The Lambda Calculus
Outline of Lectures

by Mayer Goldberg

August 29, 2014

Abstract

The following presentation outlines the lectures introducing the λ-calculus in the *Advanced Programming Languages* course, at the Department of Computer Science at Ben-Gurion University. Compiling these notes takes up a great deal of my time, and though I try to keep them in sync with the lectures, this is a failing proposition as they are *always* behind. Still, they will be updated frequently throughout the semester, and should help you study the material at home. *They are not a substitute for attendence.*

If you notice any errors or omissions, please let me know and I will update the document. The document changes often, and you can always download the latest version of this outline at `http://lambda.little-lisper.org/`.

Contents

1 Introduction to the Lambda Calculus (§1–20)

1. The foundations of mathematics takes a dual view of functions. On the one hand, we view functions as sets that map members of a domain-set to members of a range-set. On the other hand, we also view functions as computational objects that tell us how to find the specific element in the range that is matched with a given element in the domain.

2. The formal theory that characterises those functions that can be computed, how to define them, the power of expression that is required in order to define various functions, and what are the theoretical limits of this ability, is known as *computability theory.*

Figure 1: Alonzo Church

3. The languages used within computability theory to define various functions are known as *computational formalisms.* These include Turing machines, Post machines, the λ-calculus, etc.

4. The λ-calculus is a computational formalism that is at the heart of functional programming. We open the course with a study of the λ-calculus, and use it as a playground in which to explore ideas related to recursion theory, computability, type theory, and programming.

5. The λ-calculus was invented by the logician *Alonzo Church* (June 14, 1903 – August 11, 1995), as an alternative foundation for mathematics. Several contradictions that were discovered in its original formulation ended that project. After removing the parts that were responsible for the contradictions, what remains of the λ-calculus is a wonderful vehicle for the study of recursion theory, computability and programming languages.

6. Several programming languages were modeled after the λ-calculus. You will commonly see programming languages described as "an applicative-order λ-calculus with primitive types [and side effects]". One of these languages is Scheme. Another is ML. You will also commonly see the λ-calculus described as a kind of "pure Scheme/ML".[1]

7. Since most students in this course have at least two semesters of programming in Scheme and one in ML, we will use Scheme and ML for intuition as we pick up the λ-calculus.

8. Scheme can be characterized as a dynamic semifunctional programming language with run-time types. "Semifunctional" means that it includes side effects.

9. We will also be using the ML programming language in this course, as part of a general shift to that language. ML can be characterized as a static semi-functional programming language with a powerful static type inference system, and a very powerful module system.

[1] This perverse reasoning is exemplified by Homer Simpson, who upon seeing fish swimming in a pond, comments: *"Mmmm... Unprocessed fish sticks!"* (*The Simpsons*, Season 7, Episode 6: Treehouse of Horrors VI, Homer[3]).

10. ML has many advantages over Scheme, the most important of which is its static type system. For now, please take the following statement for granted: The type system in ML is so powerful, that quite often, once your program compiles without generating any warnings it will run bug-free. This is very different from Scheme, where only the most minor syntactic errors are detected during compilation, and all semantic errors are discovered (if at all) during run-time. After you have suffered the overhead of learning ML, you will be amply rewarded by having to spend far less time debugging your programs.

11. Returning to the λ-calculus, you will often be asked "to run things" or "to make sure they work", which means writing Scheme or ML code that will perform the computation described by the expressions we will study in the λ-calculus.

12. Formal definitions for the terms we use will be given later. Right now, in order to motivate the study of the λ-calculus, we will skip all the formal stuff, and jump right in, with minimal notation and no formalism, just so that we experience what the λ-calculus is all about.

13. Just as programming languages contain expressions and statements, the λ-calculus contains *terms*. Terms in the λ-calculus consist of *variables*, *applications*, and *abstractions*. Abstractions are an extension of the notion of a *function*.

14. Variables in the λ-calculus correspond roughly to variables in Scheme/ML, with the exception that there are no side effects in the λ-calculus. An implication of this is that a *store* is not needed in order to associate variables with values, and the substitution model suffices. A *store* is just a fancy word for an *environment*.

15. Applications in the λ-calculus correspond to applications in Scheme, with the following exceptions:

    - Applications in the λ-calculus are to a single argument only. If P, Q are λ-expressions, then (PQ) is the application of P to Q.
    - Expressions that look like they are applications to more than a single argument are really left-associated applications: $(PQRT)$ is really an abbreviation for $(((PQ)R)T)$: The λ-term P takes a single argument Q, and evaluates to an abstraction over a single variable, and is then applied to the single argument R. This, in turn, evaluates to an abstraction over a single variable, and is applied to the single argument T.

16. Abstractions in the λ-calculus correspond to simple λ-expressions in Scheme that take a single argument. If ν is a variable, and E is a λ-expression, then $(\lambda\nu.E)$ is roughly the same as `(lambda (`ν`) `E`)` in Scheme. Please note the following:

    - Abstractions in the λ-calculus that look like abstractions over more than a single argument are really nested abstractions: $(\lambda abc.E)$ is really an abbreviation for $(\lambda a.(\lambda b.(\lambda c.E)))$.
    - Because applications in the λ-calculus are left-associative, an expression like $F \equiv (\lambda abc.E)$ is applied to A, B, C as follows:

$$FABC \quad \equiv \quad (((FA)B)C)$$

Figure 2: Haskell Brooks Curry

- Encoding n-ary functions in the λ-calculus is typically done by using n nested λ-abstractions. The rationale behind this is as follows: we think of a function as a set of ordered pairs of *domain*$\times$*range*. A structure of the form $(\mathfrak{D}_1 \times \mathfrak{D}_2 \cdots \mathfrak{D}_n) \times \mathfrak{R}$ is isomorphic to $\mathfrak{D}_1 \times (\mathfrak{D}_2 \cdots (\mathfrak{D}_n \times \mathfrak{R}) \cdots)$, which is a single-arity function that takes an argument from $\mathfrak{D}_1$, and returns a single-arity function that takes an argument from $\mathfrak{D}_2$, etc., until finally we get a single-arity function that takes an argument from $\mathfrak{D}_n$ and returns a value from $\mathfrak{R}$. This process is known as *Currying*, after the logician *Haskell Brooks Curry* (September 12, 1900—September 1 1982)[2].

17. When writing λ-expressions, we use parenthesis to disambiguate expressions. We use them as little as possible, and only when necessary. This is quite unlike the situation in Scheme, where parenthesis denote application and are never used for disambiguation.

18. **Example:** The expression $\lambda abc.b(abc)$ would be written in Scheme as follows:

```
(lambda (a)
  (lambda (b)
    (lambda (c)
      (b ((a b) c)))))
```

In ML, the same expression would be written as

```
(fn a =>
  (fn b =>
    (fn c => b (a b c))))
```

If you are willing to give a name to the above λ-expression, then you can define a Curried function without having to specify three nested abstractions:

```
fun foo a b c = b (a b c)
```

Indeed, defining functions in ML is a lot of `fun` ☺

[2]Haskell B. Curry is the inventor of *combinatory logic*, which turned out to be an alternative formulation of the λ-calculus. The core ideas of combintory logic were discovered independently, and slightly before Curry, by *Moses Schönfinkel*.

Figure 3: Moses Ilyich Schönfinkel

19. Conspicuously absent from the language of λ-terms are primitive types (numbers, characters, Booleans, etc), and any kind of aggregation (strings, pairs, vectors, etc). This is intentional. The pure λ-calculus is minimalist, and consists only of variables, applications and abstractions. In the context of programming languages, people often speak of "the λ-calculus with constants" or "the λ-calculus with pairs". As we shall soon see, all these can be *modelled* within the λ-calculus. The reasons for adding, rather than modelling these extensions include:

    - Convenience
    - Explicit support in the type system. (we shall consider type systems for the λ-calculus later on in the course)
    - Explicit support for these types in a compiler
        - Efficiency of implementation & storage
        - Compiler optimizations

20. The syntax of the λ-calculus will be defined much more rigorously later on. In the meantime, we will rely on Scheme and ML for intuition, and move on to a demonstration as to what we can do with these λ-terms.

2 Lambda-Definability (§21–89)

21. The next items deal with a topic called λ-*definability*. Given a computational model, be it Turing machines or the λ-calclulus, definability is an area that deals with how to define the class of recursive functions in that model. Since our term language doesn't include numerals, or any arithmetical functions on them, we would like to define λ-terms that

    - model the numerals
    - model the basic number-theoretic functions on these numerals

22. Numerals provide a *representation* for numbers. Numbers are abstract Platonic entities, and are independent of any concrete representation. Numerals represent numbers in some way. Arithmetical functions are also Platonic entities, and map numbers to numbers. It is possible to model arithmetic functions over the integers using computer programs, Turing machines, and λ-expressions that operate on numerals in different representations. For any function $f : \mathfrak{D} \to \mathfrak{R}$, and pair of values $x, y \in \mathfrak{D} \times \mathfrak{R}$, a corresponding computer program, Turing machine or λ-expression would "take in", in some sense, a representation of x, and "put out", in that sense, a representation of y. The specific details of how numbers and other data are represented, how computational entities "take in" their input, and how they "put out" the values they compute, will all have to be defined precisely. Computational entities that are defined to work with specific representations (e.g., natural numbers represented in unary) are unlikely to perform the same computation on different representations (e.g., natural numbers represented in binary).

23. **Exercise 1:** Define a procedure in Scheme, that will perform the same computation regardless of the representation of its arguments. The procedure must use its argument in some way, i.e., the variable name that corresponds to that argument must occur freely within the body of the procedure.

24. **Exercise 2:** Show there are countably-many, different procedures in Scheme, that satisfy the conditions in § 23.

25. If we wish to change the representation of our numbers, i.e., if we wish to switch to a different numeral system, we will need to worry about how to redefine the arithmetical functions over the new numeral system. An easy way of doing so is to define terms that convert between the two numeral systems, but this is not always the best or most efficient way to approach the problem.

26. **Exercise 3:** Consider the following representation for the natural numbers: The number $n \in \mathbb{N}$ is represented by the function $\mathfrak{n} : \mathbb{N} \to \{0, 1\}$, and b_k is the k-th bit in the binary expansion of n, starting from the lower bit, so that $\mathfrak{n}(k) = b_k$. Implement [in your favourite functional programming language] a procedure that computes the addition function over these numerals. In other words, write the [higher-order] function Sum, such that for any $a, b \in \mathbb{N}$, with corresponding functions $\mathfrak{a}, \mathfrak{b} : \mathbb{N} \to \{0, 1\}$, we have $\mathsf{Sum}(\mathfrak{a}, \mathfrak{b})(k) = b_k$, where b_k is the k-th bit in the binary expansion of $a + b$, starting from the lower bit.

    This exercise should help you appreciate the difference between a function on the Platonic integers and a corresponding function on a representation of the Platonic integers.

27. **Exercise 4:** Implement in Java the addition function described in § 26. Be sure to define function objects that answer to a `valueAt` message.

28. Had we had some representation for the *zero* and for the *successor function* on that zero, in which z was our representation for zero, and s was our representation for the successor, then we could model the natural numbers as follows:

$$
\begin{gathered}
z \\
sz \\
s(sz) \\
\cdots \\
\underbrace{s(\cdots(s}_{n \text{ times}}\, z)\cdots)
\end{gathered}
$$

29. The most familiar numerals for the λ-calculus are the Church numerals (named after the above-mentioned Alonzo Church). Church's idea for defining numerals was to *abstract* over s, z in the above applications:

$$\begin{array}{rl} \lambda sz.z & \text{will represent the number 0} \\ \lambda sz.sz & \text{will represent the number 1} \\ \lambda sz.s(sz) & \text{will represent the number 2} \\ \lambda sz.s(s(sz)) & \text{will represent the number 3} \\ \lambda sz.s(s(s(sz))) & \text{will represent the number 4} \end{array}$$

and in general

$$\lambda sz.\underbrace{s(\cdots(s}_{n \text{ times}}\, z)\cdots) \quad \text{will represent the number } n$$

30. We denote the n-th Church numerals, i.e., Church's representation for the number n, by the symbol c_n.

31. In the previous items, we used the notion of abstraction as a mechanism for generalisation. There are different forms of abstraction in mathematics and programming. In the λ-calculus and functional programming, abstraction refers to parameterisation of an expression by a sub-expression.

    - The result of abstraction in the λ-calculus and functional programming is a *higher-order* function, i.e., a function the domain or range of which can include functions.
    - As an aside, one may also consider object-oriented abstraction, which is a way of generalising objects into the classes of which these objects are instances. Object-oriented and functional abstraction have so much in common that all the examples in this document can be written in an object-oriented language just as easily as they can be written in a functional language.

32. The variables s, z, in the λ-abstraction of the numeral c_n can be renamed to f, x,[3] giving

$$c_n \quad \equiv \quad \lambda fx.\underbrace{f(\cdots(f}_{n \text{ times}}\, x)\cdots)$$

With these variable names, it is easier to notice that the n-th Church numeral is really an abstraction over the n-th composition of some function. Put otherwise, the n-th Church numeral takes some function as an argument, and returns its n-th composition:

$$f \quad \stackrel{c_n}{\Longrightarrow} \quad f^n$$

A more colloquial notation for c_n could therefore be: $c_n \equiv \lambda f.f^n$.

33. Had we had definitions s, z for a successor function and a zero, then

$$c_n sz \quad = \quad (\underbrace{s(\cdots}_{n \text{ times}}\, z)\cdots)$$

[3] We will discuss this renaming later on, when we consider how to define formally the rules for renaming. The renaming step is called α-conversion, terms that are equivalent up to renaming of bound variables are said to be α-equivalent.

would give the number n. Therefore

$$s(c_n sz) \quad = \quad (\underbrace{s(s(\cdots z)}_{n+1 \text{ times}}\cdots))$$

It follows that $c_{n+1}sz = s(c_n sz)$. We now abstract s, z over both sides, and get that

$$\lambda sz.c_{n+1}sz \quad = \quad \lambda sz.s(c_n sz)$$

The left-hand side is just c_{n+1}.[4] We therefore have

$$c_{n+1} \quad = \quad \lambda sz.s(c_n sz)$$

Notice that the right-hand side of the equation contains c_n. We will now abstract over it: The function that takes c_n as an argument, and returns the expression on the right-hand side of the above equation is the *successor* function, and we define it by abstraction of c_n:

$$S^+ \quad \equiv \quad \lambda nsz.s(nsz)$$

The successor function takes c_n and returns c_{n+1}.

34. **Exercise 5:** Verify that $S^+c_2 = c_3$.

35. **Exercise 6:** Find a different expression that computes the successor function on Church numerals.

36. Since $c_n f = f^n$, it follows that $c_3 S^+ = (S^+)^3$, or the third composition of S^+, which is the function $\lambda n.S^+(S^+(S^+n))$, or the function that takes c_n and returns c_{n+3}, or in other words, the function that adds 3 to its argument. Therefore, it follows that $c_3 S^+ c_4 = S^+(S^+(S^+c_4)) = c_7$. In general $c_b S^+ c_a = c_{a+b}$. We define addition by abstracting a, b over the last expression:

$$+ \quad \equiv \quad \lambda ab.bS^+a$$

37. **Exercise 7:** Verify that $(+\ c_3\ c_4) = c_7$.

38. **Exercise 8:** Prove that $(+\ c_a\ c_b) = (+\ c_b\ c_a)$.

39. **Exercise 9:** Find two other definitions for addition on Church numerals.

40. Recall that λ-abstractions are Curried. This means that $+$ is a function of a single argument a, that returns a function of a single argument b, that returns $a+b$. Therefore, it makes sense to speak of the function $(+\ c_a)$, which happens to be the function that adds a to its [Church-numeral] argument. The b-th composition of the function $(+\ c_a)$ is given by $(c_b\ (+\ c_a))$, which is the function that takes a Church-numeral, and adds a to it, repeatedly, b times. If we apply this function to c_0, we get $(c_b\ (+\ c_a)\ c_0) = c_{a\times b}$. We define the multiplication function by abstracting a, b over the last expression:

$$\times \quad \equiv \quad \lambda ab.b(+\ a)c_0$$

[4] Actually, this is a non-trivial step: The rule that allows us to replace $\lambda x.fx$ with f, when x is not a free variable in f, is taken as an axiomatic rule in the equational theory with which we are working. It is quite possible to consider a version of the λ-calculus without this rule. We will have more to say about this rule later on in the course (§ 105), when we introduce the λ-calculus more formally, but for now, we just mention that this rule is called the η-rule (pronounced "eta-rule"). In the meantime, you can get an intuitive grasp of the η-rule if you think about the corresponding situation in Scheme: The η-rule in Scheme asserts, for example, that the `sin` and `(lambda (x) (sin x))` procedures are essentially the same thing.

41. **Exercise 10:** Find two other definitions for multiplication on Church numerals.

42. **Exercise 11:** Prove that $(\times\ c_a\ c_b) = (\times\ c_b\ c_a)$.

43. Similarly, the expression $(\times\ c_a)$ is the function that multiplies its argument by a. The b-th composition of $(\times\ c_a)$, takes a single argument, and multiplies it b times repeatedly, by a, effectively multiplying it by a^b. If we apply this function to c_1, we get $(c_b\ (\times\ c_a)\ c_1) = c_{a^b}$. We define the exponentiation function by abstracting a, b over the last expression:

$$\uparrow \quad \equiv \quad \lambda ab.b(\times\ a)c_1$$

44. **Exercise 12:** What is the value of $(\uparrow\ \ c_0\ c_0)$? Since the value of 0^0 is undefined in mathematics, the expression $\uparrow$ need not behave, in this special case, like the exponential function on the natural numbers. Still, you may be able to rationalise the result (no pun intended).

45. **Exercise 13:** Show that $(c_b c_a) = c_{a^b}$. Use this to derive an alternate definition for exponentiation.

46. *Ackermann's function.* We defined the multiplication function on c_a, c_b, by taking the b-th composition of $(+\ c_a)$. This is the equivalent of computing $\underbrace{c_a + \cdots + c_a}_{b \text{ times}}$.

    We defined the exponentiation function on c_a, c_b, by taking the b-th composition of $(\times\ c_a)$. This is the equivalent of computing $\underbrace{c_a \times \cdots \times c_a}_{b \text{ times}}$.

    Similarly, we can continue this process by defining the "super-exponentiation" (denoted by $\uparrow^{[2]}$) on c_a, c_b, by taking the b-th composition of $(\uparrow\ \ c_a)$. This is the equivalent of computing $\underbrace{c_a \uparrow \cdots \uparrow c_a}_{b \text{ times}}$.

    Similarly, we can then define "super-super-exponentiation" (denoted by $\uparrow^{[3]}$) on c_a, c_b, by taking the b-th composition of $(\uparrow^{[2]}\ \ c_a)$. This is the equivalent of computing $\underbrace{c_a \uparrow^{[2]} \cdots \uparrow^{[2]} c_a}_{b \text{ times}}$.

    We can go on defining faster-growing functions indefinitely.

    The $n+1$-st generalization of this process of defining functions is:

$$c_a \uparrow^{[n+1]} c_b \quad = \quad \underbrace{c_a \uparrow^{[n]} \cdots \uparrow^{[n]} c_a}_{b \text{ times}}$$

    If you were to compute these functions in some programming language, you would require a `for`-loop (or some other mechanism for *bounded iteration*) in order to compute the $n+1$-st function in using the n-th function. To compute the n-th function from the first function, you would therefore require n nested loops. A noble goal would be to define a function such as $\uparrow^{[2]}, \uparrow^{[3]}$, etc, in which n were a parameter rather than a constant. This would be an interesting function because it would not be expressible using a finite number of bounded iterations (or `for`-loops) over first-order functions (i.e., functions from $\mathbb{N} \to \mathbb{N}$).

Figure 4: Kurt Gödel

Figure 5: Wilhelm Friedrich Ackermann

The class of functions that can be computed using bounded iteration of bounded depth over the class of functions defined on integers and ordered tuples is known as the *primitive recursive functions,* and was formally defined by the Austrian-American logician *Kurt Gödel* (April 28, 1906 – January 14, 1978).

The first function that was shown *not* to be primitive recursive is known as *Ackermann's function*, after the German mathematician *Wilhelm Friedrich Ackermann* (March 29, 1896 – December 24, 1962), who came up with the first formulation of this function. As we shall see, Ackermann's function is very similar to $\lambda ab.(\uparrow^{[a]} b\ b)$.

Ackermann's original formulation of the function takes 3 integer arguments. The definition was simplified by the Hungarian mathematician *Rózsa Péter* (February 17, 1905 – February 16, 1977), into the 2-argument version that grows similarly, is not primitive recursive, and is the function that is commonly known today as *Ackermann's function,* or as it is sometimes more

Figure 6: Rózsa Péter

graciously referred to, the *Ackermann-Péter function*:

$$\begin{aligned} Ack(0,b) &= b+1 \\ Ack(a+1,0) &= Ack(a,1) \\ Ack(a+1,b+1) &= Ack(a, Ack(a+1,b)) \end{aligned}$$

The first argument of Ackermann's function determines the depth of the nesting, or the value of n in $\uparrow^{[n]}$. To see this, we expand the definition of Ack along the *second* argument:

$$\begin{aligned} Ack(5,3) &= Ack(4, Ack(5,2)) \\ &= Ack(4, Ack(4, Ack(5,1))) \\ &= Ack(4, Ack(4, Ack(4, Ack(5,0))))) \\ &= Ack(4, Ack(4, Ack(4, Ack(4,1))))) \end{aligned}$$

This would become considerably neater with a *Curried* version of Ackermann's function: $A_a(b) = Ack(a,b)$:

$$\begin{aligned} Ack(5,3) &\equiv A_5(3) \\ &= A_4(A_5(2)) \\ &= A_4(A_4(A_5(1))) \\ &= A_4(A_4(A_4(A_5(0)))) \\ &= A_4(A_4(A_4(A_4(1)))) \end{aligned}$$

In other words, $A_{a+1}(b)$ equals the $b+1$-st composition of A_a applied to 1. So A_{a+1} iterates over A_a the same way $\uparrow^{[n+1]}$ iterates over $\uparrow^{[n]}$.

Defining Ackermann's function using Church numerals turns out to be straightforward, because of the way c_n can be used to obtain the n-th composition of some function. Here is how to go about it:

Suppose we have A_a for some value of a. We can use it to define A_{a+1}, as follows:

$$A_{a+1} = \lambda b.S^+ b A_a c_1$$

Figure 7: Hendrik Pieter Barendregt

By abstracting over A_a, we define the function f that takes A_a for some value of a, and returns A_{a+1}:

$$f \quad = \quad \lambda Ab.S^+bAc_1 \tag{1}$$

We notice that $A_0 = S^+$, since $Ack(0, b) = b + 1$. We can now define Ackermann's function as taking two Church numerals c_a, c_b, applying the a-th composition of f to S^+ and b:

$$\begin{aligned} \mathbf{Ack} \quad &= \quad \lambda ab.afS^+b \\ &\longrightarrow_\eta \quad \lambda a.afS^+ \end{aligned}$$

Having stated that Ackermann's function was the first function that was shown not to be definable using a finite number of nested `for`-loops over first-order functions, we have just defined it using Church numerals as our mechanism for bounded iteration. It is therefore worthwhile to reflect for a moment on just how we accomplished this feat. Consider again the definition for Ackermann's function using Church numerals:

$$\mathbf{Ack} \quad = \quad \lambda a.\boxed{af}S^+$$

where f is defined in Eq (1). The parameter a to **Ack** is some Church numeral that is applied to the function f (the application is boxed above). But what kind of function is f? — The function f maps $A_n \to A_{n+1}$ for any n. The function A_n is a function that takes a single integer argument and returns an integer (recall that $A_a(b) = Ack(a, b)$). Therefore the type of A_n is $\mathbb{N} \to \mathbb{N}$, that is, A_n is a first-order function. This means, however, that the type of f is $((\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}))$, that is, f is a *second-order function*, that is, a function that mapes *functions from integers to integers*, to *functions from integers to integers*. That Ackermann's function is not primitive recursive means that it cannot be defined using a finite number of bounded iterations over *first-order functions*. What we have shown through our construction is precisely that Ackermann's functions is *second-order primitive recursive*.

47. **Exercise 14:** (Barendregt) Suppose the width of each character is 5 millimeters. Define a λ-expression of length 30 centimeters or shorter (meaning 60

symbols or less) that evaluates to a finite λ-expression of length $10^{10^{10^{10^{10}}}}$ kilometers or longer.

48. **Exercise 15:** Define the λ-expression M that takes a Church numeral c_n, followed by n arguments, and returns $\mathbf{I} = \lambda x.x$.

49. We would like to define a representation for *ordered pairs*. The algebra of ordered pairs involves three items:

    - The ordered pair of a, b, given by $\langle a, b \rangle$
    - The first projection, given by π_1^2
    - The second projection, given by π_2^2

    The relationship between these items is given by the following two equations:

$$\begin{array}{rcl} \pi_1^2 \langle a, b \rangle & = & a \\ \pi_2^2 \langle a, b \rangle & = & b \end{array} \tag{2}$$

    There are many adequate representations for ordered pairs and projections. These representations are not equal, as λ-terms, so we cannot use the pairing function from one representation with the projection function from another representation. However, for any λ-terms a, b, we can come up with infinitely-many different definitions for $\langle a, b \rangle, \pi_1^2, \pi_2^2$, that satisfy Eq (2).

50. The construction we introduce is based on the following ideas:

    - We represent an ordered pair of a, b as a function that takes a *selector* σ, and passes onto it a and b. It is up to the selector to "do the right thing" and return either a or b. The ordered pair $\langle a, b \rangle$ is thus defined as follows:

$$\langle a, b \rangle \quad \equiv \quad \lambda \sigma.\sigma a b$$

    - The first projection, π_1^2, takes an ordered pair p and passes onto it (i.e., applies it to) the selector $\lambda ab.a$, which returns the first of its two arguments. The first projection is thus defined as

$$\pi_1^2 \quad \equiv \quad \lambda p.p(\lambda ab.a)$$

    - The second projection, π_2^2, takes an ordered pair p and passes onto it (i.e., applies it to) the selector $\lambda ab.b$, which returns the second of its two arguments. The second projection is thus defined as

$$\pi_2^2 \quad \equiv \quad \lambda p.p(\lambda ab.b)$$

51. **Exercise 16:** Show that the representations we propose for $\langle a, b \rangle, \pi_1^2, \pi_2^2$ satisfy the relationships given in Eq (2).

52. **Exercise 17:** Find alternate definitions for $\langle a, b \rangle, \pi_1^2, \pi_2^2$

53. The construction of ordered-triples is similar to the construction of ordered pairs:

    - We represent an ordered triple of a, b, c as a function that takes a *selector* σ, and passes onto it a, b, c. It is up to the selector to "do the right thing" and return one of a, b, c. The ordered triple $\langle a, b, c \rangle$ is thus defined as follows

$$\langle a, b, c \rangle \quad \equiv \quad \lambda \sigma.\sigma a b c$$

- The first projection π_1^3, takes an ordered triple t and passes onto it (i.e., applies it to) the selector $\lambda abc.a$, which returns the first of its three arguments. The first projection is thus defined as

$$\pi_1^3 \quad \equiv \quad \lambda t.t(\lambda abc.a)$$

- The second projection π_2^3, takes an ordered triple t and passes onto it (i.e., applies it to) the selector $\lambda abc.b$, which returns the second of its three arguments. The second projection is thus defined as

$$\pi_2^3 \quad \equiv \quad \lambda t.t(\lambda abc.b)$$

- The third projection π_3^3, takes an ordered triple t and passes onto it (i.e., applies it to) the selector $\lambda abc.c$, which returns the third of its three arguments. The third projection is thus defined as

$$\pi_3^3 \quad \equiv \quad \lambda t.t(\lambda abc.c)$$

54. **Exercise 18:** Show that for any three λ-terms a, b, c, the definitions for $\langle a, b, c\rangle, \pi_1^3, \pi_2^3, \pi_3^3$, given in § 53, satisfy the relations:

$$\begin{aligned} \pi_1^3 \langle a, b, c\rangle &= a \\ \pi_2^3 \langle a, b, c\rangle &= b \\ \pi_3^3 \langle a, b, c\rangle &= c \end{aligned} \tag{3}$$

55. **Exercise 19:** Construct an alternate definition for ordered triples and their projections, using nested ordered pairs. Show that your definitions satisfy Eq (3) in § 54.

56. **Exercise 20:** Show that the ordered n-tuple and its projections can be defined in countably-many ways.

57. We would like to define the function f that maps an ordered pair to an ordered pair as follows:

$$\langle c_a, c_b\rangle \quad \stackrel{f}{\Longrightarrow} \quad \big\langle (S^+\ c_a), (\times\ c_a\ c_b)\big\rangle$$

Defining f is straightforward:

$$f \quad \equiv \quad \lambda p.\big\langle (S^+(\pi_1^2 p)), (\times(\pi_1^2 p)(\pi_2^2 p))\big\rangle$$

Starting with $\langle c_1, c_1\rangle$, repeated applications of f generate the following sequence of ordered pairs:

$$\begin{aligned} \langle c_1, c_1\rangle &\stackrel{f}{\Longrightarrow} \langle c_2, c_1\rangle \\ &\stackrel{f}{\Longrightarrow} \langle c_3, c_{1\cdot 2}\rangle \\ &\stackrel{f}{\Longrightarrow} \langle c_4, c_{1\cdot 2\cdot 3}\rangle \\ &\stackrel{f}{\Longrightarrow} \langle c_5, c_{1\cdot 2\cdot 3\cdot 4}\rangle \\ &\cdots \\ &\stackrel{f}{\Longrightarrow} \langle c_{n+1}, c_{n!}\rangle \end{aligned}$$

We can therefore define the factorial function using the n-th composition of f, as follows:

$$\textbf{Factorial} \quad \equiv \quad \lambda n.\pi_2^2(nf\,\langle c_1, c_1\rangle)$$

58. We define the function f that maps an ordered pair to an ordered pair as follows:

$$\langle c_a, c_b \rangle \quad \stackrel{f}{\Longrightarrow} \quad \langle c_b, (+\ c_a\ c_b) \rangle$$

Defining f is straightforward:

$$f \quad \equiv \quad \lambda p. \left\langle (\pi_2^2 p), (+(\pi_1^2 p)(\pi_2^2 p)) \right\rangle$$

Starting with $\langle c_0, c_1 \rangle$, repeated applications of f generate the following sequence of ordered pairs:

$$\begin{aligned}
\langle c_0, c_1 \rangle \quad &\stackrel{f}{\Longrightarrow} \quad \langle c_1, c_1 \rangle \\
&\stackrel{f}{\Longrightarrow} \quad \langle c_1, c_2 \rangle \\
&\stackrel{f}{\Longrightarrow} \quad \langle c_2, c_3 \rangle \\
&\stackrel{f}{\Longrightarrow} \quad \langle c_3, c_5 \rangle \\
&\cdots \\
&\stackrel{f}{\Longrightarrow} \quad \left\langle c_{\mathbf{Fibonacci}(n+1)}, c_{\mathbf{Fibonacci}(n)} \right\rangle
\end{aligned}$$

We can therefore define the Fibonacci function using the n-th composition of f, as follows:

$$\mathbf{Fibonacci} \quad \equiv \quad \lambda n. \pi_2^2 (n f \langle c_0, c_1 \rangle)$$

59. We define the function f that maps an ordered pair to an ordered pair as follows:

$$\langle c_a, c_b \rangle \quad \stackrel{f}{\Longrightarrow} \quad \left\langle c_b, S^+ c_b \right\rangle$$

Defining f is straightforward:

$$f \quad \equiv \quad \lambda p. \left\langle (\pi_2^2 p), (S^+ (\pi_2^2 p)) \right\rangle$$

Starting with $\langle c_0, c_0 \rangle$, repeated applications of f generate the following sequence of ordered pairs:

$$\begin{aligned}
\langle c_0, c_0 \rangle \quad &\stackrel{f}{\Longrightarrow} \quad \langle c_0, c_1 \rangle \\
&\stackrel{f}{\Longrightarrow} \quad \langle c_1, c_2 \rangle \\
&\stackrel{f}{\Longrightarrow} \quad \langle c_2, c_3 \rangle \\
&\stackrel{f}{\Longrightarrow} \quad \langle c_3, c_4 \rangle \\
&\cdots \\
&\stackrel{f}{\Longrightarrow} \quad \left\langle c_{n \dot{-} 1}, c_n \right\rangle
\end{aligned}$$

where the $\dot{-}$ function (pronounced "monus") is defined as follows:

$$a \dot{-} b \quad = \begin{cases} a - b & \text{if } a \geq b \\ 0 & \text{otherwise} \end{cases}$$

We can therefore define the predecessor function using the n-th composition of f, as follows:

$$P^- \quad \equiv \quad \lambda n. \pi_1^2 (n f \langle c_0, c_0 \rangle)$$

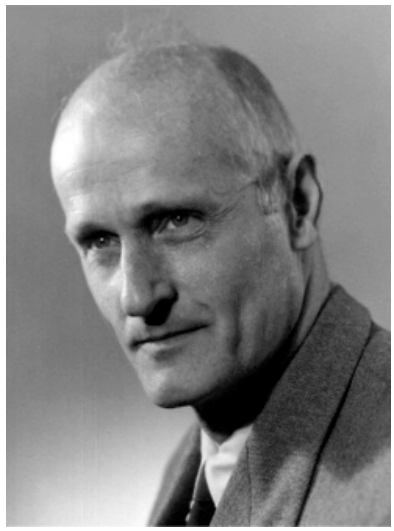


Figure 8: Stephen Cole Kleene

The technique of considering functions that map ordered n-tuples to ordered n-tuples in order to carry around additional state is due to the American logician *Stephen Cole Kleene* (January 5, 1909 — January 25, 1994), and was used in his construction of the predecessor function.[5]

60. Note that according to the above definition, $P^{-}c_0 = c_0$, because $0 \dot{-} 1 = 0$. While the number zero is not in the domain of the number-theoretic predecessor function, having the predecessor of zero be zero is a useful property to have for the recursion-theoretic definition of the predecessor function. The next item will use this property to define the λ-term that computes the monus function on Church numerals.

61. Clearly, $c_b P^{-} c_a = c_{a \dot{-} b}$. We therefore define the monus function on Church numerals by abstracting a, b over the left-hand side of the above equation:

$$\dot{-} \quad \equiv \quad \lambda ab.bP^{-}a$$

62. **Exercise 21:** Show that there are infinitely-many alternate definitions for the successor, predecessor, addition, subtraction, multiplication, and exponentiation on Church numerals.

63. The next items concern with defining and using the Boolean values *True* and *False*. The point of defining Boolean values is so that we can *dispatch* on two courses of action: <u>If</u> something is the case (i.e., evaluates to *True*), <u>then</u> return the value of one expression, <u>else</u> return the value of another. We could define the values c_0, c_1 to represent *True* and *False*, but this by itself would not provide us with the dispatching mechanism. The approach we take in this presentation is to build the dispatching mechanism right into the Boolean values themselves: We define the λ-terms **True**, **False** to model the [Platonic] Boolean values *True*, *False*, so that **True** returns the *first* of its two arguments,

[5] Kleene's construction for the predecessor function is slightly more complex than our own, because Kleene developed it in a formalism known as the $\lambda\mathbf{I}\beta\eta$-calculus, whereas our own construction is presented in the $\lambda\mathbf{K}\beta\eta$-calculus. We will have more to say on this when we reach the section on the $\lambda\mathbf{I}$-calculi.

and **False** returns the *second* of its two arguments:

$$\begin{aligned} \textbf{True} &\equiv \lambda xy.x \\ \textbf{False} &\equiv \lambda xy.y \end{aligned}$$

Let E_1, E_2 be two expressions. The λ-terms we use to model the Boolean values satisfy the following:

$$\begin{aligned} \textbf{True} E_1 E_2 &\twoheadrightarrow E_1 \\ \textbf{False} E_1 E_2 &\twoheadrightarrow E_2 \end{aligned}$$

64. We now define λ-terms that compute the elementary Boolean functions $\neg, \wedge, \vee$:

$$\begin{aligned} \textbf{Not} &\equiv \lambda b.(b\ \textbf{False}\ \textbf{True}) \\ \textbf{And} &\equiv \lambda ab.(a\ b\ \textbf{False}) \\ \textbf{Or} &\equiv \lambda ab.(a\ \textbf{True}\ b) \end{aligned}$$

We can use these functions to define any Boolean function.

65. **Exercise 22:** The Boolean functions *nand*, and *nor*, given by

$$\begin{aligned} a\ nand\ b &= \neg(a \wedge b) \\ a\ nor\ b &= \neg(a \vee b) \end{aligned}$$

are each functionally complete, i.e., can be used to define all other Boolean functions. Show how to define these directly in the λ-calculus, i.e., without using the definitions for $\neg, \wedge, \vee$.

66. **Exercise 23:** What are the difficulties in using in Scheme the encoding for Boolean values given in § 63? How might you overcome these difficulties?

67. **Exercise 24:** Define $\neg, \vee, \wedge$ using *nand*.

68. **Exercise 25:** Define $\neg, \vee, \wedge$ using *nor*.

69. **Exercise 26:** Consider the semantics of conjuction and disjunction in programming languages such as C, C++, Java, C#, etc. What would be the issues/problems/difficulties in implementing conjuction and disjunction using *nand* or *nor* ?

70. We would like to define the zero-predicate, which given a Church numeral c_n, returns **True** if $n = 0$, and **False**, otherwise. Since Church numerals are functions of two arguments, we're going to define the zero-predicate as $\textbf{Zero}? \equiv \lambda n.nAB$, that is, a λ-expression that takes a Church numeral and applies it to two expressions A, B. We now need to define A, B such that

$$\begin{aligned} (\textbf{Zero}?\ c_0) &\twoheadrightarrow c_0 AB \\ &\twoheadrightarrow (\lambda sz.z)AB \\ &\twoheadrightarrow B \\ &\equiv \textbf{True} \\ (\textbf{Zero}?\ c_{n+1}) &\twoheadrightarrow c_{n+1} AB \\ &\twoheadrightarrow A(\boxed{\underbrace{A \cdots (A}_{n} B) \cdots}) \\ &\twoheadrightarrow \textbf{False} \end{aligned}$$

We can solve these equations by defining $B = \mathbf{True}, A = \lambda x.\mathbf{False}$. Substituting our solutions for A, B into the definition for **Zero**?, we get:

$$\mathbf{Zero}? \quad \equiv \quad \lambda n.n(\lambda x.\mathbf{False})\mathbf{True}$$

71. The following definitions should be quite clear by now:

$$\begin{aligned}
\geq &\equiv \lambda ab.\mathbf{Zero}?(\dot{-}\ b\ a) \\
\leq &\equiv \lambda ab.\mathbf{Zero}?(\dot{-}\ a\ b) \\
> &\equiv \lambda ab.\mathbf{Not}(\leq\ a\ b) \\
< &\equiv \lambda ab.\mathbf{Not}(\geq\ a\ b) \\
= &\equiv \lambda ab.\mathbf{And}(\geq\ a\ b)(\leq\ a\ b) \\
\neq &\equiv \lambda ab.\mathbf{Not}(=\ a\ b)
\end{aligned}$$

72. **Exercise 27:** Define the λ-terms $\mathbf{Max}, \mathbf{Min}$ that take two Church numerals c_a, c_b, and return the larger or smaller of the two, respectively.

73. **Example:** The function $=3?$ can be defined using the equality predicate we have just defined:

$$=3? \quad \equiv \quad (=\ c_3)$$

but since 3 is known statically, we can use this information, and define $=3?$ directly:

$$\begin{aligned}
f \quad &\equiv \quad \lambda q.\langle(\pi_2^4 q), \\
&\qquad\qquad (\pi_3^4 q), \\
&\qquad\qquad (\pi_4^4 q), \\
&\qquad\qquad \mathbf{False}\rangle \\
=3? \quad &\equiv \quad \lambda n.\pi_1^4(n f\ \langle \mathbf{False}, \mathbf{False}, \mathbf{False}, \mathbf{True}\rangle)
\end{aligned}$$

The intuition behind the above definition is that the function f "shifts" the Boolean values through the ordered quadruple, from right-to-left (as in Hebrew ☺), so that after 3 shifts the first tuple now contains **True**.

74. We would like to define a function f that maps an ordered triple to another ordered triple as follows:

$$\langle c_a, c_b, c_c\rangle \quad \stackrel{f}{\Longrightarrow} \begin{cases} \langle c_a, c_b, c_c\rangle & a < b \\ \langle(\dot{-}\ c_a\ c_b), c_b, (S^+ c_c)\rangle & \text{otherwise} \end{cases}$$

Notice that for a sufficiently large n, we have

$$(f^n\ \langle c_a, c_b, c_0\rangle) \quad = \quad \left\langle c_{a\ \mathbf{mod}\ b}, c_b, c_{\lfloor a/b\rfloor}\right\rangle$$

We will thus use f to define both the quotient and the remainder functions on Church numerals. Defining f is straightforward:

$$\begin{aligned}
f \quad &\equiv \quad \lambda t.(< (\pi_1^3 t)(\pi_2^3 t) \\
&\qquad\qquad t \\
&\qquad\qquad \langle(\dot{-}\ (\pi_1^3 t)\ (\pi_2^3 t)), \\
&\qquad\qquad\ (\pi_2^3 t), \\
&\qquad\qquad\ (S^+(\pi_3^3 t))\rangle)
\end{aligned}$$

We now need to come up with a reasonable upper bound for n. This issue should also shed some light on a pecularity in the definition of f: We specifically defined f so as to leave the triple unchanged, once the first tuple is

less than the second. We did this to allow for some redundancy in the upper bound for n. In fact, c_a, the first element of the triple, is a good choice for n. We now define both the quotient and the remainder function:

$$\begin{aligned} \textbf{quotient} &\equiv \lambda ab.(\pi_3^3(af\,\langle a, b, c_0\rangle)) \\ \textbf{remainder} &\equiv \lambda ab.(\pi_1^3(af\,\langle a, b, c_0\rangle)) \end{aligned}$$

75. **Exercise 28:** Define a λ-term **Prime**?, which returns **True**, **False**, depending on whether or not the Church numeral to which it is applied represents a prime number.

76. **Exercise 29:** Define a representation for a linked list. Implement your representation, e.g., in Scheme. Write procedures to inter-convert between lists of natural numbers and your representation of a list of Church numerals.

77. **Exercise 30:** Define the λ-term **Bsort**, that performs *bubble sort* on your list representation for Church numerals, using Church numerals as the only iteration mechanism.

78. **Exercise 31:** Define the λ-term **Isort**, that performs *insertion sort* on your list representation for Church numerals, using Church numerals as the only iteration mechanism.

79. **Exercise 32:** Define the λ-term **Msort**, that performs *merge sort* on your list representation for Church numerals, using Church numerals as the only iteration mechanism.

80. **Exercise 33:** Define the λ-term **Qsort**, that performs *quicksort* on your list representation for Church numerals, using Church numerals as the only iteration mechanism.

81. **Exercise 34:** Define *integers* in the λ-calculus, as ordered pairs. For any $n \geq 0$:

$$\begin{aligned} n &\equiv \langle \textbf{True}, c_n\rangle \\ -n &\equiv \langle \textbf{False}, c_n\rangle \end{aligned}$$

    Define λ-terms that compute the basic number-theoretic functions and predicates on this representation.

82. **Exercise 35:** Define *rational numbers* in the λ-calculus, as ordered pairs of two integers (defined as above). Define λ-terms that compute the basic number-theoretic functions and predicates on this representation. Make sure they return a *reduced* fraction.

83. Up to now, we have been using Church numerals exclusively as our numeral system. Church numerals indeed offer many advantages: They are the first numeral system developed for the λ-calculus, they are well-studied and well-understood, and they provide a "builtin" bounded iteration mechanism. Still, the λ-calculus has no "builtin" representation for the natural numbers, and we are free to use other representations. The remainder of this section will be exploring alternatives to Church numerals.

84. Suppose we construct another representation for numbers, $\{d_n\}_{n\in\mathbb{N}}$, which we shall call "D-numerals". The λ-terms that compute functions over Church numerals will not work with D-numerals directly, i.e., applying these λ-terms to D-numerals will not perform the computations that we expect. We could

try to define λ-terms that compute standard functions and predicates on D-numerals, but that could be difficult, and we will need to re-do much of the work we did on Church numerals.

Another approach might be to re-use the work we had invested in λ-definability over Church numerals. We will need to define λ-terms that convert Church numerals to D-numerals, and vice versa. Suppose we define λ-terms $C_D^{\text{Church}}, C_{\text{Church}}^D$, such that

$$\begin{aligned} C_D^{\text{Church}}\ d_n &= c_n \\ C_{\text{Church}}^D\ c_n &= d_n \end{aligned}$$

Then for any λ-term E_{Church} that computes some unary function on Church numerals, we can define the corresponding λ-term E_D that computes the same function on D-numerals as follows

$$\begin{aligned} E_D &= C_{\text{Church}}^D \circ E_{\text{Church}} \circ C_D^{\text{Church}} \\ &= \lambda d.C_{\text{Church}}^D \left(E_{\text{Church}} \left(C_D^{\text{Church}} d\right)\right) \end{aligned} \tag{4}$$

If F_{Church} computes a binary function on Church numerals, then the corresponding F_D can be defined as follows:

$$F_D = \lambda d_1 d_2.C_{\text{Church}}^D \left(F_{\text{Church}} \left(C_D^{\text{Church}} d_1\right) \left(C_D^{\text{Church}} d_2\right)\right) \tag{5}$$

Functions of other arity can be defined similarly.

85. **Exercise 36:** Let $d_n = \lambda x.\underbrace{xx\cdots x}_{n+1}$ define the D-numerals. Define corresponding λ-terms for $C_D^{\text{Church}}, C_{\text{Church}}^D$, respectively.

86. **Exercise 37:** Let $e_n = \lambda xy.x\underbrace{yy\cdots y}_{n}$ define the E-numerals. Define the corresponding λ-terms for $C_E^{\text{Church}}, C_{\text{Church}}^E$, respectively.

87. **Exercise 38:** We define the F-numerals as follows:

$$\begin{aligned} f_0 &= \langle \mathbf{True}, \mathbf{True} \rangle \\ f_{n+1} &= \langle \mathbf{False}, f_n \rangle \end{aligned}$$

Define the corresponding λ-terms for $C_F^{\text{Church}}, C_{\text{Church}}^F$, respectively.

88. **Exercise 39:** Invent your own numeral system. Try to avoid trivial variations on existing numeral systems. Can you build some iteration mechanism into it, so that you could easily define functions over it, without converting to and from Church numerals?

89. **Exercise 40:** Show there exist infinitely-many distinct numeral systems in the λ-calculus.

3 Fixed-Points I (§90–137)

90. This topic is really about *names*. Up to now, we've been defining all kinds of λ-term: $S^+, c_n, \dot{-}$, etc. We've been using these terms as building blocks in constructing other, more complex terms. We've been referring to these building blocks by their *names*.

91. In theory, we could have substituted these names for their respective definitions. In this sense, the names aren't really necessary; They are just convenient *place holders* for the expressions for which they stand. As place holders, they exist in the *meta-language* of the λ-calculus as abbreviations or macros.

92. In practice, however, whenever we wrote computer code that corresponded to these definitions, the names of the functions turned into top-level procedure definitions in Scheme or in ML. The names were *free variables*.

93. The λ-calculus, however, doesn't have a notion of free variables. How limiting is their absense? Are there functions that cannot be defined in the λ-calculus because it lacks free variables?

94. The immediate suspect is *recursion*. In recursive definitions the name of the expression is not an abbreviation. Suppose, for example, we were to define the factorial function *recursively*. It might look something like:

$$\textbf{fact} \quad \equiv \quad \lambda n.(\textbf{Zero}?\ n\ c_1\ (\times\ n\ (\textbf{fact}\ (P^{-}n))))$$

If we were to substitute the name **fact** in the expression on the right-hand side for its definition, we would get an expression that would still contain a mention of the name **fact**:

$$\begin{aligned}\textbf{fact} \quad \equiv \quad \lambda n.(&\textbf{Zero}?\ n\ c_1\\ &(\times\ n\\ &(\boxed{(\lambda n.(\textbf{Zero}?\ n\ c_1\ (\times\ n\ (\textbf{fact}\ (P^{-}n)))))})\ (P^{-}n))))\end{aligned}$$

Notice that the resulting expression also mentions **fact**. Clearly, this substitution can never end. Recursive definitions are thus different from non-recursive definitions in the way they make use of names. The remainder of the presentation on fixed points is concerned with how to define recursive functions without relying on global names in ways that break the substution model.

95. Fixed points are names given to specific points in the domains of functions. For a function f, a value x_0 in the domain of f is a fixed point of f if and only if $f(x_0) = x_0$.

96. To make a [very] long story short, λ-terms that compute recursive functions are going to turn out to be fixed points of other λ-terms. The rest of the presentation will concern itself with ways of finding the fixed points of λ-terms.

97. Before we get into a more formal definition of what fixed points mean in the λ-calculus, we continue the tradition of doing things the *wrong way*, and building on your Scheme intuition. ☺ We start by considering a simple recursive procedure. Factorial is as good as any, and a predictable guess by now:

```
(define fact
  (lambda (n)
    (if (zero? n) 1
      (* n (fact (- n 1))))))
```

In light of the previous discussion, we note the circular use of the name `fact`. We now begin a series of transformations that should take us from the recursive definition of factorial to a non-recursive definition. The first thing we'll do in order to eliminate the circular use of the name is to abstract over it. Here is the definition of the term F:

```
(lambda (fact)
  (lambda (n)
    (if (zero? n) 1
      (* n (fact (- n 1))))))
```

The first thing to notice about F is that it's not a recursive function. We really don't know what `fact` stands for — It's just a parameter to the outermost `lambda`. What does F do? This really depends on what we apply F to, namely on the value of `fact`. Never mind what F does or could do. Just think about it as an expression that's syntactically related to the original recursive definition of factorial, and keep in mind that you can create corresponding expressions for any recursive procedure just by abstracting over the name of the recursive procedure.

98. Now forget about F for a moment, and consider the expression G:

```
(lambda (fact n)
  (if (zero? n) 1
    (* n (fact fact (- n 1)))))
```

What does G compute? Who knows, and who cares... Again, this would depend on what G is applied to. But we do make the following claim:

$$(G\ \ G\ \ n) \quad = \quad n!$$

It's easy to verify this claim by induction. First verify that $(G\ \ G\ \ 0)$ evaluates to `1`. Now assume that for some $n > 0$, $(G\ \ G\ \ n)$ evaluates to $n!$. We have $(G\ \ G\ \ \boxed{n+1}) = (*\ \boxed{n+1}\ \boxed{(G\ \ G\ \ n)})$, but since $(G\ \ G\ \ n) = n!$, by our induction hypothesis, it follows that $(G\ \ G\ \ \boxed{n+1)} = (*\ \boxed{n+1}\ \ n!) = \boxed{n+1}!$, which completes our proof. You should now realize that we've just found a way of computing a recursive function (factorial) without recursion. By using a construction such as G, we replace recursion with *self application*. Self application means that a procedure is applied to itself. Self application is a problematic concept, because if a function can take itself as an argument, then it follows that the function is a member of *its own* domain. Using mathematical notation, this is like writing $f(f)$. For a λ-expression to be applicable to itself and still compute a function in the mathematical and logical sense, it is necessary to define the domain of this function very carefully. We will begin to explore this, and related issues, later on in the course when we discuss *type theory*. In the meantime, we will use our intuition from object-oriented programming languages, such as Java, to make sense of self application. Suppose you have an instance `x` of a class `Foo`. You can imagine that `Foo` contains an instance method with the following signature:

```
Foo f(Foo y) { ... }
```

Therefore you can call `x.f(...)` with an argument of type `Foo`. So, in particular, you can have `x.f(x)` and get an instance of `Foo` as a return value. In fact, this kind of method calls are not as uncommon as you might imagine!

99. It is also possible to encode G in the C programming language:

```
int G(void *g, int n) {
  if (n == 0) return 1;
  else return n * ((int(*)(void*,int))g)(g, n-1);
}
```

The function G would be invoked, e.g., to compute 5!, as follows:

```
int n120 = G(&G, 5);
```

The value of `n120` is the integer 120.

Why have we used `void *` for the type of the first argument to `G`? Clearly, we mean to pass a function — `G` itself as this first argument. Why are we not using the correct type of `G`?

Before substituting the type of `G` for `void *`, we have the type of `G` as:

```
int G(void *g, int n);
```

After substituting the type of `G` once for `void *`, we have the type of `G` as:

```
int G(int (*g)(void *, int), int n);
```

After substituting the type of `G` twice for `void *`, we have the type of `G` as:

```
int G(int (*g)(int (*)(void *, int), int), int n);
```

After substituting the type of `G` thrice for `void *`, we have the type of `G` as:

```
int G(int (*g)(int (*)(int (*)(void *, int), int), int),
     int n);
```

It is easy to see that no matter how many times we substitute the type of `G` for `void *` the type of `G` will never be free of `void *`. This should remind you of § 94, in which we attemped to replace the name of a recursive procedure by its definition in the very body of the recursive procedure. We only managed to obtain larger and larger expressions, each of which was recursive. The "limit" of this substitution process was an infinite program. Here we have a similar problem: A procedure that uses self-application and has a *recursive type*. Attempting to substitute the type for its occurence results in another recursive type. The "limit" of this substitution process is a program with an infinite type. Recursive types can too be defined via fixed point operators, but this goes beyond the scope of our discussion.

What we should take from this discussion, however, is a better appreciation for the use of `void *` in the original definition for `G`. Its purpose is to spare us, as well as the compiler, from getting into such questions as what must the type of `G` be, and how to reason about, and compile recursive types. In fact, C compiles the original definition for `G` without any warning. The code is even portable. ☺

100. Back to G, we are now going to consider the term H, which is syntactically and semantically related to G:

```
(lambda (fact)
  (lambda (n)
    (if (zero? n) 1
      (* n ((fact fact) (- n 1))))))
```

You should think of H as a *Curried* version of G: Rather than having `(lambda (fact n) ... )`, we have two nested procedures. From the way we used G, you can imagine that we plan on applying H to itself, from which it follows that since H is a Curried G, we need to replace the application `(fact fact (- n 1))` in G with a left-associated version `((fact`

`fact) (- n 1))`. Similar to our claim with respect to G, we claim that we can compute the factorial function with H, as well:

$$((H\ \ H)\ \ n) \quad = \quad n!$$

Verifying this claim by induction is easy, and similar to what we did with G. The question now becomes: What should we call $(H\ \ H)$? Let's see what we know about it: It's a procedure, and when we apply it to n we get $n!$. The $(H\ \ H)$ seems to be just what we were looking for – a non-recursive version of the factorial function.

101. The procedure H bears, however, a strong relation to F:

The Text of F

```
(lambda (fact)
  (lambda (n)
    (if (zero? n) 1
      (* n (fact (- n
          1))))))
```

The Text of H

```
(lambda (fact)
  (lambda (n)
    (if (zero? n) 1
      (* n ((fact fact) (- n
          1))))))
```

The only difference between F and H is that the occurence of `fact` within F has been replaced with the self-application `(fact fact)`. What we are looking for is a way to "convert" F into H, and then applying H to itself, thereby obtaining a construction for our recursive function. But all our plans are dependent on this single question: How to convert F to H.

102. In fact, converting F to H is a *rename* operation, and we accomplish it through *function composition*. To clarify things, let's consider an analogous problem from the world of mathematics. Suppose we are given the following functions f, g:

$$\begin{aligned} f(x) &= x+1 \\ g(x) &= x^2 \end{aligned}$$

Suppose we would like to use f, g to define the following two functions:

$$\begin{aligned} q(x) &= x^2+1 \\ r(x) &= (x+1)^2 \end{aligned}$$

We can define q by replacing each occurence of x in $f(x)$ by x^2. Similarly, we can define r by replacing each occurence of x in $g(x)$ by $x+1$. We can achieve this renaming operation by composing f and g accordingly:

$$\begin{aligned} q(x) &= f(g(x)) &= (f \circ g)(x) \\ r(x) &= g(f(x)) &= (g \circ f)(x) \end{aligned}$$

Similarly, we can define H by composing F with the appropriate function. Specifically, we need a function that takes a single argument and applies it to itself. Composing F with such a function, will return an expression similar to F, with the singular difference being that each occurence of the parameter in the body of F has now been renamed to a self-application. In other words, the function with which we compose F is $(\lambda x.xx)$:

$$H \quad = \quad (F \circ (\lambda x.xx)) \quad = \quad \lambda x.F(xx)$$

Recall that we define our recursive function by self-applying H:

$$\begin{aligned}(HH) &= ((\lambda x.F(xx)) \\ &\quad\ (\lambda x.F(xx)))\end{aligned}$$

While we were discussing specific expressions F, H that are related to a specific recursive function (factorial), keep in mind that for any recursive function, we can define a corresponding F, by abstracting over the name of the function, generate a corresponding H, by composing F with $\lambda x.xx$, and then self-apply H to define the original function using self-application, rather than recursion. The natural step, there therefore, to abstract over F in the right-hand side of the expression for (HH), giving:

$$\begin{aligned}\mathrm{Y}_{\textsc{Curry}} &= \lambda f.((\lambda x.f(xx)) \\ &\quad\ (\lambda x.f(xx)))\end{aligned}$$

What kind of an expression is $\mathrm{Y}_{\textsc{Curry}}$?

- $\mathrm{Y}_{\textsc{Curry}}$ takes some F that abstracts the name of some recursive function, over the recursive definition.
- It returns an expression that defines the corresponding recursive function, using self-application instead of recursion.

You can think of $\mathrm{Y}_{\textsc{Curry}}$ as a general-purpose "recursion-maker".

103. At this point it is also easier to appreciate the motivation for introducing G as an intermediate expression on the way to H. G is simpler to understand, reason about, and implement, because unlike H, it does not require the use of *closures*. The only variables used by G are *parameters*, which is why it can be implemented in C, which has no closures but only function points.

104. In case you haven't notice, there is a minor sleight of hand in the above explanation. We conveniently shifted from Scheme to the λ-calculus, without giving any reason or motivation for this. The resulting $\mathrm{Y}_{\textsc{Curry}}$ is also given in the λ-calculus, and indeed it "works" in the λ-calculus. The situation in Scheme is a bit more complicated. If you tried to define H as we suggested, you might decide to compose F with `(lambda (x) (x x))`, giving `(lambda (x) (`F` (x x)))`. Applying H to itself, rather than giving you a recursive function, will result in an infinite loop, until the stack space is exhausted, at which point an error will occur. To understand the issue, let's examine more closely the original term H, and the one we propose to construct from F:

The Original H

```
(lambda (fact)
  (lambda (n)
    (if (zero? n) 1
      (* n ((fact fact) (- n
          1))))))
```

H Defined Using F

```
(lambda (x) (F (x x)))
```

To appreciate the difference between these two expressions, ask yourself the following question: *When* does the self-application take place? In other words, when do we compute `(fact fact)`, or `(x x)`?

Under Scheme's *applicative order* of evaluation, the expression `(fact fact)` on the left-hand side will evaluate *after* the procedure `(lambda (n) ...`

`)` gets applied, that is, after the value of `n` is already known, and after the `if`-expression is given an opportunity to stop the computation (in case `(zero? n)` happens to be true). This is why `(H H)` doesn't go into an infinite loop in the first place: The self-application is guarded by an `if`-expression.

Now consider the expression on the right-hand side. Under Scheme's applicative order, the self-application `(x x)` is evaluated as soon as the expression `(lambda (x) ... )` is applied, i.e., before the body of F is entered, and before the `if`-expression has the possibility of terminating the computation. The result is an infinite loop, caused by eagerly attempting to evaluate the following chain:

$$
\begin{array}{rcl}
\texttt{((lambda (x) (}F\texttt{ (x x)))} & = & \\
\texttt{(lambda (x) (}F\texttt{ (x x))))} & & \\
\texttt{(}F\texttt{ ((lambda (x) (}F\texttt{ (x x)))} & = & \\
\texttt{(lambda (x) (}F\texttt{ (x x)))))} & & \\
\texttt{(}F\texttt{ (}F\texttt{ ((lambda (x) (}F\texttt{ (x x)))} & = & \underbrace{\texttt{(}F\texttt{ (}F\texttt{ (}F\texttt{ }\cdots\texttt{)))}}_{\to\infty} \\
\texttt{(lambda (x) (}F\texttt{ (x x))))))} & &
\end{array}
$$

We can prevent the infinite loop if we can delay the evaluation of `(x x)` until the `else`-clause of the `if`-expression gets evaluated. The classical mechanism for delaying evaluation in Scheme and LISP is known as η-expansion (pronounced "eta-expansion").

105. Imagine you are a teaching assistant for a numerical analysis course, and you ask your students to implement the *cosine* function. Most of your students did what you expected them: They impelmented a summation of the Taylor series for $\cos x$, made sure that the number of terms and the accuracy was such that the result was within the errors you specified, etc. One student, however, turned in the following C code:

```
#include <math.h>
...
double cosine(double x) {
  return cos(x);
}
```

You may think this student should fail, but how are you going to justify failing him? After all, the code does work, it's very efficient, it's very well debugged, etc. How would you articulate your intuition that the student didn't solve the problem?

One thing you may note is that the function `cosine` is just an *interface* for the `cos` function that is already built into the standard math library. One way to say this is that `cosine` does nothing except take an argument and pass it onto `cos`. It's the latter that does all the work. Having formulated this intuition, you might want to specify it as a transformation, starting with a program like `cosine` and ending up with the corresponding "simplified" `cos`. In the λ-calculus, we call this transformation by the name of η-reduction. The inverse of η-reduction is called η-expansion, and is often denoted by η^{-1}:

$$
\boxed{\texttt{(lambda (x) (cos x))}} \quad \underset{\eta^{-1}}{\overset{\eta}{\rightleftarrows}} \quad \boxed{\texttt{cos}}
$$

The thing about η-expansion is that according Scheme's applicative-order semantics, the body of a procedure is not evaluated before the procedure is applied. This means that while `(/ 5 0)` would generate a run-time error, `(lambda () (/ 5 0))` is just a constant, and can be returned or passed as value without problems. Upon evaluation, any application of this constant will generate a run-time error, of course.

106. Eta-reduction and η-expansion highlight another difference between the λ-calculus and Scheme, or any "real-world" programming language for that matter: In the λ-calculus, any term M is η-equivalent to $(\lambda x.Mx)$, for any variable $x \notin \mathsf{FreeVars}(M)$. But in Scheme, the number `4` is surely not equivalent, in any realistic sense, to `(lambda (x) (4 x))`. This equivalence holds only for expressions that can be applied, and `4` cannot be applied. In the λ-calculus, however, any expression can be applied, so η-equivalence is unrestricted.

107. Getting back to delaying the evlauation of `(x x)` in the Scheme version of $\mathrm{Y}_{\text{Curry}}$, we're going to use η-expansion to as the delay mechanism. Rather than using `(x x)`, we're going to wrap `(x x)` with a `lambda`, that takes as many parameters as `(x x)` is expected to take. The following table should clarify the situation:

We use	**when `(x x)` takes**
`(lambda (arg) ((x x) arg))`	1 argument
`(lambda (arg1 arg2) ((x x) arg1 arg2))`	2 argument
...	...
`(lambda args (apply (x x) args))`	any number of arguments

We will have more to say about η-expansion later, when we go back and re-do our introduction to the λ-calculus more formally.

108. We now encode $\mathrm{Y}_{\text{Curry}}$ in Scheme, for procedures that take any number of arguments, by delaying the self-application in the Scheme encoding of H, and then applying this encoding to itself:

```
(define Ycurry
  (lambda (f)
    ((lambda (x)
       (f (lambda args
            (apply (x x) args))))
     (lambda (x)
       (f (lambda args
            (apply (x x) args)))))))
```

109. We can use `Ycurry` to define recursive procedures without recursion, by replacing recursion with self-application:

```
(define fact
  (Ycurry
    (lambda (!)
      (lambda (n)
        (if (zero? n) 1
          (* n (! (- n 1))))))))
```

Since `Ycurry` uses a variadic procedure for implementing the η-expansion, we can use the same procedure to define recursive procedures of more than a single argument:

```
(define ackermann
  (Ycurry
    (lambda (ack)
      (lambda (a b)
        (cond ((zero? a) (+ b 1))
              ((zero? b) (ack (- a 1) 1))
              (else (ack (- a 1)
                         (ack a (- b 1)))))))))
```

110. Let us go back to the procedure `F` defined earlier, in the context of defining the factorial procedure:

```
(define F
  (lambda (fact)
    (lambda (n)
      (if (zero? n) 1
        (* n (fact (- n 1)))))))
```

Just what kind of a procedure is it? What can we say about it? After all, it *is not* the same as the factorial procedure, although it is clearly related to it in some sense.

It is clear that `fact`, the argument to `F`, is a procedure, because we can see an application of it within the body of `F`: `(fact (- n 1))`. What do we get when we apply `F` to the factorial function?[6]

We notice that `(F fact)` is some function of an integer `n`. Applying this function to `0`, we see that `((F fact) 0)` $\Longrightarrow$ `1`. Recall that $0! = 1$. For $n > 0$, we skip the *then*-clause of the `if`-expression, and evaluate the *else*-clause: Evaluating `(* n (fact (- n 1)))`. But since `fact` computes the factorial function, it follows that `(fact (- n 1))` $\Longrightarrow$ `n-1`!, so `(F fact)` computes the factorial function for all $n \geq 0$. Since `(F fact)` $\equiv$ `fact`, it follows that `fact` is a fixed point of `F`.

111. A function can have more than a single fixed point. For example, the function $f(x) = x^2$ has two: $0, 1$. Any value in the domain of the function $g(x) = x$ is a fixed point of g. While it is interesting that `fact` is a fixed point of `F`, we are really after a slightly stronger property: We want `fact` to bear a unique relation to `F`.

This is how we come by the notion of a *least fixed point* of some function. The notion should be familiar to you from domains on which there is a total ordering. For example, 0 is the least of the two fixed points $0, 1$ of $f(x) = x^2$. What does it mean to be a least fixed point on a domain that consists of functions, and what kind of ordering are we considering? These questions are a part of an area known as *lattice theory*. We will not pursue these questions in this course, but we can give some intuition of what is meant by them. The point of this discussion is to establish an intuitive understanding of why the factorial function is the *least fixed point* of `F`.

112. There are essentially two ways of thinking about functions. A function can be thought of as

[6] When we say "applying `F` to the factorial function" we mean *any implementation* of the factorial function, and we do not care whether this implementation was obtained through iteration, recursion, self-application, or the use of a fixed-point combinator such as $\mathrm{Y}_{\mathrm{Curry}}$.

- A function f is a cross product $domain \times range$, i.e., a set of ordered pairs the first tuple of which is an element of the domain of the function, and the second tuple of which is the corresponding element of in the range, with the one restriction being that if $\langle x, y\rangle, \langle x, z\rangle \in f$, then $y = z$. This is, perhaps, how a set-theorist might think of a function.
- A "black box" that "takes" an input and "generates" an output. This is, perhaps, how a computer scientist might think of a function.

Up to now, we were discussing λ-terms that compute functions, meaning, we were closer to the computer scientist's way of thinking about functions. Now let us shift tacitly into the logician's way of thinking about functions, i.e., as certain sets of ordered pairs.

113. The ordering we are going to consider is based on the set relation ($\subseteq$). In lattice theory, we speak abstractly about the ordering, and use a slightly different notation ($\sqsubseteq$), and while this need not hold in general, in this specific case, the meaning of $\sqsubseteq$ is just the set-theoretic $\subseteq$. Put otherwise, we can say that the symbol $\subseteq$ comes from the set-theoretic characterisation of functions, and in lattice theory, we consider lattices in general (rather than specifically lattices of functions), and certainly apart from how functions are represented in set theory. But for the rest of this discussion, we will stick with the set-theoretic view of functions, and use the symbol $\subseteq$.

Shifting from Scheme back into the λ-calculus, what we claim, in fact, is that for F defined as follows:

$$\begin{array}{rcl} F & = & \lambda fact. \\ & & \quad \lambda n.\mathbf{Zero}?\ n\ c_1 \\ & & \qquad (\times\ c_n\ (fact\ (P^-\ c_n))) \end{array}$$

the factorial function is the least fixed point of F. Moving to the set-theoretic characterisation of functions, this means that if *fact* is the factorial function, and g is any fixed point of F, then $fact \subseteq g$. Moving for a moment into lattice-theoretic language, if we can show that $fact \sqsubseteq g$, for any g that is a fixed point of F, it would follow that *fact* is the least fixed point of F.

But this is still just a claim. How would we go about proving this claim? Perhaps surprisingly, a simple proof by induction will suffice:

Claim: Let F be as above, let *fact* be the factorial function, and let g be any fixed point of F. We claim that $fact \sqsubseteq g$.

Proof: To show $fact \sqsubseteq g$ we really need to show $fact \subseteq g$. The structure of our proof is by induction, and we begin by showing the base case, namely that $\langle 0, 1\rangle \in g$. We will then show that $\langle n+1, (n+1)!\rangle \in g$ follows from the induction hypothesis that $\langle n, n!\rangle \in g$. This will complete our proof.

Since g is a fixed point of F, then $g = (Fg)$.

- Base Case: It is clear, by examing F, that

$$\begin{array}{rcl} (gc_0) & = & ((Fg)c_0) \\ & = & c_1 \end{array}$$

so $\langle 0, 1\rangle \in g$, and the base case holds.

- INDUCTION STEP: We assume that for some $n \geq 0$, $\langle n, n! \rangle \in g$.

$$\begin{array}{rcll} (g\ c_{n+1}) & = & ((F\ g)c_{n+1}) & \\ & = & (\times\ c_{n+1}\ (g\ (P^{-}\ c_{n+1}))) & \\ & = & (\times\ c_{n+1} \boxed{(g\ c_n)}) & \text{where } \boxed{(g\ c_n)} \text{ is equal to } n!\text{, by our induction hypothesis that } \langle n, n! \rangle \in g \\ & = & (\times\ c_{n+1}\ c_{n!}) & \\ & = & c_{(n+1)!} & \end{array}$$

So $\langle n, n! \rangle \in g$, for all $n \in \mathbb{N}$, so $fact = \{\langle n, n! \rangle : n \in \mathbb{N}\} \subseteq g$, so $fact \sqsubseteq g$. ■

113(i) This is the sense in which *fact* is the *least* fixed point of F: Any computation expressed by *fact* is embedded witnin any other fixed point of F, meaning that any other fixed point of F computes *fact* as well, in addition to any other computation it may also perform.

114. It should now be clear why expressions such as $\mathrm{Y}_{\text{CURRY}}$ are called *least fixed-point combinators*: Given any expression M, $(\mathrm{Y}_{\text{CURRY}}\ M)$ is equal to the least fixed point of M.

115. Returning to F, there is an additional property of F that is worth mentioning: The relation between F and the *partial* factorial functions.

116. If you think of a function as a set of ordered pairs, then you can think of a partial function as a subset of that set. A partial function computes values for a subset of the domain of the *total* function. Let us now consider a particular sequence of partial factorial functions.

Partial Function	**Domain**	**The Function as a Set**
$!_\emptyset$	$\emptyset$	$\emptyset$
$!_0$	$\{0\}$	$\{\langle 0, 1 \rangle\}$
$!_1$	$\{0, 1\}$	$\{\langle 0, 1 \rangle, \langle 1, 1 \rangle\}$
$!_2$	$\{0, 1, 2\}$	$\{\langle 0, 1 \rangle, \langle 1, 1 \rangle, \langle 2, 2 \rangle\}$
$\vdots$	$\vdots$	$\vdots$
$!_n$	$\{0, \ldots, n\}$	$\{\langle k, k! \rangle : 0 \leq k \leq n\}$
$\vdots$	$\vdots$	$\vdots$

It is easy to show by induction that $!_{n+1} = (F\ !_n)$. So another way of thinking about the factorial function as the least fixed point of F is as a kind of limit: $fact = \lim_{n\to\infty} !_n = \lim_{n\to\infty}(F^n\ !_\emptyset)$, where F^n is the n-th composition of F.

117. If the previous item, with its talk about partial functions seemed a bit woolly to you, or if you are not clear about the relation between the partial functions and the F functional, then following session at the Scheme prompt should clarify things. Let us define F in Scheme:

```
(define F
  (lambda (fact)
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (- n 1)))))))
```

We now define `fact-nothing`, which corresponds to $!_\emptyset$ in § 116:

```
(define fact-nothing
  (lambda (n)
    (error 'fact-nothing
           "Can't␣compute␣(fact␣~a)"
           n)))
```

We now define the respective partial functions by repeatedly applying F to the previous partial function:

```
(define fact-0 (F fact-nothing))
(define fact-1 (F fact-0))
(define fact-2 (F fact-1))
(define fact-3 (F fact-2))
(define fact-4 (F fact-3))
(define fact-5 (F fact-4))
```

And now we can use these partial functions. Please pay careful attention to when we generate an error message, i.e., when we call `fact-nothing`:

```
(fact-nothing 0)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
> (fact-0 0)
1
> (fact-0 1)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
> (fact-1 0)
1
> (fact-1 1)
1
> (fact-1 2)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
> (fact-2 0)
1
> (fact-2 1)
1
> (fact-2 2)
2
> (fact-2 3)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
> (fact-3 0)
1
> (fact-3 1)
1
> (fact-3 2)
2
> (fact-3 3)
```

```
6
> (fact-3 4)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
> (fact-4 0)
1
> (fact-4 1)
1
> (fact-4 2)
2
> (fact-4 3)
6
> (fact-4 4)
24
> (fact-4 5)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
> (fact-5 0)
1
> (fact-5 1)
1
> (fact-5 2)
2
> (fact-5 3)
6
> (fact-5 4)
24
> (fact-5 5)
120
> (fact-5 6)

Error in fact-nothing: Can't compute (fact 0).
Type (debug) to enter the debugger.
```

Finally, in case you need one more jogger, how about this:

```
> ((F (F (F (F cos)))) 4)
24
> ((F (F (F (F (F cos))))) 5)
120
```

but on the other hand:

```
> ((F (F (F (F (F cos))))) 6)
389.0176602250606
```

And yes, that was the *cosine* function. In radians. ☺

The point of this exercise is to help you see that

- The factorial function is a fixed point of the F functional.
- The fixed point, i.e., the factorial function, is obtainable by the usual fixed-point iteration, i.e., is the limit of the process $F(F(F(\cdots)))$ in the same sense as the limits you were taught in your calculus course, and the fixed-point iteration you were taught in your numerical analysis course.

- That the fixed point is the same as the infinite composition of the F functional:

$$! \quad = \quad F^{\infty}(\boxed{\textit{anything}})$$

and therefore

$$n! \quad = \quad \left(F^{\infty}(\boxed{\textit{anything}})\right)(n)$$

where $\boxed{\textit{anything}}$ means literally anything![7]

When $F^{\infty}(\textit{anything})$ is applied to any finite number n, only n compositions of F are needed to compute $n!$. Hence, when using $F^{\infty}(\boxed{\textit{anything}})$ we should never have to care about what this $\boxed{\textit{anything}}$ might stand for — it could even be the *cosine* function — because we should never have to apply it!

118. Since only a finite number n of compositions of the F functional are needed to compute a value for any given input, it is possible, in case an upper bound can be determined, to use a Church numeral, rather than a fixed-point combinator, to iterate over the functional F. It is not necessary to determine the exact number of iterations up front; An upper bound will do nicely. For example, let's compute the factorial function using the appropriate higher-order functional F, and a Church numeral rather than a fixed-point combinator:

ADD SOMETHING HERE !!!

119. Let us consider a different angle to the question of the leastness of the fixed point. We know that any λ-term is a fixed point of the identity combinator $\mathbf{I} = \lambda x.x$. So what is the least fixed point of $\mathbf{I}$?

$$\begin{aligned}(\mathrm{Y}_{\text{Curry}}\ (\lambda x.x)) &= ((\lambda x.((\lambda x.x)(xx))) \\ &\quad\ (\lambda x.((\lambda x.x)(xx)))) \\ &= ((\lambda x.xx) \\ &\quad\ (\lambda x.xx))\end{aligned}$$

We recognize this last term – It's the simplest way of encoding an "infinite loop" in the lambda calculus (or in "pure Scheme"). But what kind of a function is the infinite loop? It is really the [partial] function defined over the empty set domain, meaning the function that takes nothing and returns nothing. This function, the set-theoretic representation of which is the empty set, is clearly a subset of any other function, hence it is clear that the infinite loop is the least fixed point of $\mathbf{I}$.

120. **Exercise 41:** The infinite loop, i.e., the λ-term denoting the partial function over the empty set, is also a subset of the factorial function, as well as any of the partial factorial functions. Why, then, isn't the infinite loop the least fixed point of F?

121. **Exercise 42:** Can you encode $\mathrm{Y}_{\text{Curry}}$ in Java? What about C?

[7] In fact, it would have been more appropriate to use the first infinite ordinal ω, and to talk about F^{ω}, but we are not going to assume familiarity with ordinal arithmetic.

122. **Exercise 43:** Define a λ-expression that takes an argument f and returns f^∞ — the infinite composition of f. A good name for such a λ-expression would be c_∞ — the "infinite" Church numeral.

123. **Exercise 44:** In § 107, we defined the η-expansion appropriate for functions of 1 variable, 2 variables, and *variadic* functions that can take any number of variables. We used these forms of η-expansion to write applicative-order fixed-point combinators (e.g., in § 108). Define an applicative-order version of $\mathrm{Y}_{\text{Curry}}$, in Scheme, that takes *nilary* procedures, i.e., *procedures of zero arguments.*

124. **Exercise 45:** Find a λ-expression the fixed points of which are precisely the set of Church numerals. No more and no less. This means that you should find an expression P such that for all $n \in \mathbb{N}$, we have $(Pc_n) = c_n$, and that for any x for which $(Px) = x$, it follows that x is a Church numeral.

125. **Exercise 46:** Show that for any λ-expression P, there exists a λ-expression Q, such that Q is the least fixed point of (PQ).

126. Our derivation of $\mathrm{Y}_{\text{Curry}}$ was a "bottom-up" extraction or factoring-out of a recursion-maker out of a recursive function, through a series of transformations. This is a low-level and limited view of fixed-point combinators. Luckily, we are now able to characterize the bahaviour of expressions such as $\mathrm{Y}_{\text{Curry}}$ in terms of fixed points. The insights we gain from this broader characterization will help us, among other things, construct many more "recursion makers" as well as find for such expression additional uses that are not directly related to defining recursive functions.

127. For any λ-term F, a λ-term x is a fixed point of F if and only if $Fx = x$.

128. A λ-term Φ is said to be a fixed-point combinator if and only if for any λ-term F, (ΦF) is a fixed point of F. We can now substitute (ΦF) for x in the equation that defines a fixed point, and get that Φ is a fixed-point combinator if and only if it satisfies the following relation for any λ-term F:

$$\Phi F \quad = \quad F(\Phi F)$$

129. We define recursive functions using fixed-point combinators by finding a λ-term F, the least fixed-point of which is the recursive function, and obtaining this least fixed point by applying a fixed-point combinator to F.

130. The English logician and mathematician *Alan Mathison Turing* (June 23 1912 — June 7 1954) discovered a different single fixed-point combinator:

$$\begin{aligned} \mathrm{Y}_{\text{Turing}} \quad = \quad & ((\lambda xf.f(xxf)) \\ & (\lambda xf.f(xxf))) \end{aligned}$$

If you are looking for some intuition behind the construction of $\mathrm{Y}_{\text{Turing}}$, think of the terms $\lambda xf.f(xxf)$ as "carrying f along with them" from one application the other, whereas in $\mathrm{Y}_{\text{Curry}}$, we have terms $\lambda x.f(xx)$ that "do not carry f around", but rather have f fixed from one application to the other.

131. **Exercise 47:** Show that $\mathrm{Y}_{\text{Turing}}$ is indeed a fixed-point combinator, i.e., that it satisfies the fixed-point equation.

132. In order to encode $\mathrm{Y}_{\text{Turing}}$ in the Scheme, which is applicative-order, we need to η-expand the two expressions (xxf) that occur in its definition:

Figure 9: Alan Mathison Turing

Figure 10: Dana Stewart Scott

```
(define Yturing
  ((lambda (x)
     (lambda (f)
       (f (lambda args
            (apply ((x x) f) args)))))
   (lambda (x)
     (lambda (f)
       (f (lambda args
            (apply ((x x) f) args)))))))
```

133. **Exercise 48:** Verify that you can use the Scheme version of Y_{TURING} to compute recursive functions.

134. *Scott numerals.* The logician *Dana Scott* (October 11, 1932 – *present*) defined

the following numeral system:

$$\begin{array}{rcl} s_0 & = & \lambda xy.x \\ s_1 & = & \lambda xy.y(\lambda xy.x) \\ & \ldots & \\ s_{n+1} & = & \lambda xy.ys_n \\ S^+_{\text{SCOTT}} & = & \lambda nxy.yn \end{array}$$

Just as Church numerals encapsulate the idea of a bounded composition as a form of *iteration*, Scott numerals encapsulate the idea of *selection*. Notice that s_0 ignores its second argument (and returns the first), while s_{n+1} ignores its first argument. Specifically:

$$\begin{array}{rcl} (s_0\ \langle \mathit{ReturnIfZero}\rangle\ \langle \mathit{ApplyIfNonZero}\rangle) & = & \langle \mathit{ReturnIfZero}\rangle \\ (s_{n+1}\ \langle \mathit{ReturnIfZero}\rangle\ \langle \mathit{ApplyIfNonZero}\rangle) & = & (\langle \mathit{ApplyIfNonZero}\rangle\ s_n) \end{array}$$

This is a natural mechanism for testing for zero, and if non-zero, applying a given function $\langle \mathit{ApplyIfNonZero}\rangle$ to *the predecessor of the number*. In itself, this is not a mechanism for iteration, but it works well in conjunction with a general iteration mechanism, such as a fixed-point combinator, because many number-theoretic functions are defined in that way (though not all!).

135. *Addition on Scott numerals.*

$$(+_{\text{SCOTT}}\ a\ b) \quad = \quad (a\ b\ (\lambda n.S^+_{\text{SCOTT}}\ (+_{\text{SCOTT}}\ n\ b)))$$

The above definition is recursive, and we can convert it into a fixed-point equation, and solve it with a fixed-point combinator Φ:

$$+_{\text{SCOTT}} \quad = \quad (\Phi\ (\lambda fab.(a\ b\ (\lambda n.S^+_{\text{SCOTT}}\ (f\ n\ b)))))$$

136. **Exercise 49:** Define the combinators $C^{\text{SCOTT}}_{\text{CHURCH}}, C^{\text{CHURCH}}_{\text{SCOTT}}$, that convert between Church and Scott numerals.

137. **Exercise 50:** Define $\text{Fact}_{\text{SCOTT}}$ *directly*, i.e., without resorting to $C^{\text{SCOTT}}_{\text{CHURCH}}, C^{\text{CHURCH}}_{\text{SCOTT}}$.

4 Syntax, Terms & Variables (§138–161)

It is high time we introduce the syntax of the λ-calculus more formally. In this section we will treat *terms* and variables. This material is somewhat technical, dry and boring, but needs to be covered so that we can make some finer observations than those we've been able to make so far. Brace yourself; Most of this material will pass quickly.

138. *Variables.* Let Vars be an infinite set of symbols or letters, which we shall call *variables*. It doesn't matter how we write the terms in Vars. Some ultra-formal texts like to index a single letter, as in $x_1, x_2, \ldots$, while most others just assume a set of symbols of some sort, e.g., $a, b, c, \ldots$. This doesn't really matter, since *names* are only a convenience in the λ-calculus. We are always able to replace names with numbers, à la de Bruijn numbers. In fact, de Bruijn numbering was originally invented in the context of the λ-calculus, and then extended to Scheme and other programming languages (See § 177). We can remove all variables altogether from our language, by shifting to *combinatory logic*. More on that option later. For the time being, we will assume that whenever we need a variable name, it is miraculously available to us in the set Vars.

139. *A fresh variable.* In Scheme, whenever we needed to generate a new variable name, one about which we could be sure that it is not used anywhere else in the program, when we needed a "fresh variable", we used the procedure `gensym` to generate a new, uninterned symbol. There is no such facility in the λ-calculus, but in the meta-language of the λ-calculus[8] we can use the Greek lowercase letter ν (pronounced "nu" and meant to remind of "new") in order to refer to a "fresh variable". The Greek letter ν has, by now, become synonymous for a fresh variable. If you need more than one such variable, you had better index ν, as in $\nu_1, \nu_2, \ldots$ rather than invent a new convention.

140. The set of λ-terms Λ is defined as the smallest set W that satisfies the following conditions:

- $\mathsf{Vars} \in \mathsf{W}$
- For all $M, N \in W$: $(MN) \in \mathsf{W}$
- For all $x \in \mathsf{Vars}, M \in \mathsf{W}$: $(\lambda x.M) \in \mathsf{W}$

In set theory, we would use *transfinite induction*, or induction over the *ordinal numbers* to define this set:

$$\begin{aligned} \Lambda_0 &= \mathsf{Vars} \\ \Lambda_{n+1} &= \Lambda_n \cup \{(MN) : M, N \in \Lambda_n\} \\ &\quad \cup \{(\lambda x.M) : x \in \mathsf{Vars}, M \in \Lambda_n\} \\ \Lambda \ (= \ \Lambda_\omega) &= \bigcup_{\alpha \in \omega} \Lambda_\alpha \end{aligned}$$

But don't worry if you haven't studied ordinal numbers or if the last definition doesn't make all that much sense to you. Just go by the previous definition of Λ as the *smallest set* W that has the properties listed above, and you'll be fine.

141. The corresponding definition in ML defines an abstract syntax for the representation of λ-expressions in ML. Such a definition is necessary if we wish to write computer programs that manipulate λ-expressions:

```
datatype Expr = Var of string
              | Applic of Expr * Expr
              | Lambda of string * Expr;
```

If you would like to experiment with the syntax of the λ-calculus, you may use the following scanner & parser, written in Standard ML: `http://www.little-lisper.org/website/files/lambda-calculus-parser.sml`. You can use it to work on the exercises below.

142. We will allow ourselves to relax some of the syntactic rules that define λ-terms, as long as this will not lead to any misunderstanding. For example, we will drop parenthesis whenever possible, associate applications to the *left*, and Curry λ-abstractions to the *right*.

[8] For any formal language L, the meta-language of L is the language we use for discourse about L. For example, the language of *vectors*, which is a subset of the language of mathematics, involves the ability to talk about ordered n-tuples, for any specific n. The ability to talk about ordered n-tuples, where n is a variable, requires us to use ellipses $(\cdots)$ notation, as in $x_1, \ldots, x_n$, and strictly speaking, is outside the language of vectors. Ellipses are an example of a meta-linguistic construct. The meta-language of mathematics contains many other such examples.

143. **Examples:**

We may write	in place of
ABC	$((AB)C)$
$\lambda xy.x$	$\lambda x.\lambda y.x$
$\lambda abc.ac(bc)$	$\lambda a.\lambda b.\lambda c.((ac)(bc))$

144. For any λ-term M, the set of variables *that appear in a λ-term* is denoted by $\mathsf{Vars}(M)$, and is defined as follows:

$$\begin{aligned}
\mathsf{Vars}(x) &= \{x\}\text{, where } x \in \mathsf{Vars} \\
\mathsf{Vars}(MN) &= \mathsf{Vars}(M) \cup \mathsf{Vars}(N) \\
\mathsf{Vars}(\lambda x.M) &= \mathsf{Vars}(M)
\end{aligned}$$

Notice that parameters to λ-abstractions are not counted in the definition of $\mathsf{Vars}(M)$ unless they actually appear as variable occurences within the body of the abstraction.

145. Below are some examples of the Vars function:

$$\begin{aligned}
\mathsf{Vars}(\lambda x.x) &= \{x\} \\
\mathsf{Vars}(\lambda x.xy) &= \{x, y\} \\
\mathsf{Vars}(\lambda x.x(\lambda x.x)) &= \{x\}
\end{aligned}$$

146. **Exercise 51:** Implement the Vars function using the abstract-syntax tree defined in § 141.

147. For any λ-term M, the set of free variables *that appear in a λ-term* is denoted by $\mathsf{FreeVars}(M)$, and is defined as follows:

$$\begin{aligned}
\mathsf{FreeVars}(x) &= \{x\}\text{, where } x \in \mathsf{Vars} \\
\mathsf{FreeVars}(MN) &= \mathsf{FreeVars}(M) \cup \mathsf{FreeVars}(N) \\
\mathsf{FreeVars}(\lambda x.M) &= \mathsf{FreeVars}(M) - \{x\}
\end{aligned}$$

148. Below are some examples of the $\mathsf{FreeVars}$ function:

$$\begin{aligned}
\mathsf{FreeVars}(\lambda x.x) &= \emptyset \\
\mathsf{FreeVars}(\lambda x.xy) &= \{y\} \\
\mathsf{FreeVars}(\lambda xy.xz(yz)) &= \{z\} \\
\mathsf{FreeVars}(\lambda xyz.xz(yz)) &= \emptyset
\end{aligned}$$

149. **Exercise 52:** Implement the $\mathsf{FreeVars}$ function using the abstract-syntax tree defined in § 141.

150. For any λ-term M, the set of bound variables *that appear in a λ-term* is denoted by $\mathsf{BoundVars}(M)$, and is defined as follows:

$$\begin{aligned}
\mathsf{BoundVars}(M) &= \mathsf{BoundVars}'(M, \emptyset) \\
\mathsf{BoundVars}'(x, \mathsf{S}) &= \begin{cases} \{x\} & \text{if } x \in \mathsf{S} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{BoundVars}'((MN), \mathsf{S}) &= \mathsf{BoundVars}'(M, \mathsf{S}) \cup \mathsf{BoundVars}'(N, \mathsf{S}) \\
\mathsf{BoundVars}'((\lambda x.M), \mathsf{S}) &= \mathsf{BoundVars}'(M, \{x\} \cup \mathsf{S})
\end{aligned}$$

151. Below are some examples of the $\mathsf{BoundVars}$ function:

$$\begin{aligned}
\mathsf{BoundVars}(\lambda x.x) &= \{x\} \\
\mathsf{BoundVars}(\lambda x.xy) &= \{x\} \\
\mathsf{BoundVars}(\lambda xy.xz(yz)) &= \{x, y\}
\end{aligned}$$

152. **Exercise 53:** Implement the $\mathsf{BoundVars}$ function using the abstract-syntax tree defined in § 141.

153. **Exercise 54:** Construct an expression in which the variable x occurs both free and bound.

154. **Exercise 55:** Prove that there exists no λ-expression that has no variables.

155. A λ-term M is a *combinator* if and only if $\mathsf{FreeVars}(M) = \emptyset$. The set of all combinators is written as Λ^0, and can be defined as follows:

$$\Lambda^0 = \{M \in \Lambda : \mathsf{FreeVars}(M) = \emptyset\}$$

Most of the terms we will be working with in this course will be combinators.

156. The *length* of a λ-term. The length of a term M is defined inductively as follows:

$$\begin{aligned}
\|\nu\| &= 1 \\
\|\lambda\nu.M\| &= 1 + \|M\| \\
\|(MN)\| &= 1 + \|M\| + \|N\|
\end{aligned}$$

157. Below are some examples of the *length* function:

$$\begin{aligned}
\|\lambda x.x\| &= 2 \\
\|\lambda xy.x\| &= 3 \\
\|\lambda xyz.xz(yz)\| &= 10
\end{aligned}$$

158. **Exercise 56:** Implement the above *length* function over the abstract-syntax tree defined in § 141.

159. **Exercise 57:** Write a function that takes an integer argument n, and returns the list of all combinators of length n. Use the abstract-syntax tree defined in § 141.

160. **Exercise 58:** Write a function that takes an integer argument n, and returns the number of combinators of length n. Your function should work for $n = 40$. Make sure you use a language like Scheme or Java that supports *large integers*.

161. **Exercise 59:** Modify your code in § 160 so that it returns a value for $n = 300$ within a reasonable time.

162. The $\lambda\mathbf{K}\beta\eta$-calculus

ADD SOMETHING HERE !!!

163. The $\lambda\mathbf{I}\beta\eta$-calculus

ADD SOMETHING HERE !!!

5 Reduction (§164–188)

164. Up to now, there is a gap in our presentation: We have been using the syntax of the λ-calculus, and the semantics of Scheme. This means that we assumed that λ-expressions are evaluated in the same way as Scheme expressions. Most of our journey thus far, we have managed to get away with this kind of sloppiness. There were only two places at which there were significant differences between the semantics of Scheme and the λ-calculus, to the point where we were unable to encode expessions in the λ-calculus directly in Scheme:

    - Conditional expressions
    - Fixed-point combinators

    At those points,we made some changes in the expressions we used, "so that they might run in Scheme".

    This gap was intentional. Our reasoning was that we wanted to let you "try before you buy", i.e., enjoy some of the things that can be done with the λ-calculus before you were required to pay any heavy tuition of formal notation, definitions, and semantics. We have now reached the point where it is best to close this gap.

165. In other words, our understanding of the λ-calculus is at the notational level. All we can do is define terms. We cannot yet compute anything given what we know about the λ-calculus. To compute or evaluate, we needed to convert our λ-terms into Scheme expressions and use a Scheme evaluator.

166. The notion of computing in the λ-calculus is based on *reduction.* Reduction can be thought of as a rule, by which one λ-term may be replaced by another λ-term. In some situations, there may be more than one way to reduce an expression. The choice of how to reduce an expression, i.e., which reduction to apply at any given opportunity is called a *reduction strategy.* Different reduction strategies have different properties: For a given λ-expression, some reduction strategies may be more efficient than others, some may terminate sooner, others may never terminate.

    When no further reduction is possible, we have a normal form. This corresponds to saying, with respect to Scheme, that evaluation of a Scheme expressions has terminated, and that we have a value.

    As we shall see by the end of this section, for any λ-term, all reductions that do terminate will end up with the same result. Put otherwise, it is not possible to start with a single λ-term and arrive at different values using different reduction strategies. This property is known as *confluence.*

167. **Context.** Think of the syntactic relationship between some λ-expression and one of its sub-expressions. For example, consider the boxed sub-expression in:

    $$((\lambda x.xx)\ \boxed{(\lambda x.xx)})$$

    Think of "pulling out" the boxed expression, giving the boxed expression

    $$\boxed{(\lambda x.xx)}$$

    and the *context*:

    $$((\lambda x.xx)\ \boxed{})$$

    You can think of a context as a λ-expression with a *single* "hole" in it. The requirement that there be only one hole is needed so that the question of how/which hole to fill becomes unambiguous.

168. Notationally, we denote the context as $\mathcal{C}[\![\ ]\!]$. The original expression is recovered when the original boxed expression is put back into the original context. We can, however, put other expressions into that context.

169. **Example:** Let $\Omega = ((\lambda x.xx)\ \boxed{(\lambda x.xx)})$. Then the context of the boxed expression is $\mathcal{C}[\![\ ]\!] = ((\lambda x.xx)\ \boxed{})$. The expression $\mathcal{C}[\![(\lambda x.xx)]\!]$ is syntactically identical to the original Ω.

170. **Compatibility.** A relation R is *compatible* if for any $\langle P, Q\rangle \in R$, and any context $\mathcal{C}[\![\ ]\!]$, we have $\langle \mathcal{C}[\![P]\!], \mathcal{C}[\![Q]\!]\rangle \in R$.

171. In the most abstract sense, reduction is simply a *compatible* binary relation over $\Lambda \times \Lambda$. If $R \subseteq \Lambda \times \Lambda$ is a binary relation, and for $P, Q \in \Lambda$, we have $\langle P, Q\rangle \in R$, then we may write $P \longrightarrow_R Q$, and say that "P reduces to Q".

172. When the relationship R is implied from the context, we can drop it, and simply write $P \longrightarrow Q$. It is only when dealing with several different notions of reduction, each corresponding to a different relationship, that we need to mention the relationship explicitely.

173. In practice, we are most often interested in situations where P reduces to Q after *any number of reductions*. Notationally, we can write this as $P \longrightarrow \cdots \longrightarrow Q$, but this is cumbersome, and the common abbreviation for this is simply $P \twoheadrightarrow Q$. This can be pronounced "P double-arrow Q". The relation $\twoheadrightarrow$ is simply the transitive and reflexive closure of $\longrightarrow$. Reflexivity means that $\twoheadrightarrow$ includes *zero* reductions, so while $P \longrightarrow P$ is not a trivial property of some expressions, the property $P \twoheadrightarrow P$ is satisfied by all expressions trivially. Another way to distinguish between $\longrightarrow$ and $\twoheadrightarrow$, is to refer to the first as the *one-step reduction*, and to the second as the *multi-step reduction*.

174. The relationship $=_R$ is reflexive, symmetric, transitive closure of $\longrightarrow_R$. Another way of thinking about $=_R$ is to say that it is the equivalence relation induced by $\longrightarrow_R$. The statement $P =_R Q$ is pronounced "P is equal to Q modulo R", or "P is equal modulo R to Q". Once again, we may drop the subscript R from the relation $=_R$ when R is clear from the context. **However,** beware that it is easy to confuse equality modolo a relation with textual equality.

175. A useful notion that is not in itself a reduction, but that is necessary for some reductions, is that of **α-equivalence**. Alpha-equivalence is an equivalence relation modulo bound variable names.

176. We would like to capture the intuition that in some sense, e.g., $\lambda xy.x$ and $\lambda yx.y$ are two syntactic representations for the same expression. According to this notion of equivalence we can think of a λ-expression as a directed graph, in which each bound variable occurance is a node from which an arrow points to the λ-abstraction that introduces the given bound variable. This notion of equivalence, known as α-equivalence.

177. One way to formalise α-equivalence is to use the notion of *lexical addressing*, introduced in the *Compiler Construction* course. Lexical addressing was originally defined in the λ-calculus by the Dutch mathematician Nicolaas Govert de Bruijn (July 9 1918 — February 17 2012), and is known as *de Bruijn numbering*. It was later exteded to programming languages. In fact, de Bruijn

Figure 11: Nicolaas Govert de Bruijn

numbering is somewhat simpler than lexical addressing over Scheme programs. Let $\mathcal{L}[\![\ ]\!]$ be defined as follows:

$$\begin{array}{rcl}
\mathcal{L}[\![E]\!] & = & \mathcal{L}[\![E, \emptyset, 0]\!] \\
\mathcal{L}[\![\nu, \langle\{\nu \mapsto r\}, \mathcal{M}\rangle, n]\!] & = & r \\
\mathcal{L}[\![\nu, \langle\{\nu' \mapsto r\}, \mathcal{M}\rangle, n]\!] & = & \mathcal{L}[\![\nu, \mathcal{M}, n]\!], \quad \text{for } \nu \neq \nu' \\
\mathcal{L}[\![(PQ), \mathcal{M}, n]\!] & = & (\mathcal{L}[\![P, \mathcal{M}, n]\!]\ \mathcal{L}[\![Q, \mathcal{M}, n]\!]) \\
\mathcal{L}[\![(\lambda\nu.P), \mathcal{M}, n]\!] & = & (\boldsymbol{\lambda n.}\mathcal{L}[\![P, \langle\{\nu \mapsto n\}, \mathcal{M}\rangle, n+1]\!])
\end{array}$$

Below are some examples of the de Bruijn numbering:

$$\begin{array}{rcl}
\mathcal{L}[\![\lambda x.x]\!] & = & \boldsymbol{\lambda 0.0} \\
\mathcal{L}[\![\lambda x.xx]\!] & = & \boldsymbol{\lambda 0.00} \\
\mathcal{L}[\![\lambda xy.x]\!] & = & \boldsymbol{\lambda 01.0} \\
\mathcal{L}[\![\lambda yx.y]\!] & = & \boldsymbol{\lambda 01.0} \\
\mathcal{L}[\![\lambda xyz.xz(yz)]\!] & = & \boldsymbol{\lambda 012.02(12)}
\end{array}$$

The de Bruijn numbering of a λ-expression is some expression in a syntax related to, but not the same as that of λ-expressions.

178. **Exercise 60:** The above definition of de Bruijn numbers is given using an encoding for a *lexical environment*. Let $\{\nu \mapsto r\}$ be a variable binding, and M be an environment. Why did we use the ordered pair $\langle\{\nu \mapsto r\}, M\rangle$ to denote the *extended environment*, rather than set union: $\{\nu \mapsto r\} \cup M$?

179. Combinators P, Q are α-equivalent if they have the same de Bruijn numbering. Put otherwise, $P =_\alpha Q$ iff $\mathcal{L}[\![P]\!] = \mathcal{L}[\![Q]\!]$.

180. This definition fails for λ-expressions that are not combinators. We can change the definition of $\mathcal{L}[\![\ ]\!]$ to accomodate all λ-terms by replacing the line

$$\mathcal{L}[\![E]\!] \quad = \quad \mathcal{L}[\![E, \emptyset, 0]\!]$$

with

$$\mathcal{L}[\![E]\!] \quad = \quad \mathcal{L}[\![E, \mathcal{M}_{\text{fv}}, 0]\!]$$

where $\mathcal{M}_{\text{fv}}$ maps all variable names to themselves.

181. The rest of this section introduces the specific relations we will be working with in our reductions. There really are only two such relations: β-reduction, which captures application under the substitution model, and η-reduction, which was introduced intuitively in Section 3.

182. Expressions of the form $((\lambda\nu.P)Q)$ are called *redexes*,[9] and can be replaced with expressions of the form $P[\nu := Q]$, i.e., P in which every free occurance of the variable ν has been replaced by Q.

183. **$\boldsymbol{\beta}$-reduction.** Beta reduction captures the notion of application. It is defined as the smallest compatible relations over $\Lambda \times \Lambda$, that has the subset

$$\{\langle((\lambda x.P)Q), P[x := Q]\rangle : P, Q \in \Lambda, x \in \mathsf{Vars}\}$$

184. You can now see the purpose of requiring that the reduction be compatible. Let P be some λ-expression that contains a redex, such that:

$$\begin{aligned} P &= \mathcal{C}[\![((\lambda x.M)N)]\!] \\ P' &= \mathcal{C}[\![M[x := N]]\!] \end{aligned}$$

It follows from the compatibility of $\boldsymbol{\beta}$ that $\langle P, P'\rangle \in \boldsymbol{\beta}$. Ordinarily we would write $P \longrightarrow_\beta P'$.

185. **$\boldsymbol{\eta}$-reduction.** Eta reduction was explained already in Section 3 (and mainly in § 105). We define it formally as the set

$$\eta \quad = \quad \{\langle \lambda x.Mx, M\rangle : M \in \Lambda, x \notin \mathsf{FreeVars}(M)\}$$

186. **$\boldsymbol{\beta\eta}$-reduction.** This is a reduction step that consists of either a β-step or an η-step. We define it formally as $\beta\eta = \beta \cup \eta$.

187. **Exercise 61:** For any λ-expression P, if $P \twoheadrightarrow P$ in one or more $\beta\eta$-steps, we say that P is *interesting*. Show there are infinitely-many distinct interesting expressions, i.e., expressions P for which

$$P \quad \underbrace{\longrightarrow \cdots \longrightarrow}_{>0} P$$

[9] The term 'redex' is an abbreviation for *red*ucible *ex*pression, and contrary to popular belief, is not a Latin term. There is no difficulty, however, in regarding it as a Latin word, because it is *related* to many Latin words, and has the structure of a Latin noun of the third declension. As a Latin word, it should be declined as follows:

	Singular	Plural
Nominative	**rēdex**	**rēdicēs**
Genitive	**rēdicis**	**rēdicum**
Dative	**rēdicī**	**rēdicibus**
Accusative	**rēdicem**	**rēdicēs**
Ablative	**rēdice**	**rēdicibus**
Vocative	**rēdex**	**rēdicēs**

A 'little redex' would be **rēdiculus**! ☺

188. **Exercise 62:** (Barendregt) For any $n \in \mathbb{N}$, show how to create a non-trivial sequence $E_1, \ldots, E_n$ of n terms, such that

$$E_1 \twoheadrightarrow E_2 \twoheadrightarrow E_3 \cdots \twoheadrightarrow E_n \twoheadrightarrow E_1$$

$$\boxed{\text{TO BE CONTINUED...}}$$

- order of evaluation
- CR1
- CR2
- normal order of evaluation
- applicative order of evaluation

6 Fixed Points II (§189–222)

189. We now resume our study of fixed points and fixed-point combinators. An interesting question to consider is how many fixed-point combinators are there.

190. How many fixed-point combinators are there? At this point, we have seen two fixed-point combinators, of Curry's, $\mathrm{Y}_{\text{Curry}}$, and of Turing's, $\mathrm{Y}_{\text{Turing}}$. But fixed-point combinators are defined *implicitely* using the relation in § 128, and any λ-expression that will satisfy this relation is a fixed-point combinator. We have no reason to assume that $\mathrm{Y}_{\text{Curry}}, \mathrm{Y}_{\text{Turing}}$ are the only ones.

191. In fact, it is straightforward to show that there are infinitely-many fixed-point combinators. Consider again the relation that defines fixed-point combinators:

$$\Phi F \quad = \quad F(\Phi F)$$

We can rewrite it as

$$\Phi \quad = \quad \lambda f.f(\Phi f)$$

Now if you forget what Φ is "supposed" to be, and just focus on the text of the definition, it will become readily apparent that Φ is a recursive function, since the body of Φ contains the free variable Φ. Therefore, we can define Φ as the least fixed point of the following equation:

$$\Phi \quad = \quad (\boxed{(\lambda \phi f.f(\phi f))}\ \Phi)$$

And we can therefore define Φ by applying any fixed-point combinator Ψ, to the above boxed expression:

$$\Phi \quad = \quad \Psi(\lambda \phi f.f(\phi f))$$

In other words, you need one fixed-point combinator in order to get another... Luckily, we have a reliable fixed-point combinator obtained independently, namely, $\mathrm{Y}_{\text{Curry}}$.

Let $M = \lambda \phi x.x(\phi x)$. We can now define the following infinite sequence inductively:

$$\begin{aligned} \Phi_0 &= \mathrm{Y}_{\text{Curry}} \\ \Phi_{n+1} &= \Phi_n\ M \qquad \text{for all } n \geq 0 \end{aligned}$$

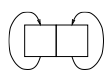
We still need to show that these terms are distinct, namely, that if $i \neq j$ then $\Phi_i \neq \Phi_j$.

TO BE CONTINUED...

192. We have seen the use of fixed-point combinators in defining recursive functions. These turned out to be the least fixed-points of the corresponding [higher-order] functional. We shall now see other uses of fixed-point combinators.

193. **Circular structures.** A circular data structure is one that contains a pointer or link to itself. Circular structures can be visualised as directed graphs that contain a loop. They can be implemented in a programming language using side-effets to place within a data structure a pointer or link to an element higher-up in the structure. Below are some examples of circular structures and the code that defines them in Scheme.

194. **Example:**



```
(let ((m `(m . m)))
  (set-car! m m)
  (set-cdr! m m)
  m)
```

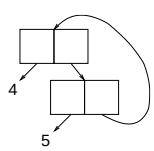
We define the above structure in the λ-calculus is by describing it as the least solution to some fixed-point equation, and then using any fixed-point combinator to solve that equation. Let M be the above structure. It is clear that $M = \langle M, M\rangle$. We can now abstract M out of the right-hand side and get a fixed-point equation:

$$\begin{aligned} M &= \langle M, M\rangle \\ &= (\boxed{(\lambda m.\, \langle m, m\rangle)}\ M) \end{aligned}$$

We now define M using $\mathrm{Y}_{\textsc{Curry}}$:

$$M = (\mathrm{Y}_{\textsc{Curry}}\ (\lambda m.\, \langle m, m\rangle))$$

195. **Example:**




```
(let ((m `(4 . (5 . m))))
  (set-cdr! (cdr m) m)
  m)
```

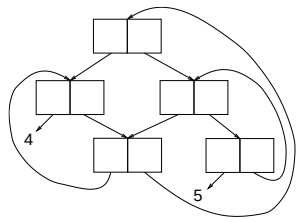
We define the above structure in the λ-calculus is by describing it as the least solution to some fixed-point equation, and then using any fixed-point combinator to solve that equation. Let M be the above structure. It is clear that

$$\begin{aligned} M &= \langle c_4, \langle c_5, M\rangle\rangle \\ &= (\boxed{(\lambda m.\, \langle c_4, \langle c_5, m\rangle\rangle)}\ M) \end{aligned}$$

We now define M using $\mathrm{Y}_{\textsc{Curry}}$:

$$M = (\mathrm{Y}_{\textsc{Curry}}\ (\lambda m.\, \langle c_4, \langle c_5, m\rangle\rangle))$$

196. **Example:**




```
(let ((m `((4 . (a-m . m)) .
           (da-m . (5 . d-m)))))
  (set-car! (cdar m) (car m))
  (set-cdr! (cdar m) m)
  (set-car! (cdr m) (cdar m))
  (set-cdr! (cddr m) (cdr m))
  m)
```

We define the above structure in the λ-calculus is by describing it as the least solution to some fixed-point equation, and then using any fixed-point combinator to solve that equation. Let M be the above structure. It is clear that

$$\begin{aligned} M &= \left\langle\left\langle c_4, \left\langle(\pi_1^2 M, M)\right\rangle\right\rangle, \left\langle(\pi_2^2(\pi_1^2 M)), \left\langle c_5, (\pi_2^2 M)\right\rangle\right\rangle\right\rangle \\ &= (\boxed{(\lambda m. \left\langle\left\langle c_4, \left\langle(\pi_1^2 m, m)\right\rangle\right\rangle, \left\langle(\pi_2^2(\pi_1^2 m)), \left\langle c_5, (\pi_2^2 m)\right\rangle\right\rangle\right\rangle)}\ M) \end{aligned}$$

We now define M using $\mathrm{Y}_{\textsc{Curry}}$:

$$M = (\mathrm{Y}_{\textsc{Curry}}\ (\lambda m. \left\langle\left\langle c_4, \left\langle(\pi_1^2 m, m)\right\rangle\right\rangle, \left\langle(\pi_2^2(\pi_1^2 m)), \left\langle c_5, (\pi_2^2 m)\right\rangle\right\rangle\right\rangle))$$

197. **Multiple fixed points.** A natural extension of fixed-point equations is the multiple fixed-point equations. Just as recursive functions are the least fixed points of single fixed-point equations, so are mutually-recursive functions the least fixed points of sets of multiple fixed-point equations.

198. Circular data structures can also be defined as the least multiple fixed-points of multiple fixed-point equations. For example, § 196 contained multiple cycles. We defined internal cycles by describing a path that leads to a node in the cycle *through the top node*, which made our definition rather complex: Cycles that were small but deep were defined from the top element, resulting in longer, more complex definitions that are more prone to errors. Being able to "name" various points in a circular data structure and describe it using all these names, rather than just the name of the entire data structure will considerably simplify the task of defining such structures. In § 213 we will redefine the structure introduced in § 196 using multiple fixed-point equations.

199. Even though the situation with n multiple fixed-point equations is a natural extension of the situation with one equation, the expressions involved tend to be very large, contain several levels of ellipsis ('$\cdots$')[10], and can initially be difficult to follow. Therefore, we shall always consider the case where $n = 2$ before moving on to the general case.

200. The definition of two multiple fixed-point equations is as follows: Given two λ-terms F, G, their multiple fixed points x, y are two λ-terms x, y that satisfy:

$$\begin{aligned} x &= Fxy \\ y &= Gxy \end{aligned}$$

201. The definition of a system of n multiple fixed-point equations is as follows: Given n λ-terms $F_1, \ldots, F_n$, their multiple fixed points are the n λ-terms

[10] Ellipsis are a meta-mathematical shorthand for an inductively-defined syntax. Multiple levels of ellipsis correspond to *nested* inductive definitions.

$x_1, \ldots, x_n$ that satisfy:

$$\begin{array}{rcl} x_1 & = & F_1 x_1 \cdots x_n \\ x_2 & = & F_2 x_1 \cdots x_n \\ & \cdots & \\ x_n & = & F_n x_1 \cdots x_n \end{array}$$

202. The expressions Φ_1^2, Φ_2^2 are two multiple least fixed-point combinators if for any λ-terms F, G, the terms x, y defined below

$$\begin{array}{rcl} x & = & (\Phi_1^2 FG) \\ y & = & (\Phi_2^2 FG) \end{array}$$

are the two least fixed-point combinators of F, G.

203. The definition commonly found in the literature has x, y replaced by their respective definitions, giving the utterly terse definition:

204. **Alternate Definition:** The expressions Φ_1^2, Φ_2^2 are two multiple least fixed-point combinators if they satisfy, for any λ-terms F, G, the following equations:

$$\begin{array}{rcl} (\Phi_1^2 FG) & = & F(\Phi_1^2 FG)(\Phi_2^2 FG) \\ (\Phi_2^2 FG) & = & G(\Phi_1^2 FG)(\Phi_2^2 FG) \end{array}$$

But this is just a more complicated way of stating § 202.

205. A set of n **multiple fixed-point combinators** is the set $\{\Phi_1^n, \ldots, \Phi_n^n\}$, such that for any n terms $F_1, \ldots, F_n$, we have:

Simple Way. Let $x_1, \ldots, x_n$ be defined as follows:

$$\begin{array}{rcl} x_1 & = & (\Phi_1^n F_1 \cdots F_n) \\ & \cdots & \\ x_n & = & (\Phi_n^n F_1 \cdots F_n) \end{array}$$

Then $x_1, \ldots, x_n$ are the n least fixed points of $F_1, \ldots F_n$.

Terse Way.

$$\begin{array}{rcl} (\Phi_1^n x_1 \cdots x_n) & = & (x_1(\Phi_1^n x_1 \cdots x_n) \\ & & \quad \cdots \\ & & \quad (\Phi_n^n x_1 \cdots x_n)) \\ & \cdots & \\ (\Phi_n^n x_1 \cdots x_n) & = & (x_n(\Phi_1^n x_1 \cdots x_n) \\ & & \quad \cdots \\ & & \quad (\Phi_n^n x_1 \cdots x_n)) \end{array}$$

206. Of course, the above are just definitions. We have not yet shown the existence of multiple fixed-point combinators. This is the topic of the next few items.

207. A construction that is common in set theory, is to use a circular data structure to define multiple fixed points in terms of a single fixed point, in a way similar to how we defined circular data structures in Items 194, 195, and 196.

For example, when defining a *pair* of recursive functions in this way, the cycle in the data structure "happens" because one of the functions within the data structure calls another function within the data structure and accesses it via the top of the data structure, i.e., the top *pair*.

208. **Example:** Let us define the structure $M = \langle \mathit{Even}?, \mathit{Odd}? \rangle$ using a single fixed-point combinator. The key observation is that within the definitions of $\mathit{Even}?, \mathit{Odd}?$ we can use $(\pi_1^2 M), (\pi_2^2 M)$ in place of $\mathit{Even}?, \mathit{Odd}?$, respectively, in the recursive definition of these functions:

$$\begin{aligned} M &= \langle (\lambda n.\mathit{Zero}?\ n\ \mathbf{True}\ (\boxed{(\pi_2^2\ M)}\ (P^-\ n))), \\ &\quad\ (\lambda n.\mathit{Zero}?\ n\ \mathbf{False}\ (\boxed{(\pi_1^2\ M)}\ (P^-\ n)))\rangle \\ &= ((\lambda m.\langle (\lambda n.\mathit{Zero}?\ n\ \mathbf{True}\ (\pi_2^2\ m\ (P^-\ n))), \\ &\qquad\qquad (\lambda n.\mathit{Zero}?\ n\ \mathbf{False}\ (\pi_1^2\ m\ (P^-\ n)))\rangle)\ M) \end{aligned}$$

We can now define M using any fixed-point combinator, e.g., $\mathrm{Y}_{\text{Curry}}$:

$$\begin{aligned} M &= (\mathrm{Y}_{\text{Curry}}\ (\lambda m.\langle (\lambda n.\mathit{Zero}?\ n\ \mathbf{True}\ (\pi_2^2\ m\ (P^-\ n))), \\ &\qquad\qquad (\lambda n.\mathit{Zero}?\ n\ \mathbf{False}\ (\pi_1^2\ m\ (P^-\ n)))\rangle)) \end{aligned}$$

Having just defined M, we can define the functions $\mathit{Even}?, \mathit{Odd}?$ as its first and second projections:

$$\begin{aligned} \mathit{Even}? &\equiv (\pi_1^2\ M) \\ \mathit{Odd}? &\equiv (\pi_2^2\ M) \end{aligned}$$

And we have just defined two mutually-recursive functions. The exact same approach can be used to define two multiple least fixed-point combinators.

209. **Multiple fixed-point combinators I (defined using circular structures).** We wish to define the structure $M = \langle \Phi_1^2, \Phi_2^2 \rangle$. We know that for all F, G, Φ_1^2, Φ_2^2 must satisfy

$$\begin{aligned} (\Phi_1^2 FG) &= F(\Phi_1^2 FG)(\Phi_2^2 FG) \\ (\Phi_2^2 FG) &= G(\Phi_1^2 FG)(\Phi_2^2 FG) \end{aligned}$$

or put otherwise:

$$\begin{aligned} \Phi_1^2 &= \lambda fg.f(\Phi_1^2 fg)(\Phi_2^2 fg) \\ \Phi_2^2 &= \lambda fg.g(\Phi_1^2 fg)(\Phi_2^2 fg) \end{aligned}$$

Now just forget for a moment that Φ_1^2, Φ_2^2 are multiple fixed-point combinators, and just look at the above definition. Clearly Φ_1^2, Φ_2^2 are just two mutually-recursive functions, and we can define them using a circular data structure in much the same way as we defined $\mathit{Even}?, \mathit{Odd}?$ in § 208.

As before, the key observation here is that within the respective definitions of Φ_1^2, Φ_2^2 we can use $(\pi_1^2 M), (\pi_2^2 M)$ in place of Φ_1^2, Φ_2^2:

$$\begin{aligned} M &= \langle (\lambda fg.f(\boxed{(\pi_1^2\ M)}\ fg)(\boxed{(\pi_2^2\ M)}\ fg)) \\ &\quad\ (\lambda fg.g(\boxed{(\pi_1^2\ M)}\ fg)(\boxed{(\pi_2^2\ M)}\ fg))\rangle \\ &= ((\lambda m.\langle (\lambda fg.f(\pi_1^2\ m\ fg)(\pi_2^2\ m\ fg)) \\ &\qquad\qquad (\lambda fg.g(\pi_1^2\ m\ fg)(\pi_2^2\ m\ fg))\rangle)\ M) \end{aligned}$$

We can see that M is the solution of a single fixed-point equation. Therefore, we can define M using a single fixed-point combinator. Let Φ be any *single* fixed-point combinator. We can define M as follows:

$$\begin{aligned} M &= (\Phi\ (\lambda m.\langle (\lambda fg.f(\pi_1^2\ m\ fg)(\pi_2^2\ m\ fg)) \\ &\qquad\qquad (\lambda fg.g(\pi_1^2\ m\ fg)(\pi_2^2\ m\ fg))\rangle)) \end{aligned}$$

Having just defined M, we can now define Φ_1^2, Φ_2^2 as its first and second projections:

$$\begin{aligned} \Phi_1^2 &= (\pi_1^2\ M) \\ \Phi_2^2 &= (\pi_2^2\ M) \end{aligned}$$

and this is our first definition for multiple fixed-point combinators.

210. We can now return to our original problem of defining mutually-recursive functions using multiple fixed-point combinators. We start with a pair of mutually-recursive functions, e.g., *Even*?, *Odd*?, defined as follows:

$$\begin{aligned} \mathit{Even}? &= \lambda n.\mathit{Zero}?\ n\ \mathbf{True}\ (\mathit{Odd}?\ (P^-\ n)) \\ \mathit{Odd}? &= \lambda n.\mathit{Zero}?\ n\ \mathbf{False}\ (\mathit{Even}?\ (P^-\ n)) \end{aligned}$$

Abstracing *Even*?, *Odd*? over the left-hand sides of both definitions, we get two multiple fixed point equations:

$$\begin{aligned} \mathit{Even}? &= (\boxed{(\lambda eon.\mathit{Zero}?\ n\ \mathbf{True}\ (o\ (P^-\ n)))}\ \mathit{Even}?\ \mathit{Odd}?) \\ \mathit{Odd}? &= (\boxed{(\lambda eon.\mathit{Zero}?\ n\ \mathbf{False}\ (e\ (P^-\ n)))}\ \mathit{Even}?\ \mathit{Odd}?) \end{aligned}$$

We can now define *Even*?, *Odd*? as the multiple fixed points of these equations using two multiple fixed-point combinators:

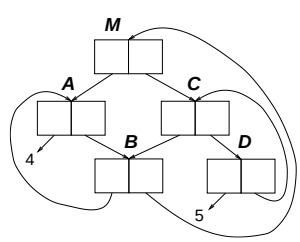
$$\begin{aligned} \mathit{Even}? &= (\Phi_1^2\ (\lambda eon.\mathit{Zero}?\ n\ \mathbf{True}\ (o\ (P^-\ n))) \\ &\qquad (\lambda eon.\mathit{Zero}?\ n\ \mathbf{False}\ (e\ (P^-\ n)))) \\ \mathit{Odd}? &= (\Phi_2^2\ (\lambda eon.\mathit{Zero}?\ n\ \mathbf{True}\ (o\ (P^-\ n))) \\ &\qquad (\lambda eon.\mathit{Zero}?\ n\ \mathbf{False}\ (e\ (P^-\ n)))) \end{aligned}$$

While these definitions are equivalent to the ones in § 208, the details of working with circular structures is now buried within our definitions of Φ_1^2, Φ_2^2. We have "paid our dues", so to speak, and can now enjoy a simpler, more direct construction of mutually-recursive functions.

211. **Exercise 63:** Extend the definitions of Φ_1^2, Φ_2^2 in § 209 to n multiple fixed-point combinators. You may assume the ordered n-tuple and its projections.

212. **Exercise 64:** Even though this is not a very interesting way to define single fixed-point combinators, verify that your definition in § 211 works for $n = 1$, and can be used to define a singularly-recursive function, such as *factorial* or *Ackermann's function*. Note that this fixed-point combinator will needlessly make use of a circular data structure.

213. **Example:** We are now ready to re-visit the structure defined in § 196, and define it using multiple fixed-point combinators. One difference is that rather than name only the top node and use it to define all other nodes, we name each node separately:



We define the structure using the following equations:

$$\begin{aligned} M &= \langle A, C\rangle \\ A &= \langle c_4, B\rangle \\ B &= \langle A, M\rangle \\ C &= \langle B, D\rangle \\ D &= \langle c_5, C\rangle \end{aligned}$$

We then convert the above equations to multiple fixed-point equations:

$$\begin{aligned} M &= (\boxed{\lambda mabcd.\, \langle a, c\rangle}\ MABCD) \\ A &= (\boxed{\lambda mabcd.\, \langle c_4, b\rangle}\ MABCD) \\ B &= (\boxed{\lambda mabcd.\, \langle a, m\rangle}\ MABCD) \\ C &= (\boxed{\lambda mabcd.\, \langle b, d\rangle}\ MABCD) \\ D &= (\boxed{\lambda mabcd.\, \langle c_5, c\rangle}\ MABCD) \end{aligned}$$

Notice that these definitions are inter-connected and circular. We can define each of the five nodes independently, using $\Phi_1^5, \ldots, \Phi_5^5$, though we are only interested in M:

$$\begin{aligned} M &= (\Phi_1^5\ (\lambda mabcd.\, \langle a, c\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_4, b\rangle) \\ &\qquad (\lambda mabcd.\, \langle a, m\rangle) \\ &\qquad (\lambda mabcd.\, \langle b, d\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_5, c\rangle)) \\ A &= (\Phi_2^5\ (\lambda mabcd.\, \langle a, c\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_4, b\rangle) \\ &\qquad (\lambda mabcd.\, \langle a, m\rangle) \\ &\qquad (\lambda mabcd.\, \langle b, d\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_5, c\rangle)) \\ B &= (\Phi_3^5\ (\lambda mabcd.\, \langle a, c\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_4, b\rangle) \\ &\qquad (\lambda mabcd.\, \langle a, m\rangle) \\ &\qquad (\lambda mabcd.\, \langle b, d\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_5, c\rangle)) \\ C &= (\Phi_4^5\ (\lambda mabcd.\, \langle a, c\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_4, b\rangle) \\ &\qquad (\lambda mabcd.\, \langle a, m\rangle) \\ &\qquad (\lambda mabcd.\, \langle b, d\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_5, c\rangle)) \\ D &= (\Phi_5^5\ (\lambda mabcd.\, \langle a, c\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_4, b\rangle) \\ &\qquad (\lambda mabcd.\, \langle a, m\rangle) \\ &\qquad (\lambda mabcd.\, \langle b, d\rangle) \\ &\qquad (\lambda mabcd.\, \langle c_5, c\rangle)) \end{aligned}$$

Having invested the effort to define Φ_k^n for all k, n such that $0 \leq k \leq n$, using multiple fixed-point combinators to define structures with multiple circularities is much simpler, and less prone to errors than doing the same with single fixed-point combinators.

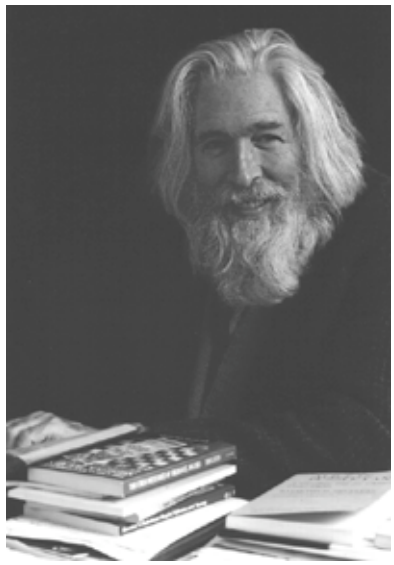


Figure 12: Raymond Merrill Smullyan

214. **Multiple fixed-point combinators II (Smullyan's construction).** Another way to define n multiple fixed-point combinators using single fixed-point combinators is due to the American logician *Raymond Merrill Smullyan* (1919 — present). The basic intuition is that for any $n \geq 2$, the equations defining Φ_k^n for $k = 1, \ldots, n$ are very similar. The idea is to factor out what all these expressions have in common.

215. We first consider the case where $n = 2$. We would like to define a λ-term Ψ such that:

$$\Psi ABC \quad = \quad C(\Psi ABA)(\Psi ABB)$$

It follows that

$$\begin{aligned} \Psi &= \lambda abc.c(\Psi aba)(\Psi abb) \\ &= ((\lambda \psi abc.c(\psi aba)(\psi abb))\ \Psi) \end{aligned}$$

We can now solve the fixed-point equation using any single fixed-point combinator Φ:

$$\Psi \quad = \quad (\Phi\ (\lambda \psi abc.c(\psi aba)(\psi abb)))$$

Recall that Ψ abstracts over what is common to Φ_1^2, Φ_2^2, so we now define these by specialising Ψ:

$$\begin{aligned} \Phi_1^2 &= \lambda fg.(\Psi fgf) \\ \Phi_2^2 &= \lambda fg.(\Psi fgg) \end{aligned}$$

216. **Exercise 65:** Verify that Φ_1^2, Φ_2^2, as defined in § 215 are indeed multiple fixed-point combinators.

217. **Exercise 66:** Extending Smullyan's fixed-point combinators for two multiple fixed points to n multiple fixed points: Let Φ be a single fixed-point combi-

nator, and Ψ_n be defined as follows:

$$\begin{aligned}\Psi_n \quad = \quad (\Phi\ (\lambda\psi x_1 \cdots x_n y.y(&\psi x_1 \cdots x_n x_1)\\ &(\psi x_1 \cdots x_n x_2)\\ &\cdots\\ &(\psi x_1 \cdots x_n x_n)))\end{aligned}$$

Use Ψ_n to define the multiple fixed-point combinators $\Phi_1^n, \ldots, \Phi_n^n$.

218. **Multiple fixed-point combinators III (extending Curry's fixed-point combinator).** Recall Curry's singular fixed-point combinator (§ 102):

$$\begin{aligned}\mathrm{Y}_{\textsc{Curry}} \quad = \quad \lambda f.((&\lambda x.f(xx))\\ &(\lambda x.f(xx)))\end{aligned}$$

Can we define an extension of it to n finding multiple fixed points? The answer depends on what we mean by "extension": What properties are we adding and what properties do we claim to be preserving.

Curry's fixed-point combinator is an abstraction over f that surrounds an application $((\lambda x.f(xx))(\lambda x.f(xx)))$. Both the procedure and the argument in this application are identical. The application results in a single application of f with an argument that is identical to the original application:[11]

$$\boxed{((\lambda x.\boldsymbol{f}(xx))(\lambda x.\boldsymbol{f}(xx)))} \quad = \quad (\boldsymbol{f}\boxed{((\lambda x.\boldsymbol{f}(xx))(\lambda x.\boldsymbol{f}(xx)))}) \tag{6}$$

We expect an extension of Curry's fixed-point combinator to abstract $f_1, \ldots, f_n$ over an application similar to Eq (6). We must realize that the application itself will need to be modified so as to pass around n multiple fixed points. A minimal construction that seems to capture these properties could be:

$$\begin{aligned}\mathrm{Y}_{\textsc{Curry}_1^n} \quad = \quad \lambda f_1 f_2 \cdots f_n.((&\lambda x_1 x_2 \cdots x_n.f_1(x_1 x_1 \cdots x_n)\\ &\qquad (x_2 x_1 \cdots x_n)\\ &\qquad \cdots\\ &\qquad (x_n x_1 \cdots x_n))\\ &(\lambda x_1 x_2 \cdots x_n.f_1(x_1 x_1 \cdots x_n)\\ &\qquad (x_2 x_1 \cdots x_n)\\ &\qquad \cdots\\ &\qquad (x_n x_1 \cdots x_n))\\ &(\lambda x_1 x_2 \cdots x_n.f_2(x_1 x_1 \cdots x_n)\\ &\qquad (x_2 x_1 \cdots x_n)\\ &\qquad \cdots\\ &\qquad (x_n x_1 \cdots x_n))\\ &(\lambda x_1 x_2 \cdots x_n.f_n(x_1 x_1 \cdots x_n)\\ &\qquad (x_2 x_1 \cdots x_n)\\ &\qquad \cdots\\ &\qquad (x_n x_1 \cdots x_n)))\end{aligned}$$

[11] We note f in boldbace because it occurs free in the expression.

$$
\begin{aligned}
\mathrm{Y}_{\textsc{Curry}_{2}^{n}} \quad = \quad & \lambda f_1 f_2 \cdots f_n.((\lambda x_1 x_2 \cdots x_n.f_2(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_1(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_2(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_n(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Y}_{\textsc{Curry}_{j}^{n}} \quad = \quad & \lambda f_1 f_2 \cdots f_n.((\lambda x_1 x_2 \cdots x_n.f_j(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_1(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_2(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_n(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)))
\end{aligned}
$$

$$\cdots$$

$$
\begin{aligned}
\mathrm{Y}_{\textsc{Curry}_{n}^{n}} \quad = \quad & \lambda f_1 f_2 \cdots f_n.((\lambda x_1 x_2 \cdots x_n.f_n(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_1(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_2(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)) \\
& \qquad\qquad\qquad\qquad (\lambda x_1 x_2 \cdots x_n.f_n(x_1 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_2 x_1 \cdots x_n) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x_n x_1 \cdots x_n)))
\end{aligned}
$$

The construction, while adding nothing that is not required by the problem, is nevertheless quite complex. To help notice how the sequence of multiple fixed-

point combinators is really constructed, we colored the variable j in $\mathrm{Y}_{\textsc{Curry}}{}^{n}_{j}$ both in the name of the combinator as well as in its definition. Please note that only one colored j appears in the body of each fixed-point combinator.

219. **Exercise 67:** In § 218, we presented the extension of Curry's fixed-point combinator to n multiple fixed points. This definition works great in the pure λ-calculus, where no order of evaluation is specified, or under the *normal order of evaluation* ("lazy evaluation"). It will not, however, work in *applicative order* semantics, and specifically, will diverge (i.e., go into an infinite loop) when typed in Scheme. Define an applicative-order version of the n-ary multiple fixed-point combinator, and use it in Scheme to define mutually-recursive procedures.

220. **Multiple fixed-point combinators IV (extending Turing's fixed-point combinator)** Can we construct, just as we did for Curry's fixed-point combinator, an extension of Turing's fixed-point combinator for n multiple fixed points? Recall Turing's singular fixed-point combinator (§ 130):

$$\begin{array}{rcl} \mathrm{Y}_{\textsc{Turing}} & = & ((\lambda xf.f(xxf)) \\ & & \ (\lambda xf.f(xxf))) \end{array}$$

Unlike Curry's fixed-point combinator, Turing's fixed-point combinator does not abstract the function f over the application, but rather passes it along:

$$\boxed{((\lambda xf.f(xxf))(\lambda xf.f(xxf)))} \quad = \quad (f\boxed{((\lambda xf.f(xxf))(\lambda xf.f(xxf)))})$$

We expect an extension of Turing's fixed-point combinator to pass along $f_1, \ldots f_n$ with each application. Of course, the applications themselves will become more complex, since they will be juggling n fixed points. A minimal construction that seems to capture these properties could be:

$$\begin{array}{rcl} \mathrm{Y}_{\textsc{Turing}}{}^{n}_{1} & = & ((\lambda x_1x_2\cdots x_nf_1f_2\cdots f_n.f_1(x_1x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_2x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_nx_1x_2\cdots x_nf_1f_2\cdots f_n)) \\ & & (\lambda x_1x_2\cdots x_nf_1f_2\cdots f_n.f_1(x_1x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_2x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_nx_1x_2\cdots x_nf_1f_2\cdots f_n)) \\ & & (\lambda x_1x_2\cdots x_nf_1f_2\cdots f_n.f_2(x_1x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_2x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_nx_1x_2\cdots x_nf_1f_2\cdots f_n)) \\ & & \cdots \\ & & (\lambda x_1x_2\cdots x_nf_1f_2\cdots f_n.f_n(x_1x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_2x_1x_2\cdots x_nf_1f_2\cdots f_n) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ & & \qquad\qquad\qquad\qquad\qquad\qquad (x_nx_1x_2\cdots x_nf_1f_2\cdots f_n))) \end{array}$$

$$
\begin{aligned}
\mathrm{Y}_{\mathrm{TURING}_{2}^{n}} \quad = \quad & ((\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_2 (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_1 (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_2 (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& \cdots \\
& (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_n (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Y}_{\mathrm{TURING}_{j}^{n}} \quad = \quad & ((\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_j (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_1 (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_2 (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& \cdots \\
& (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_n (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& \qquad \cdots \\
& \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n))) \\
\cdots &
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathrm{Y}_{\mathrm{TURING}_n^n} & = & ((\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_n (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad \cdots \\
& & \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& & (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_1 (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad \cdots \\
& & \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& & (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_2 (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad \cdots \\
& & \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)) \\
& & \cdots \\
& & (\lambda x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n . f_n (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_1 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad (x_2 x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n) \\
& & \qquad \cdots \\
& & \qquad (x_n x_1 x_2 \cdots x_n f_1 f_2 \cdots f_n)))
\end{array}
$$

221. **Exercise 68:** Similarly to the exercise in § 218, define an applicative-order version of the n-ary multiple fixed-point combinator defined in § 220, and use it in Scheme to define mutually-recursive procedures.

222. In § 191, we showed that there are infinitely-many fixed-point combinators. It is straightforward to extend that proof to show that for any $n > 0$, there are infinitely-many sets of multiple fixed-point combinators $\{\Phi_1^n, \ldots, \Phi_n^n\}$. To show this, we return to the definition of a set of multiple fixed-point combinators (§ 205). The terms $\Phi_1^n, \ldots, \Phi_n^n$ are multiple fixed-point combinators if for any λ-terms $x_1, \ldots, x_n$, the following holds:

$$
\begin{array}{rcll}
(\Phi_1^n x_1 \cdots x_n) & = & (x_1 (\Phi_1^n x_1 \cdots x_n) & (7) \\
& & \qquad \cdots \\
& & \qquad (\Phi_n^n x_1 \cdots x_n)) & \\
& \cdots & & (8) \\
(\Phi_n^n x_1 \cdots x_n) & = & (x_n (\Phi_1^n x_1 \cdots x_n) & (9) \\
& & \qquad \cdots \\
& & \qquad (\Phi_n^n x_1 \cdots x_n) &
\end{array}
$$

By abstracting $x_1, \ldots, x_n$ over both the left-hand side and the right-hand side of Eq (7) we get:

$$
\begin{array}{rcl}
\Phi_1^n & = & \lambda x_1 \cdots x_n . (x_1 (\Phi_1^n x_1 \cdots x_n) \\
& & \qquad \cdots \\
& & \qquad (\Phi_n^n x_1 \cdots x_n)) \\
\cdots & & \\
\Phi_n^n & = & \lambda x_1 \cdots x_n . (x_n (\Phi_1^n x_1 \cdots x_n) \\
& & \qquad \cdots \\
& & \qquad (\Phi_n^n x_1 \cdots x_n))
\end{array}
$$

Now forget for a moment what $\Phi_1^n, \ldots, \Phi_n^n$ are supposed to "do", and just examine their definition. It should be evident that $\Phi_1^n, \ldots, \Phi_n^n$ are just n

mutually-recursive procedures. Hence, we can re-write them as solutions to a set of n multiple fixed point equations, and solve them using a set of n multiple fixed-point combinators:

$$
\begin{array}{rcl}
\Phi^n_1 & = & \lambda x_1 \cdots x_n.(x_1(\Phi^n_1 x_1 \cdots x_n) \\
 & & \qquad\qquad \cdots \\
 & & \qquad\qquad (\Phi^n_n x_1 \cdots x_n)) \\
 & = & ((\lambda \phi_1 \cdots \phi_n.(x_1(\phi_1 x_1 \cdots x_n) \\
 & & \qquad\qquad \cdots \\
 & & \qquad\qquad (\phi_n x_1 \cdots x_n))) \\
 & & \Phi^n_1 \cdots \Phi^b_n) \\
\cdots & & \\
\Phi^n_n & = & \lambda x_1 \cdots x_n.(x_1(\Phi^n_1 x_1 \cdots x_n) \\
 & & \qquad\qquad \cdots \\
 & & \qquad\qquad (\Phi^n_n x_1 \cdots x_n)) \\
 & = & ((\lambda \phi_1 \cdots \phi_n.(x_n(\phi_1 x_1 \cdots x_n) \\
 & & \qquad\qquad \cdots \\
 & & \qquad\qquad (\phi_n x_1 \cdots x_n))) \\
 & & \Phi^n_1 \cdots \Phi^b_n)
\end{array}
$$

We can now solve this system of multiple fixed-point equations using *any* set of multiple fixed-point combinators of size n. Accordingly, we define

$$
\begin{array}{rcl}
\Psi^n_j & = & \lambda \phi_1 \cdots \phi_n.(x_j(\phi_1 x_1 \cdots x_n) \\
 & & \qquad\qquad \cdots \\
 & & \qquad\qquad (\phi_n x_1 \cdots x_n))
\end{array}
$$

Let Φ^n_j be the j-th multiple fixed-point combinator of size n. We can obtain a different j-th multiple fixed-point combinator of size n by applying it to Ψ^n_j:

$$
\Phi'^n_j \quad = \quad (\Phi^n_j \ \Psi^n_j)
$$

for all $j = 1, \ldots, n$. ■

ADD SOMETHING HERE !!!

- The variadic multiple fixed-point combinator
- non-standard fpcs
- happy birthday combinators

7 Bases (§223–276)

223. Recall how the Boolean functions *conjunction* ($\wedge$), *disjunction* ($\vee$) and *negation* ($\neg$) form a functionally-complete set. This means that all Boolean functions can be expressed by composing conjunctions, disjunctions and negations. A similar situation exists for the set of combinators: We can express all combinators as applications of a terms from a finite set of λ-terms known as a *basis*. In fact, we need not restrict ourselves to the set of combinators; We can define bases for sets of terms with constants, but this is a more esoteric topic and we will treat it later. For the remainder of this section, when we mention *bases* we will assume we mean the set of combinators.

224. For any set S of λ-terms, the set S^+ is defined as the smallest set W that satisfies the following:

$$\begin{array}{rcl} \mathsf{S} & \subseteq & \mathsf{W} \\ (PQ) & \in & \mathsf{W} \quad \text{for all } P, Q \in \mathsf{W} \end{array}$$

The set S^+ is called the set *generated* from S.

225. Another way to think of the set S^+ is that it is the closure of S under the operation of application.

226. A finite set B is set to be *basis* for the set S if for any term $P \in \mathsf{S}$, there exists a term $Q \in \mathsf{B}^+$, such that $P = Q$.

227. Note that when we write "$P = Q$", we mean in the sense of $=_{\beta\eta}$, and consequently, we cannot write something like $\mathsf{S} \subseteq \mathsf{B}^+$, because this would imply syntactic equality, rather than by the equivalence relation induced by the one-step $\beta\eta$-reduction.

228. Note that our definition allows a basis for some set S to generate terms that are not in S.

229. **Example:** The set $\{S^+, c_0\}$, where S^+ is the Church successor, and c_0 is the 0-th Church numeral, generates the set of Church numerals.

230. **Exercise 69:** Find a term generated by the set $\{S^+, c_0\}$ that is not a Church numeral.

231. When considering bases for the set of combinators, we generally drop the qualifier "for the set ..." and just speak of a basis.

232. We are now going to introduce a specific basis for the set of all combinators. We will also give an algorithm for writing any λ-term in terms of this basis. Such an algorithm is known as an *abstraction algorithm*.

233. Suppose we are given a λ-expression M of the form $\lambda x.P$, where $x \notin \mathsf{FreeVars}(P)$. Then

$$\begin{array}{rcl} M & = & \lambda x.P \\ & = & (\boxed{(\lambda px.p)}\ P) \end{array}$$

We name the boxed combinators **K**.

234. Suppose we are given a λ-expression M of the form $\lambda x.P_x Q$, where $x \in \mathsf{FreeVars}(P), x \notin \mathsf{FreeVars}(Q)$. Then

$$\begin{array}{rcl} M & = & \lambda x.P_x Q \\ & = & (\boxed{(\lambda pqx.pxq)}\ (\lambda x.P_x)\ Q) \end{array}$$

We name the boxed combinator **C**.

235. Suppose we are given a λ-expression M of the form $\lambda x.PQ_x$, where $x \notin \mathsf{FreeVars}(P), x \in \mathsf{FreeVars}(Q_x)$. Then

$$\begin{array}{rcl} M & = & \lambda x.PQ_x \\ & = & (\boxed{(\lambda pqx.p(qx))}\ P\ (\lambda x.Q_x)) \end{array}$$

We name the boxed combinator **B**.

236. Suppose we are given a λ-expression M of the form $\lambda x.P_x Q_x$, where $x \in \mathsf{FreeVars}(P_x), x \in \mathsf{FreeVars}(Q_x)$. Then

$$\begin{aligned} M &= \lambda x.P_x Q_x \\ &= (\boxed{(\lambda pqx.px(qx))}\ (\lambda x.P_x)\ (\lambda x.Q_x)) \end{aligned}$$

We name the boxed combinator $\mathbf{S}$.

237. Let the *identity combinator* be given by $\mathbf{I} = \lambda x.x$.

238. We claim that the set $\mathsf{B} = \{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ is a basis for the set Λ^0 of all combinators. Put otherwise, we claim that any combinator can be written using only applications of terms from B; No additional terms or any use of λ-abstraction are needed. Just applications of terms from B.

239. Before we proceed to prove this claim, let us recall that the four combinators $\mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ were introduced as part of a way of writing larger λ-expressions in terms of simpler λ-expressions. This intuition will form the heart of the proof we now give. The $\mathbf{I}$ was introduced differently, as a kind of base case for the following algorithm.

240. Let $\mathsf{B} = \{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$. We now introduce an abstraction algorithm $\mathcal{A}^{\mathsf{B}}[\![\ ]\!]$ on the inductive structure of a combinator. Pick $M \in \Lambda^0$,

- If $M \in \mathsf{B}$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = M$$

  This rule summarizes the base cases.
- If $M = (PQ)$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = (\mathcal{A}^{\mathsf{B}}[\![P]\!]\ \mathcal{A}^{\mathsf{B}}[\![Q]\!])$$

  We call this rule *application-elimination*.
- If $M = \lambda x.\lambda y.P$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = \mathcal{A}^{\mathsf{B}}[\![\lambda x.\mathcal{A}^{\mathsf{B}}[\![\lambda y.P]\!]]\!]$$

  We call this rule *abstraction-elimination*.
- If $M = \lambda x.P$, and $x \notin \mathsf{FreeVars}(P)$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = (\mathbf{K}\ \mathcal{A}^{\mathsf{B}}[\![P]\!])$$

  We call this rule $\mathbf{K}$-*introduction*.
- If $M = \lambda x.P_x Q$, and $x \in \mathsf{FreeVars}(P), x \notin \mathsf{FreeVars}(Q)$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = (\mathbf{C}\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.P_x)]\!]\ \mathcal{A}^{\mathsf{B}}[\![Q]\!])$$

  We call this rule $\mathbf{C}$-*introduction*.
- If $M = \lambda x.PQ_x$, and $x \notin \mathsf{FreeVars}(P), x \in \mathsf{FreeVars}(Q_x)$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = (\mathbf{B}\ \mathcal{A}^{\mathsf{B}}[\![P]\!]\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.Q_x)]\!])$$

  We call this rule $\mathbf{B}$-*introduction*.
- If $M = \lambda x.P_x Q_x$, and $x \in \mathsf{FreeVars}(P_x), x \in \mathsf{FreeVars}(Q_x)$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = (\mathbf{S}\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.P_x)]\!]\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.Q_x)]\!])$$

  We call this rule $\mathbf{S}$-*introduction*.

241. **Example:** Consider the λ-expression $\omega = \lambda x.xx$. The abstraction algorithm proceeds over it as follows:

$$\begin{aligned} \mathcal{A}^{\mathsf{B}}[\![\lambda x.xx]\!] &= (\mathbf{S}\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.x)]\!]\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.x)]\!]) \\ &= (\mathbf{S}\ \mathbf{I}\ \mathbf{I}) \end{aligned}$$

242. **Example:** Let $M = \lambda x.Px$, for some combinator P. Then

$$\begin{aligned} \mathcal{A}^{\mathsf{B}}[\![M]\!] &= (\mathbf{B}\ \mathcal{A}^{\mathsf{B}}[\![P]\!]\ \mathcal{A}^{\mathsf{B}}[\![(\lambda x.x)]\!]) \\ &= (\mathbf{B}\ P\ \mathbf{I}) \end{aligned}$$

But of course, $M \longrightarrow_\eta P$. Our abstraction algorithm is "unaware" of η-reduction, and introduces correct, but redundant combinators. We can fix this by introducing one additional rule which will be considered *before* the rule **B**-*introduction*:

- If $M = \lambda x.Px$, and $x \notin \mathsf{FreeVars}(P)$, then

$$\mathcal{A}^{\mathsf{B}}[\![M]\!] = \mathcal{A}^{\mathsf{B}}[\![P]\!]$$

  We call this rule η-*elimination*.

243. **Example:** Let us apply the abstraction algorithm to S^+, the successor on Church numerals:

$$\begin{aligned} \mathcal{A}^{\mathsf{B}}[\![S^+]\!] &\equiv \mathcal{A}^{\mathsf{B}}[\![\lambda abc.b(abc)]\!] \\ &= \mathcal{A}^{\mathsf{B}}[\![\lambda a.\mathcal{A}^{\mathsf{B}}[\![\lambda b.\mathcal{A}^{\mathsf{B}}[\![\lambda c.b(abc)]\!]]\!]]\!] \\ &= \mathcal{A}^{\mathsf{B}}[\![\lambda a.\mathcal{A}^{\mathsf{B}}[\![\lambda b.(\mathbf{B}\ \mathcal{A}^{\mathsf{B}}[\![b]\!]\ \mathcal{A}^{\mathsf{B}}[\![\lambda c.abc]\!])]\!]]\!] \\ &=_\eta \mathcal{A}^{\mathsf{B}}[\![\lambda a.\mathcal{A}^{\mathsf{B}}[\![\lambda b.(\mathbf{B}\ b\ (ab))]\!]]\!] \\ &= \mathcal{A}^{\mathsf{B}}[\![\lambda a.(\mathbf{S}\ \mathcal{A}^{\mathsf{B}}[\![\lambda b.\mathbf{B}b]\!]\ \mathcal{A}^{\mathsf{B}}[\![\lambda b.ab]\!])]\!] \\ &=_\eta \mathcal{A}^{\mathsf{B}}[\![\lambda a.(\mathbf{S}\ \mathbf{B}\ a)]\!] \\ &=_\eta (\mathbf{S}\ \mathbf{B}) \end{aligned}$$

244. The definitions for each of the combinators in the basis B, as well as their original name in German, are given below:

$\mathbf{I}$	$=$	$\lambda x.x$	*Identitätsfunktion*
$\mathbf{K}$	$=$	$\lambda xy.x$	*Konstanzfunktion*
$\mathbf{B}$	$=$	$\lambda xyz.x(yz)$	*Zusammensetzungsfunktion*
$\mathbf{C}$	$=$	$\lambda xyz.xzy$	*Vertauschungsfunktion*
$\mathbf{S}$	$=$	$\lambda xyz.xz(yz)$	*Verschmelzungsfunktion*

245. **Exercise 70:** Apply the above abstraction algorithm to $\mathrm{Y}_{\text{Curry}}$.

246. **Exercise 71:** Apply the above abstraction algorithm to $\lambda abcd.a$.

247. **Exercise 72:** Generalize the solution to § 246 to $\lambda x_1 x_2 \cdots x_n.x_1$.

248. **Exercise 73:** Generalize the solution to § 247 to $\lambda x_1 x_2 \cdots x_n.x_k$, where $k \in \{1, \ldots, n\}$.

249. **Exercise 74:** Write a computer program, either in Scheme or in ML, to compile λ-expressions into $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$.

250. The terms $\mathbf{I}, \mathbf{B}, \mathbf{C}$ can be expressed in terms of the remaining $\mathbf{K}, \mathbf{S}$:

$$\begin{aligned} \mathbf{I}x &= x \\ &= \mathbf{K}x(\mathbf{K}x) \\ &= \mathbf{SKK}x \end{aligned}$$

So by extensionality,[12] we have $\mathbf{I} = (\mathbf{SKK})$.

In fact, there is something overly specific in the above derivation. We could have written more generally that:

$$\begin{aligned} \mathbf{I}x &= x \\ &= \mathbf{K}x(Mx) \\ &= \mathbf{SK}Mx \end{aligned}$$

for any term M that does not contain x as a free variable. So a stronger claim is that $\mathbf{I} = (\mathbf{SK}\langle anything \rangle)$.

$$\begin{aligned} \mathbf{B}xyz &= x(yz) \\ &= \mathbf{K}xz(yz) \\ &= \mathbf{S}(\mathbf{K}x)yz \\ &= \mathbf{KS}x(\mathbf{K}x)yz \\ &= \mathbf{S}(\mathbf{KS})\mathbf{K}xyz \end{aligned}$$

So by extensionality, we have $\mathbf{B} = (\mathbf{S}(\mathbf{KS})\mathbf{K})$.

$$\begin{aligned} \mathbf{C}xyz &= xzy \\ &= xz(\mathbf{K}yz) \\ &= \mathbf{S}x(\mathbf{K}y)z \\ &= \mathbf{K}(\mathbf{S}x)y(\mathbf{K}y)z \\ &= \mathbf{S}(\mathbf{K}(\mathbf{S}x))\mathbf{K}yz \\ &= \mathbf{S}(\mathbf{KK}x(\mathbf{S}x))\mathbf{K}yz \\ &= \mathbf{S}(\mathbf{S}(\mathbf{KK})\mathbf{S}x)\mathbf{K}yz \\ &= \mathbf{KS}x(\mathbf{S}(\mathbf{KK})\mathbf{S}x)\mathbf{K}yz \\ &= \mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S})x\mathbf{K}yz \\ &= \mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S})x(\mathbf{KK}x)yz \\ &= \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}))(\mathbf{KK})xyz \end{aligned}$$

So by extensionality, we have $\mathbf{C} = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}))(\mathbf{KK})$.

251. **Exercise 75:** Express $\lambda x.xx$ using $\{\mathbf{K}, \mathbf{S}\}$. Try to follow the approach in § 250, rather than first converting to $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ and then substituting $\mathbf{I}, \mathbf{B}, \mathbf{C}$ with their equivalents in $\{\mathbf{K}, \mathbf{S}\}$.

252. **Exercise 76:** Express $\lambda xy.xyy$ using $\{\mathbf{K}, \mathbf{S}\}$. As before, try to follow the approach in § 250.

[12] The principle of extensionality on sets states that two sets are equal if they have the same members. The corresponding principle of extensionality for functions states that two functions are equal if they have the share the same domain, the same range, and map the same elements to the same elements: $(\forall x)(f(x) = g(x)) \Leftrightarrow (f = g)$. The corresponding principle of extensionality for λ-terms states that two λ-terms are equal if their applications to the same terms are also equal: $(\forall x)((Mx = Nx) \Leftrightarrow (M = N))$. Philosophically, the principle of extensionality is a restricted expression of the Leibnizian principle of the *identity of the indiscernables*.

253. **Exercise 77:** Write a computer program, either in Scheme or in ML, to compile λ-expressions into $\{\mathbf{K}, \mathbf{S}\}$. Try *not* to use the result of § 249, but rather observe how variables were "extracted" or "factored out" in § 250, and try to formalize this observation as an algorithm. The expressions you generate will be much shorter if you take this route.

254. **Exercise 78:** Show that neither $\{\mathbf{K}\}$ nor $\{\mathbf{S}\}$ are bases for the $\lambda\mathbf{K}\beta\eta$-calculus.

255. Let X be defined as follows:

$$X = \langle \mathbf{S}, \mathbf{K} \rangle$$

We wish to show that $\mathbf{K}, \mathbf{S} \in \{X\}^+$.

$$\begin{aligned}
(XX) &= X\mathbf{SK} \\
&= \mathbf{SSKK} \\
&= \mathbf{SK}(\mathbf{KK}) \\
(X^2X) &= X(\mathbf{SK}(\mathbf{KK})) \\
&= \mathbf{SK}(\mathbf{KK})\mathbf{SK} \\
&= \mathbf{KS}(\mathbf{KKS})\mathbf{K} \\
&= \mathbf{SK} \\
(X^3X) &= X(\mathbf{SK}) \\
&= \mathbf{SKSK} \\
&= \mathbf{KK}(\mathbf{SK}) \\
&= \mathbf{K} \\
(X^4X) &= X\mathbf{K} \\
&= \mathbf{KSK} \\
&= \mathbf{S}
\end{aligned}$$

So $X^3X \twoheadrightarrow K, X^4X \twoheadrightarrow S$. ■

This shows that $\{X\}$ is a basis for the set of all combinators. A basis that consists of just one term is called a *one-point basis*.

256. **Exercise 79:** Show that:

$$\begin{aligned}
(X^{5n+3}\ X) &= \mathbf{K} \\
(X^{5n+4}\ X) &= \mathbf{S}
\end{aligned}$$

Use this fact to show there are infinitely-many abstraction algorithms for $\{X\}$.

257. **Exercise 80:** Show that any bases for Λ^0 has infinitely-many different abstraction algorithms.

258. Throughout this document, we have made ample use of the meta-linguistic ellipsis ('$\cdots$') to describe the *syntax* of λ-expressions inductively. But the syntax of an expression is nothing more than a string of characters, and ellipsis are a meta-linguistic device, and not a part of the actual syntax of λ-expressions. Knowing how to write a λ-expression inductively doesn't give us a clue as to how to define it in the λ-calculus, i.e., without the ellipsis.

A basis provides a set of "building blocks" from which complex λ-expressions can be constructed out of simpler ones. By converting λ-expressions to a given basis (using an *abstraction algorithm*), we can often expose their inductive *structure*, as opposed to their inductive *syntax*. Once the inductive structure is understood, it is straightforward to define the term without any ellipsis.

259. **Case Study:** *Using basis to construct inductively-defined expressions.* Following §53–56, the ordered n-tuple maker is given by the following expression:

$$\lambda x_1 x_2 \cdots x_n z.z x_1 x_2 \cdots x_n$$

How might we construct a λ-expression that takes the Church numeral c_n and returns the corresponding n-tuple maker? If you try to tackle this problem directly, you may find it is rather tricky.

Sometimes converting an expression to $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ can give us a lot of insight as to the structure of our term. For example, if you did the exercise in § 249, and use it to compile n-tuple makers for various values of n, it will become immediately clear that

$$\begin{aligned} \lambda x_1 \cdots x_n. \langle x_1, \ldots, x_n \rangle &\equiv \lambda x_1 x_2 \cdots x_n z.z x_1 x_2 \cdots x_n \\ &= \underbrace{(\mathbf{B}^{n-1}\mathbf{C}(\mathbf{B}^{n-2}\mathbf{C} \cdots (\mathbf{B}\mathbf{C}(\mathbf{C}}_{n-1}\mathbf{I})) \cdots)) \end{aligned}$$

Once you realize that for any λ-expression M, $M^0 = \mathbf{I}$, it should be clear that $(\mathbf{B}^0\mathbf{C}\mathbf{I}) = (\mathbf{I}\mathbf{C}\mathbf{I}) = (\mathbf{C}\mathbf{I})$, and the pattern should be recognizable.

Once you recognize the pattern, *you should prove it by induction.*

Keeping in mind that $c_n\mathbf{B} = \mathbf{B}^n$, consider the function f that maps ordered pairs to ordered pairs, as follows:

$$\langle c_n, M \rangle \quad \stackrel{f}{\Longrightarrow} \quad \langle c_{n+1}, (c_n \mathbf{B}\mathbf{C}M) \rangle$$

We can define f as follows:

$$f \quad \equiv \quad \lambda p. \left\langle S^+(\pi_1^2 p), \pi_1^2 p \mathbf{B}\mathbf{C}(\pi_2^2 p) \right\rangle$$

We can now define the n-tuple maker by taking the n-th composition of f and applying it to $\mathbf{I}$. A singularly appropriate name for this term seems to be *malloc*, after the C library function:

$$\mathbf{Malloc} \quad \equiv \quad \lambda n. \pi_2^2 (n f \langle c_0, \mathbf{I} \rangle)$$

260. **Exercise 81:** Define the combinator FoldL, such that

$$\text{FoldL } f\ z\ c_n \quad = \quad \lambda x_1 \cdots x_n. (f \cdots\ (f\ (f\ z\ x_1)\ x_2) \cdots x_n)$$

261. **Exercise 82:** Define the combinator FoldR, such that

$$\text{FoldR } f\ z\ c_n \quad = \quad \lambda x_1 \cdots x_n. (f\ x_1\ (f\ x_2 \cdots\ (f\ x_n\ z) \cdots))$$

262. **Exercise 83:** Write the procedure $=n?$, which takes a Church numeral c_n and returns the predicate that takes a Church numeral c_k as an argument, and tests whether $k = n$. A simple way to do this would be to use the equality predicate on Church numerals:

$$=n? \quad = \quad (=\ c_n)$$

Instead, your solution should generalize over the definition of $=3?$ in § 73.

263. **Exercise 84:** In § 84 we saw how to define the λ-terms E_D, F_D that compute unary (Eq (4)) and binary functions (Eq (5)), and binary function F_D, on D-numerals, from the λ-terms $E_{\text{Church}}, F_{\text{Church}}$ that compute the corresponding functions on Church numerals.

In this problem, we would like to generalize over Eq (4), Eq (5), for the n-variable case. Define a λ-term **Grab** that takes 4 arguments

- A λ-term c, that maps from D-numerals to Church numerals
- A λ-term c', that maps from Church numerals to D-numerals
- A λ-term f, that computes some function over n arguments
- A λ-term n, the value of which is c_n

and returns a λ-expression that takes n arguments $x_1, \ldots, x_n$, and returns

$$c'(f(c\ x_1)\cdots(c\ x_n))$$

Colloquially, as a gross abuse of notation, we could write:

$$\textbf{Grab} \quad = \quad \lambda cc'fc_nx_1\cdots x_n.c'(f(c\ x_1)\cdots(c\ x_n))$$

Write a proper definition for **Grab**.

264. There exist infinitely-many different bases for the $\lambda\mathbf{K}\beta\eta$-calculus. Each basis requires a different abstraction algorithm in order to express λ-terms using the given basis. We will not explore various bases that have interesting and curious properties.

265. It is straightforward to express a stack in the λ-calculus using nested ordered-pairs, in much the same way as stacks are represented as nested ordered-pairs in LISP or Scheme.

We define the terms $\uparrow, \alpha, \downarrow$, denoting "push", "apply", and "pop", respectively, as follows:

$$\begin{aligned}
\uparrow &\equiv \lambda sam.m[a,s] \\
\alpha &\equiv \lambda sm.m[\pi_1^2 s(\pi_1^2(\pi_2^2 s)), \pi_2^2(\pi_2^2 s)] \\
\downarrow &\equiv \lambda s.\pi_1^2 s \ =_\eta \ \pi_1^2
\end{aligned}$$

We refer to these terms as "instructions" in some abstract stack architecture.

- The "push instruction" $\uparrow$ takes a stack s, some term a, and a "an instruction" m (remember that m is one of $\uparrow, \alpha, \downarrow$). We want to "push" a onto the stack s, and pass this new stack to the "instruction" m. We therefore apply m to the pair of a, s.
- The "apply instruction" α takes a stack s, and an "instruction" m. It applies the first term on the stack to the second term on the stack, and pushes the result onto the remaining stack. This new stack is passed on to the "instruction". We therefore apply m to the stack containing the application of the first and second elements of the original stack, and the rest of the stack (shortened by two elements).
- The "pop instruction" $\downarrow$ takes a stack s and returns its top element. It is therefore η-equivalent to first projection of an ordered pair.

It does not matter what we use for an initial stack, so we arbitrarily pick the identity combinator.

We could define an "empty stack instruction" $\Sigma_{\mathbf{I}}$ as the term that takes an "instruction" m and applies it to $\mathbf{I}$:

$$\Sigma_{\mathbf{I}} \quad \equiv \quad \lambda m.m\mathbf{I}$$

But we could just as easily pass on $\mathbf{I}$ to the first "instruction".

266. *Flat terms.* Terms that do not require parenthesis are said to be flat. Such terms have no right-associated applications.

267. **Example:** The following terms are flat: $\lambda x.x$, $\lambda xyz.xzy$, $\lambda x.xxx$.

268. **Example:** The following terms are not flat: $\lambda x.x(xx)$, $\lambda xy.x(x(xy))$.

269. Using the terms $\uparrow, \alpha, \downarrow$, it is possible to flatten any term, i.e., to rewrite it so that it does not contain any right-associated applications.

270. **Example:** The term $A(BC)$ can be written as $\Sigma_{\mathbf{I}} \uparrow C \uparrow B\alpha \uparrow A\alpha \downarrow$.

We can "trace" the reduction of the flattened term using the Σ-notation:

$$\begin{aligned}
\Sigma_{\mathbf{I}} \uparrow C \uparrow B\alpha \uparrow A\alpha \downarrow &= \Sigma_{[C,\mathbf{I}]} \uparrow B\alpha \uparrow A\alpha \downarrow \\
&= \Sigma_{[B,[C,\mathbf{I}]]}\alpha \uparrow A\alpha \downarrow \\
&= \Sigma_{[(BC),\mathbf{I}]} \uparrow A\alpha \downarrow \\
&= \Sigma_{[A,[(BC),\mathbf{I}]]}\alpha \downarrow \\
&= \Sigma_{[A(BC),\mathbf{I}]} \downarrow \\
&= A(BC)
\end{aligned}$$

271. **Example:** If we are not to use $\Sigma_{\mathbf{I}}$, we can still rewrite $A(BC)$ as $\uparrow \mathbf{I}C \uparrow B\alpha \uparrow A\alpha \downarrow$.

272. **Exercise 85:** Flatten the expression $A(BC)(D(EF)G)$ using the "instructions" $\uparrow, \alpha, \downarrow$.

273. **Exercise 86:** Write a computer program to take a combinator and translate it to a flat expression in terms of $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}, \uparrow, \downarrow, \alpha\}$.

274. **Exercise 87:** When flattening expressions using $\uparrow, \downarrow, \alpha$, the terms are pushed onto the stack from left to right. Define the corresponding terms $\uparrow_{\mathrm{RL}}, \downarrow_{\mathrm{RL}}, \alpha_{\mathrm{RL}}$ that will also flatten expression, but with the arguments pushed from right to left. For example, the expression $(A(BC)D)$ could be flattened as follows:

$$(A(BC)D) \quad = \quad \Sigma_{\mathbf{I}} \uparrow_{\mathrm{RL}} D \ \uparrow_{\mathrm{RL}} C \ \uparrow_{\mathrm{RL}} B \ \alpha_{\mathrm{RL}} \uparrow_{\mathrm{RL}} A \ \alpha_{\mathrm{RL}} \alpha_{\mathrm{RL}} \downarrow_{\mathrm{RL}}$$

ADD SOMETHING HERE !!!

275. **Exercise 88:** Give a closed formula for $T(n, k)$, the number of expressions that can be written using n terms chosen from a basis of k λ-terms (with possible repetition of terms from the basis).

276. **Exercise 89:** Regarding $T(n, k)$ in § 275, show that

$$T(n+1, k) \quad = \quad \frac{2k(2n-1)}{n+1} T(n, k)$$

ADD SOMETHING HERE !!!

- an abstract combinator machine
- flat bases and encoding combinators
- a basis for the $\lambda\mathbf{I}\beta\eta$-calculus
- Church's original basis for the $\lambda\mathbf{I}\beta\eta$-calculus
- Variable-arity bases

Acknowledgements

This document has been iterated over these past four years and is still far from reaching a fixed point. I am adding new material, new chapters, and many exercises all the time. Someone following the *waterfall* model for software development might say that this document is at a stage too early for editorial work, since each change and addition introduces new errors, typos and problems.

This is why I am evermore grateful to those kind souls who had gone over this very rough document, and sent me their corrections, comments and suggestions. I happy to acknowledge the helpful and insightful comments of *Kristoffer Andersen*, *Michael Bar-Sinai*, *Omer Basha*, *Dan Carmon*, *Yuri Chernyavsky*, *Christian Clausen*, *Aviad Cohen*, *Olivier Danvy*, *Joël Duet*, *Michael Frank*, *Arie Gartenlaub*, *Kent Grigo*, *Nathan Grunzweig*, *Tomer Heber*, *John Holland*, *Vitaly Khait*, *Igal Khitron*, *Roman Smelyansky*, *Guy Wiener*, and the APL classes of 2007–2013.