{ll}
 & (\lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda z.\, z)) \\
\mapsto & ((\lambda y.\, y)\ (\lambda z.\, z))\ ((\lambda y.\, y)\ (\lambda z.\, z)) \\
\mapsto & (\lambda z.\, z)\ ((\lambda y.\, y)\ (\lambda z.\, z)) \\
\mapsto & (\lambda y.\, y)\ (\lambda z.\, z) \\
\mapsto & \lambda z.\, z
\end{array}
$$

On the other hand, the call-by-name strategy never evaluates expressions that do not contribute to evaluations. For example,

$$
(\lambda t.\, \lambda f.\, f)\ ((\lambda y.\, y)\ (\lambda z.\, z))\ ((\lambda y'.\, y')\ (\lambda z'.\, z'))
$$

does not evaluate $(\lambda y.\, y)\ (\lambda z.\, z)$ at all because it is not used in the evaluation.

The call-by-name strategy is adopted by the functional language Haskell. Haskell is called a *lazy* or *non-strict* functional language because it evaluates arguments to functions only if necessary (*i.e.*, "lazily"). The actual implementation of Haskell uses another reduction strategy called *call-by-need*, which is semantically equivalent to the call-by-name strategy but never evaluates the same expression more than once. The call-by-value strategy is adopted by SML which is called an *eager* or *strict* functional language because it always evaluates arguments to functions regardless of whether they are actually used in function bodies (*i.e.*, "eagerly").

1.3 Substitution

This section presents a precise definition of substitution $[e'/x]e$ to complete the operational semantics of the λ-calculus. While an informal interpretation of $[e'/x]e$ is obvious, its formal definition is a lot trickier than it appears.

First we need the notion of *free* variable which is the opposite of the notion of bound variable and plays a key role in the definition of substitution. A free variable is a variable which is not bound in any enclosing λ-abstraction. For example, y in $\lambda x.\, y$ is a free variable because no λ-abstraction of the form $\lambda y.\, e$ encloses its occurrence. To formalize the notion of free variable, we introduce a mapping $FV(e)$ to mean the set of free variables in e:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x.\, e) & = & FV(e) - \{x\} \\ FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \end{array}$$

Since a variable is either free or bound, a variable $x \notin FV(e)$ must be bound somewhere in e if it appears in e. We say that an expression is *closed* if it contains no free variables.

A substitution $[e/x]e'$ is defined inductively with the following cases:

$$\begin{array}{rcll} [e/x]x & = & e & \\ [e/x]y & = & y & \textit{if } x \neq y \\ [e/x](e_1\ e_2) & = & [e/x]e_1\ [e/x]e_2 & \end{array}$$

In order to give a precise definition of the remaining case $[e/x]\lambda y.\, e'$, we need to understand two properties of variables. The first property is that the name of a bound variable does not matter, which also conforms to our intuition. For example, an identity function $\lambda x.\, x$ inside an expression e may be rewritten as $\lambda y.\, y$ for an arbitrary variable y without changing the intended meaning of e. Formally we use a judgment $e \equiv_\alpha e'$ to mean that e can be rewritten as e' by renaming bound variables in e (which does not change the meaning of e), or vice versa. By a historical accident, $\equiv_\alpha$ is called the *α-equivalence relation*, or we say that an *α-conversion* of e into e' rewrites e as e' by renaming bound variables in e. It turns out the a precise definition of $e \equiv_\alpha e'$ is also considerably tricky, which is given at the end of the present section.

The first property justifies the following case of substitution:

$$[e'/x]\lambda x.\, e \quad = \quad \lambda x.\, e$$

Intuitively, if we rewrite $\lambda x.\, e$ as another λ-abstraction of the form $\lambda y.\, e''$ where y is a fresh variable such that $x \neq y$, the substitution $[e'/x]$ is effectively ignored because x is found nowhere in $\lambda y.\, e''$. A generalization of the case is that $[e'/x]$ has no effect on e if x is a bound variable in e, or equivalently, if x is not a free variable in e:

$$[e'/x]e \quad = \quad e \qquad \textit{if } x \notin FV(e)$$

That is, we want to apply $[e'/x]$ to e only if x is a free variable in e.

The second property is that a free variable x in an expression e never turns into a bound variable; when explicitly replaced by another expression e', as in $[e'/x]e$, it simply disappears. To better understand the second property, let use consider a naive definition of $[e'/x]\lambda y.\, e$ (which turns out to be wrong):

$$[e'/x]\lambda y.\, e \quad = \quad \lambda y.\, [e'/x]e \qquad \textit{if } x \neq y$$

Now, a free variable y in e' automatically becomes a bound variable in $\lambda y.\, [e'/x]e$, which is not acceptable. Here is an example showing such an anomaly:

$$(\lambda x.\, \lambda y.\, x)\ y \mapsto [y/x]\lambda y.\, x = \lambda y.\, [y/x]x = \lambda y.\, y$$

Before the substitution, $\lambda y.\, x$ is a λ-abstraction that ignores its argument and returns x, but after the substitution, it turns into an identity function! What happens in the example is that a free variable y to be substituted for x is supposed to remain free after the substitution, but is accidentally *captured* by the λ-abstraction

$\lambda y.\, x$ and becomes a bound variable. Such a phenomenon is called a *variable capture* which destroys the intuition that a free variable remains free unless it is replaced by another expression, and thus $[y/x]\lambda y.\, x$ is not defined. This observation is generalized in the following definition of $[e'/x]\lambda y.\, e$ which is called a *capture-avoiding substitution*:

$$[e'/x]\lambda y.\, e \quad = \quad \lambda y.\, [e'/x]e \qquad \textit{if } x \neq y, y \notin FV(e')$$

When a variable capture occurs in $[e'/x]\lambda y.\, e$, we implicitly apply an α-conversion to rename y to another variable that is not free in e. For example, $(\lambda x.\, \lambda y.\, x)\ y$ can be reduced by renaming the bound variable y to a fresh variable z so as to avoid a variable capture:

$$(\lambda x.\, \lambda y.\, x)\ y \mapsto [y/x]\lambda y.\, x \equiv_\alpha [y/x]\lambda z.\, x = \lambda z.\, y$$

In the literature, the unqualified term "substitution" universally means a capture-avoiding substitution that renames bound variables as necessary.

Finally we give a precise definition of the judgment $e \equiv_\alpha e'$. We need the notion of *variable swapping* $[x \leftrightarrow y]e$ which is obtained by replacing *all* occurrences of x in e by y and *all* occurrences of y in e by x. We emphasize that "all" occurrences include even those next to λ in λ-abstractions, which makes it straightforward to implement $[x \leftrightarrow y]e$. Here is an example:

$$[x \leftrightarrow y]\lambda x.\, \lambda y.\, x\ y \quad = \quad \lambda y.\, \lambda x.\, y\ x$$

The definition of $e \equiv_\alpha e'$ is given inductively by the following inference rules:

$$\frac{}{x \equiv_\alpha x}\ \mathit{Var}_\alpha \qquad \frac{e_1 \equiv_\alpha e_1' \quad e_2 \equiv_\alpha e_2'}{e_1\ e_2 \equiv_\alpha e_1'\ e_2'}\ \mathit{App}_\alpha$$

$$\frac{e \equiv_\alpha e'}{\lambda x.\, e \equiv_\alpha \lambda x.\, e'}\ \mathit{Lam}_\alpha \qquad \frac{x \neq y \quad y \notin FV(e) \quad [x \leftrightarrow y]e \equiv_\alpha e'}{\lambda x.\, e \equiv_\alpha \lambda y.\, e'}\ \mathit{Lam}_\alpha'$$

The rule Lam_α says that to compare $\lambda x.\, e$ and $\lambda x.\, e'$ which bind the same variable, we compare their bodies e and e'. To compare two λ-abstractions binding different variables, we use the rule Lam_α'.

To see why the rule Lam_α' works, we need to understand the implication of the premise $y \notin FV(e)$. Since $y \notin FV(e)$ implies $y \notin FV(\lambda x.\, e)$ and we have $x \notin FV(\lambda x.\, e)$, an outside observer would notice no difference even if the two variables x and y were literally swapped in $\lambda x.\, e$. In other words, $\lambda x.\, e$ and $[x \leftrightarrow y]\lambda x.\, e$ are effectively the same from the point of view of an outside observer. Since $[x \leftrightarrow y]\lambda x.\, e = \lambda y.\, [x \leftrightarrow y]e$, we compare $[x \leftrightarrow y]e$ with e', which is precisely the third premise in the rule Lam_α'.

1.4 Programming in the λ-calculus

In order to develop the λ-calculus to a full-fledged functional language, we need to show how to encode common datatypes such as boolean values, integers, and lists in the λ-calculus. Since all values in the λ-calculus are λ-abstractions, all such datatypes are also encoded with λ-abstractions. Once we show how to encode specific datatypes, we may use them as if they were built-in datatypes.

1.4.1 Church booleans

The inherent capability of a boolean value is to choose one of two different options. For example, a boolean truth chooses the first of two different options, as in an SML expression `if true then` e_1 `else` e_2. Thus boolean values in the λ-calculus, called Church booleans, are written as follows:

$$\begin{array}{rcl} \mathsf{tt} & = & \lambda t.\, \lambda f.\, t \\ \mathsf{ff} & = & \lambda t.\, \lambda f.\, f \end{array}$$

Then a conditional construct $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ is defined as follows:

$$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \quad = \quad e\ e_1\ e_2$$

Here are examples of reducing conditional constructs under the call-by-name strategy:

$$\begin{array}{l} \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{tt}\ e_1\ e_2 = \underline{(\lambda t.\, \lambda f.\, t)\ e_1}\ e_2 \mapsto \underline{(\lambda f.\, e_1)\ e_2} \mapsto e_1 \\ \mathsf{if}\ \mathsf{ff}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{ff}\ e_1\ e_2 = \underline{(\lambda t.\, \lambda f.\, f)\ e_1}\ e_2 \mapsto \underline{(\lambda f.\, f)\ e_2} \mapsto e_2 \end{array}$$

Logical operators on boolean values are defined as follows:

$$\begin{array}{rcl} \mathsf{and} & = & \lambda x.\, \lambda y.\, x\ y\ \mathsf{ff} \\ \mathsf{or} & = & \lambda x.\, \lambda y.\, x\ \mathsf{tt}\ y \\ \mathsf{not} & = & \lambda x.\, x\ \mathsf{ff}\ \mathsf{tt} \end{array}$$

Exercise 1.1. Consider the conditional construct $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ defined as $e\ e_1\ e_2$ under the call-by-value strategy. How is it different from the conditional construct in SML?

Exercise 1.2. Define the logical operator xor. An easy way to define it is to use a conditional construct and the logical operator not.

1.4.2 Pairs

The inherent capability of a pair is to carry two unrelated values and to retrieve either value when requested. Thus, in order to represent a pair of e_1 and e_2, we build a λ-abstraction which returns e_1 and e_2 when applied to tt and ff, respectively. Projection operators treat a pair as a λ-abstraction and applies it to either tt or ff.

$$\begin{array}{rcl} \mathsf{pair} & = & \lambda x.\, \lambda y.\, \lambda b.\, b\ x\ y \\ \mathsf{fst} & = & \lambda p.\, p\ \mathsf{tt} \\ \mathsf{snd} & = & \lambda p.\, p\ \mathsf{ff} \end{array}$$

Exercise 1.3. Show the reduction sequence of fst $(\mathsf{pair}\ e_1\ e_2)$ under the call-by-name strategy.

1.4.3 Church numerals

The inherent capability of a natural number n is to repeat a given process n times. In the case of the λ-calculus, n is encoded as a λ-abstraction $\hat{n}$, called a Church numeral, that takes a function f and returns $f^n = f \circ f \cdots \circ f$ (n times). Note that f^0 is an identity function $\lambda x.\, x$ because f is applied 0 times to its argument x, and that $\hat{1}$ itself is an identity function $\lambda f.\, f$.

$$\begin{array}{rcccl} \hat{0} & = & \lambda f.\, f^0 & = & \lambda f.\, \lambda x.\, x \\ \hat{1} & = & \lambda f.\, f^1 & = & \lambda f.\, \lambda x.\, f\ x \\ \hat{2} & = & \lambda f.\, f^2 & = & \lambda f.\, \lambda x.\, f\ (f\ x) \\ \hat{3} & = & \lambda f.\, f^3 & = & \lambda f.\, \lambda x.\, f\ (f\ (f\ x)) \\ & & \cdots & & \\ \hat{n} & = & \lambda f.\, f^n & = & \lambda f.\, \lambda x.\, f\ (f\ (f\ \cdots (f\ x) \cdots)) \end{array}$$

If we read f as `succ` and x as `zero`, $\hat{n}\ f\ x$ returns the representation of the natural number n shown in Chapter **??**.

Now let us define arithmetic operations on natural numbers. The addition operation add $\hat{m}\ \hat{n}$ returns $\widehat{m+n}$ which is a λ-abstraction taking a function f and returning f^{m+n}. Since f^{m+n} may be written as $\lambda x.\, f^{m+n}\ x$, we develop add as follows; in order to differentiate natural numbers (*e.g.*, n) from their encoded form (*e.g.*, $\hat{n}$), we use $\hat{m}$ and $\hat{n}$ as variables:

$$\begin{array}{rcl} \mathsf{add} & = & \lambda \hat{m}.\, \lambda \hat{n}.\, \lambda f.\, f^{m+n} \\ & = & \lambda \hat{m}.\, \lambda \hat{n}.\, \lambda f.\, \lambda x.\, f^{m+n}\ x \\ & = & \lambda \hat{m}.\, \lambda \hat{n}.\, \lambda f.\, \lambda x.\, f^m\ (f^n\ x) \\ & = & \lambda \hat{m}.\, \lambda \hat{n}.\, \lambda f.\, \lambda x.\, \hat{m}\ f\ (\hat{n}\ f\ x) \end{array}$$

Note that f^m is obtained as $\hat{m}\ f$ (and similarly for f^n).

Exercise 1.4. Define the multiplication operation $\mathsf{mult}\ \hat{m}\ \hat{n}$ which returns $\widehat{m * n}$.

The multiplication operation can be defined in two ways. An easy way is to exploit the equation $m * n = m + m + \cdots + m$ (n times). That is, $m * n$ is obtained by adding m to zero exactly n times. Since $\mathsf{add}\ \hat{m}$ is conceptually a function adding m to its argument, we apply $\mathsf{add}\ \hat{m}$ to $\hat{0}$ exactly n times to obtain $\widehat{m * n}$, or equivalently apply $(\mathsf{add}\ \hat{m})^n$ to $\hat{0}$:

$$\begin{array}{rcl} \mathsf{mult} & = & \lambda \hat{m}.\, \lambda \hat{n}.\, (\mathsf{add}\ \hat{m})^n\ \hat{0} \\ & = & \lambda \hat{m}.\, \lambda \hat{n}.\, \hat{n}\ (\mathsf{add}\ \hat{m})\ \hat{0} \end{array}$$

An alternative way (which may in fact be easier to figure out than the first solution) is to exploit the equation $f^{m*n} = (f^m)^n = (\hat{m}\ f)^n = \hat{n}\ (\hat{m}\ f)$:

$$\mathsf{mult} \quad = \quad \lambda \hat{m}.\, \lambda \hat{n}.\, \lambda f.\, \hat{n}\ (\hat{m}\ f)$$

The subtraction operation is more difficult to define than the previous two operations. Suppose that we have a predecessor function pred computing the predecessor of a given natural number: $\mathsf{pred}\ \hat{n}$ returns $\widehat{n-1}$ if $n > 0$ and $\hat{0}$ otherwise. To define the subtraction operation $\mathsf{sub}\ \hat{m}\ \hat{n}$ which returns $\widehat{m-n}$ if $m > n$ and $\hat{0}$ otherwise, we apply pred to $\hat{m}$ exactly n times:

$$\begin{array}{rcl} \mathsf{sub} & = & \lambda \hat{m}.\, \lambda \hat{n}.\, \mathsf{pred}^n\ \hat{m} \\ & = & \lambda \hat{m}.\, \lambda \hat{n}.\, (\hat{n}\ \mathsf{pred})\ \hat{m} \end{array}$$

Exercise 1.5. Define the predecessor function pred. Use an idea similar to the one used in a tail-recursive implementation of the Fibonacci function.

The predecessor function pred uses an auxiliary function next which takes $\mathsf{pair}\ \hat{k}\ \hat{m}$, ignores $\hat{k}$, and returns $\mathsf{pair}\ \hat{m}\ \widehat{m+1}$. By applying next to $\mathsf{pair}\ \hat{0}\ \hat{0}$ exactly n times, we obtain $\mathsf{pair}\ \widehat{n-1}\ \hat{n}$ if $n > 0$. Since the predecessor of 0 is 0 anyway, the first component of $\mathsf{next}^n\ (\mathsf{pair}\ \hat{0}\ \hat{0})$ encodes the predecessor of n. Thus pred is defined as follows:

$$\begin{array}{rcl} \mathsf{next} & = & \lambda p.\, \mathsf{pair}\ (\mathsf{snd}\ p)\ (\mathsf{add}\ (\mathsf{snd}\ p)\ \hat{1}) \\ \mathsf{pred} & = & \lambda \hat{n}.\, \mathsf{fst}\ (\mathsf{next}^n\ (\mathsf{pair}\ \hat{0}\ \hat{0})) \\ & = & \lambda \hat{n}.\, \mathsf{fst}\ (\hat{n}\ \mathsf{next}\ (\mathsf{pair}\ \hat{0}\ \hat{0})) \end{array}$$

Exercise 1.6. Define a function $\mathsf{isZero} = \lambda \hat{n}. \cdots$ which tests if a given Church numeral is $\hat{0}$. Use it to define another function $\mathsf{eq} = \lambda \hat{m}.\, \lambda \hat{n}. \cdots$ which tests if two given Church numerals are equal.

1.5 Fixed point combinator

Since the λ-calculus is equally powerful as Turing machines, every Turing machine can be simulated by a certain expression in the λ-calculus. In particular, there are expressions in the λ-calculus that correspond to Turing machines that do not terminate and Turing machines that compute recursive functions.

It is relatively easy to find an expression whose reduction does not terminate. Suppose that we wish to find an expression omega such that $\mathsf{omega} \mapsto \mathsf{omega}$. Since it reduces to the same expression, its reduction never terminates. We rewrite omega as $(\lambda x.\, e)\ e'$ on the assumption that the β-reduction is applied to the whole expression omega. Then we have

$$(\lambda x.\, e)\ e' \mapsto [e'/x]e = (\lambda x.\, e)\ e'.$$

If we let $e = e''\ x$, we get

$$[e'/x](e''\ x) = [e'/x]e''\ e' = (\lambda x.\, e''\ x)\ e'.$$

By letting $e'' = x$, we obtain $e' = \lambda x.\, x\ x$. The resultant expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, abbreviated as omega, is an example whose reduction never terminates.

Then how do we write recursive functions in the λ-calculus? We begin by assuming a *recursive function construct* $\mathsf{fun}\ f\ x.\, e$ which defines a recursive function f whose argument is x and whose body is e. Note that the body e may contain references to f. Our goal is to show that $\mathsf{fun}\ f\ x.\, e$ is *syntactic sugar* (which dissolves in the λ-calculus) in the sense that it can be rewritten as an existing expression in the λ-calculus and thus its addition does not increase the expressive power of the λ-calculus.

As a working example, we use a factorial function fac:

$$\mathsf{fac} \quad = \quad \mathsf{fun}\ f\ n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (f\ (\mathsf{pred}\ n))$$

Semantically f in the body refers to the very function fac being defined. First we mechanically derive a λ-abstraction $\mathsf{FAC} = \lambda f.\, \lambda n.\, e$ from $\mathsf{fac} = \mathsf{fun}\ f\ n.\, e$:

$$\mathsf{FAC} \quad = \quad \lambda f.\, \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (f\ (\mathsf{pred}\ n))$$

Note that FAC has totally different characteristics than fac: while fac takes a natural number n to return another natural number, FAC takes a function f to return another function. (If fac and FAC were allowed to have types, fac would have type $\mathsf{nat} \rightarrow \mathsf{nat}$ whereas FAC would have type $(\mathsf{nat} \rightarrow \mathsf{nat}) \rightarrow (\mathsf{nat} \rightarrow \mathsf{nat})$.)

The key idea behind constructing FAC is that given a *partial* implementation f of the factorial function, $\mathsf{FAC}\ f$ returns an *improved* implementation of the factorial function. Suppose that f correctly computes the factorial of any natural number up to n. Then $\mathsf{FAC}\ f$ correctly computes the factorial of any natural number up to $n+1$, which is an improvement over f. Note also that $\mathsf{FAC}\ f$ correctly computes the factorial of 0 regardless of f. In particular, even when given a least informative function $f = \lambda n.\, \mathsf{omega}$ (which does nothing because it never returns), $\mathsf{FAC}\ f$ correctly computes the factorial of 0. Thus we can imagine an infinite chain of functions $\{\mathsf{fac}_0, \mathsf{fac}_1, \cdots, \mathsf{fac}_i, \cdots\}$ which begins with $\mathsf{fac}_0 = \mathsf{FAC}\ \lambda n.\, \mathsf{omega}$ and repeatedly applies the equation $\mathsf{fac}_{i+1} = \mathsf{FAC}\ \mathsf{fac}_i$:

$$\begin{array}{rclcl}
\mathsf{fac}_0 & = & & & \mathsf{FAC}\ \lambda n.\, \mathsf{omega} \\
\mathsf{fac}_1 & = & \mathsf{FAC}\ \mathsf{fac}_0 & = & \mathsf{FAC}^2\ \lambda n.\, \mathsf{omega} \\
\mathsf{fac}_2 & = & \mathsf{FAC}\ \mathsf{fac}_1 & = & \mathsf{FAC}^3\ \lambda n.\, \mathsf{omega} \\
 & \vdots & & & \\
\mathsf{fac}_i & = & \mathsf{FAC}\ \mathsf{fac}_{i-1} & = & \mathsf{FAC}^{i+1}\ \lambda n.\, \mathsf{omega} \\
 & \vdots & & &
\end{array}$$

Note that fac_i correctly computes the factorial of any natural number up to i. Then, if ω denotes an infinite natural number (greater than any natural number), we may take fac_ω as a correct implementation of the factorial function fac, *i.e.*, $\mathsf{fac} = \mathsf{fac}_\omega$.

Another important observation is that given a correct implementation fac of the factorial function, $\mathsf{FAC}\ \mathsf{fac}$ returns another correct implementation of the factorial function, That is, if fac is a correct implementation of the factorial function,

$$\lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (\mathsf{fac}\ (\mathsf{pred}\ n))$$

is also a correct implementation of the factorial function. Since the two functions are essentially identical in that both return the same result for any argument, we may let $\mathsf{fac} = \mathsf{FAC}\ \mathsf{fac}$. If we substitute fac_ω for fac in the equation, we obtain $\mathsf{fac}_\omega = \mathsf{fac}_{\omega+1}$ which also makes sense because $\omega \leq \omega + 1$ by the definition of $+$ and $\omega + 1 \leq \omega$ by the definition of ω (which is greater than any natural number including $\omega + 1$).

Now it seems that FAC contains all necessary information to derive $\mathsf{fac} = \mathsf{fac}_\omega = \mathsf{FAC}\ \mathsf{fac}$, but exactly how? It turns out that fac is obtained by applying the *fixed point combinator* fix to FAC, *i.e.*, $\mathsf{fac} = \mathsf{fix}\ \mathsf{FAC}$, where fix is defined as follows:

$$\mathsf{fix} \quad = \quad \lambda F.\, (\lambda f.\, F\ (\lambda x.\, f\ f\ x))\ (\lambda f.\, F\ (\lambda x.\, f\ f\ x))$$

Here we assume the call-by-value strategy; for the call-by-name strategy, we simplify $\lambda x.\, f\ f\ x$ into $f\ f$ and use the following fixed point combinator $\mathsf{fix}_{\mathsf{CBN}}$:

$$\mathsf{fix}_{\mathsf{CBN}} \quad = \quad \lambda F.\, (\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f))$$

To understand how the fixed point combinator fix works, we need to learn the concept of *fixed point*.[2] A fixed point of a function f is a value v such that $v = f(v)$. For example, the fixed point of a function $f(x) = 2 - x$ is 1 because $1 = f(1)$. As its name suggests, fix takes a function F (which itself transforms a function f into another function f') and returns its fixed point. That is, $\mathsf{fix}\ F$ is a fixed point of F:

$$\mathsf{fix}\ F \quad = \quad F\ (\mathsf{fix}\ F)$$

Informally the left expression transforms into the right expression via the following steps; we use a symbol $\approx$ to emphasize "informally" because the transformation is not completely justified by the β-reduction alone:

$$\begin{array}{lcll}
\mathsf{fix}\ F & \mapsto & g\ g & \textit{where}\ g = \lambda f.\, F\ (\lambda x.\, f\ f\ x) \\
& \mapsto & F\ (\lambda x.\, g\ g\ x) & \\
& \approx & F\ (g\ g) & \textit{because}\ \lambda x.\, g\ g\ x \approx g\ g \\
& \approx & F\ (\mathsf{fix}\ F) & \textit{because}\ \mathsf{fix}\ F \mapsto g\ g
\end{array}$$

Now we can explain why $\mathsf{fix}\ \mathsf{FAC}$ gives an implementation of the factorial function. By the nature of the fixed point combinator fix, we have

$$\mathsf{fix}\ \mathsf{FAC} \quad = \quad \mathsf{FAC}\ (\mathsf{fix}\ \mathsf{FAC}).$$

That is, $\mathsf{fix}\ \mathsf{FAC}$ returns a function f satisfying $f = \mathsf{FAC}\, f$, which is precisely the property that fac needs to satisfy! Therefore we take $\mathsf{fix}\ \mathsf{FAC}$ as an equivalent of fac.[3]

An alternative way to explain the behavior of $\mathsf{fix}\ \mathsf{FAC}$ is as follows. Suppose that we wish to compute $\mathsf{fac}\ n$ for an arbitrary natural number n. Since $\mathsf{fix}\ \mathsf{FAC}$ is a fixed point of FAC, we have the following equation:

$$\begin{array}{lclcl}
\mathsf{fix}\ \mathsf{FAC} & = & & & \mathsf{FAC}\ (\mathsf{fix}\ \mathsf{FAC}) \\
& = & \mathsf{FAC}\ (\mathsf{FAC}\ (\mathsf{fix}\ \mathsf{FAC})) & = & \mathsf{FAC}^2\ (\mathsf{fix}\ \mathsf{FAC}) \\
& = & \mathsf{FAC}^2\ (\mathsf{FAC}\ (\mathsf{fix}\ \mathsf{FAC})) & = & \mathsf{FAC}^3\ (\mathsf{fix}\ \mathsf{FAC}) \\
& \vdots & & & \\
& = & \mathsf{FAC}^n\ (\mathsf{FAC}\ (\mathsf{fix}\ \mathsf{FAC})) & = & \mathsf{FAC}^{n+1}\ (\mathsf{fix}\ \mathsf{FAC})
\end{array}$$

The key observation is that $\mathsf{FAC}^{n+1}\ (\mathsf{fix}\ \mathsf{FAC})$ correctly computes the factorial of any natural number up to n *regardless of what* $\mathsf{fix}\ \mathsf{FAC}$ *does* (see Page 9). Since we have $\mathsf{fix}\ \mathsf{FAC} = \mathsf{FAC}^{n+1}\ (\mathsf{fix}\ \mathsf{FAC})$, it follows that $\mathsf{fix}\ \mathsf{FAC}$ correctly computes the factorial of an *arbitrary* natural number. That is, $\mathsf{fix}\ \mathsf{FAC}$ does precisely what fac does.

In summary, in order to encode a recursive function $\mathsf{fun}\ f\ x.\, e$ in the λ-calculus, we first derive a λ-abstraction $F = \lambda f.\, \lambda x.\, e$. Then $\mathsf{fix}\ F$ auto*magically* returns a function that exhibits the same behavior as $\mathsf{fun}\ f\ x.\, e$ does.

Exercise 1.7. Under the call-by-value strategy, fac, or equivalently $\mathsf{fix}\ \mathsf{FAC}$, never terminates when applied to any natural number! Why?

1.6 Deriving the fixed point combinator

This section explains how to derive the fixed point combinator. As its formal derivation is extremely intricate, we will illustrate the key idea with an example. Students may choose to skip this section if they wish.

[2] Never use the word *fixpoint*! Dana Scott, who coined the word *fixed point*, says that *fixpoint* is wrong!

[3] The fixed point combinator fix actually yields what is called the *least* fixed point. That is, a function F may have many fixed points and fix returns the least one in the sense that the least one is the most informative one. The least fixed point is what we usually expect.

Let us try to write a factorial function fac without using the fixed point combinator. Consider the following function $\mathsf{fac_{wrong}}$:

$$\mathsf{fac_{wrong}} \quad = \quad \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (f\ (\mathsf{pred}\ n))$$

$\mathsf{fac_{wrong}}$ is simply wrong because its body contains a reference to an unbound variable f. If, however, f points to a correct implementation fac of the factorial function, $\mathsf{fac_{wrong}}$ would also be a correct implementation. Since there is no way to use a free variable f in reducing an expression, we have to introduce it in a λ-abstraction anyway:

$$\mathsf{FAC} \quad = \quad \lambda f.\, \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (f\ (\mathsf{pred}\ n))$$

FAC is definitely an improvement over $\mathsf{fac_{wrong}}$, but it is not a function taking a natural number; rather it takes a function f to return another function which refines f. More importantly, there seems to be no way to make a recursive call with FAC because FAC calls only its argument f in its body and never makes a recursive call to itself.

Then how do we make a recursive call with FAC? The problem at hand is that the body of FAC, which needs to call fac, calls only its argument f. Our instinct, however, says that FAC contains all necessary information to derive fac (*i.e.*, FAC $\approx$ fac) because its body resembles a typical implementation of the factorial function. Thus we are led to try substituting FAC itself for f. That is, we make a call to FAC using FAC itself as an argument — what a crazy idea it is!:

$$\mathsf{FAC}\ \mathsf{FAC} \quad = \quad \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (\mathsf{FAC}\ (\mathsf{pred}\ n))$$

Unfortunately FAC FAC returns a function which does not make sense: in its body, a call to FAC is made with an argument $\mathsf{pred}\ n$, but FAC expects not a natural number but a function. It is, however, easy to fix the problem: if FAC FAC returns a correct implementation of the factorial function, we only need to replace FAC in the body by FAC FAC. That is, what we want in the end is the following equation

$$\mathsf{FAC}\ \mathsf{FAC} \quad = \quad \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (\mathsf{FAC}\ \mathsf{FAC}\ (\mathsf{pred}\ n))$$

where FAC FAC serves as a correct implementation of the factorial function.

Let us change the definition of FAC so that it satisfies the above equation. All we need to do is to replace a reference to f in its body by an application $f\ f$. Thus we obtain a new function Fac defined as follows:

$$\mathsf{Fac} \quad = \quad \lambda f.\, \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (f\ f\ (\mathsf{pred}\ n))$$

It is easy to see that Fac satisfies the following equation:

$$\mathsf{Fac}\ \mathsf{Fac} \quad = \quad \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (\mathsf{Fac}\ \mathsf{Fac}\ (\mathsf{pred}\ n))$$

Since Fac Fac returns a correct implementation of the factorial function, we define fac as follows:

$$\mathsf{fac} \quad = \quad \mathsf{Fac}\ \mathsf{Fac}$$

Now let us derive the fixed point combinator fix by rewriting fac in terms of fix (and FAC as it turns out). Consider the body of Fac:

$$\mathsf{Fac} \quad = \quad \lambda f.\, \underline{\lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (f\ f\ (\mathsf{pred}\ n))}$$

The underlined expression is almost the body of a typical factorial function except for the application $f\ f$. The following definition of Fac abstracts from the application $f\ f$ by replacing it by a reference to a single function g:

$$\begin{array}{rcl} \mathsf{Fac} & = & \lambda f.\, \underline{(\lambda g.\, \lambda n.\, \mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n\ (g\ (\mathsf{pred}\ n)))}\ (f\ f) \\ & = & \lambda f.\, \underline{\mathsf{FAC}}\ (f\ f) \end{array}$$

Then fac is rewritten as follows:

$$\begin{array}{rcl} \mathsf{fac} & = & \mathsf{Fac}\ \mathsf{Fac} \\ & = & (\lambda f.\, \mathsf{FAC}\ (f\ f))\ (\lambda f.\, \mathsf{FAC}\ (f\ f)) \\ & = & \lambda F.\, ((\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f)))\ \mathsf{FAC} \\ & = & \mathsf{fix}_{\mathsf{CBN}}\ \mathsf{FAC} \end{array}$$

In the case of the call-by-value strategy, $\mathsf{fix}_{\mathsf{CBN}}$ FAC always diverges. A quick fix is to rewrite $f\ f$ as $\lambda x.\, f\ f\ x$ and we obtain fix:

$$\begin{array}{rcl} \mathsf{fac} & = & \mathsf{Fac}\ \mathsf{Fac} \\ & = & \lambda F.\, ((\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f)))\ \mathsf{FAC} \\ & = & \lambda F.\, ((\lambda f.\, F\ (\lambda x.\, f\ f\ x))\ (\lambda f.\, F\ (\lambda x.\, f\ f\ x)))\ \mathsf{FAC} \\ & = & \mathsf{fix}\ \mathsf{FAC} \end{array}$$

This is how to derive the fixed point combinator fix!