A.8. Use the λ-calculus axioms and rules of inference to prove that

\[ \lambda n. \lambda f. \lambda x. (f ((n f) x)) = \lambda f. \lambda x. (f (f (f x))) \]

(or, in other words, that the successor of two is three!).

A.9. By hand, evaluate the λ-calculus expression

\[ (((\lambda x. \lambda y. \lambda z. ((z \lambda w. y) x) u) v) \lambda x. \lambda y. y) \]

by performing as many successive beta-reductions on it as possible. You may choose any evaluation order.

A.10. In an abnormal-order evaluator for the λ-calculus, we choose always to reduce the rightmost maximal beta-redex in the current expression, and we keep reducing until no more beta-redexes remain. Implement an abnormal-order evaluator for LAMBDA. Give an example of a LAMBDA expression that has a normal form, but runs forever in the abnormal-order evaluator.

A.11. Give an example of a LAMBDA expression that terminates with a normal form in a normal-order evaluator, terminates but does not yield a normal form in an applicative-order evaluator without partial evaluation, and does not terminate in an applicative-order evaluator with partial evaluation.

A.12. Modify the implementation of the run procedure in the normal-form evaluator for LAMBDA so that it not only returns the normal form of each expression that it is given, but also displays, as a side effect, the number of beta-reductions that the evaluator performed in reaching that normal form.

A.13. An arithmetic predicate in the λ-calculus is a function that returns either true or false when given any Church numeral (in other words, the application of the function to the Church numeral is always beta-reducible to true or to false).

Define and test a function minarg that takes any arithmetic predicate \( f \) and returns the Church numeral for the least natural number \( n \) that satisfies \( f \) (that is, the least one for which the application is beta-reducible to true). It is all right, and indeed desirable, for there to be no normal form when minarg is applied to an arithmetic predicate that returns false for every Church numeral.

A.14. In the λ-calculus, define and test a “binary” function equal? that tests whether two given Church numerals represent the same number. In other words, ((equal? m) n) should be beta-reducible to true when the same Church numeral is substituted for both \( m \) and \( n \), but to false when different Church numerals are dropped in.

A.15. In the λ-calculus, define and test a “binary” function append that constructs and returns the result of concatenating two given lists.

A.16. Implement Wadler’s non-deterministic choice monad as an extension of the CALL-BY-NEED language.

A.17. How many values can the Icon expression not (0 to 5) generate, and what are those values? Why do Griswold and Griswold describe not as a “control structure” (p. 23) rather than an operator?
A.18. Add a not-expression to the GENERATORS language, with the syntax

\[
\text{not <expression>}
\]

When evaluated, a not-expression should fail if its subexpression succeeds and should succeed (once), producing the value 0, if its subexpression fails.

A.19. Read chapter 7 of *The Icon programming language*, then add to GENERATORS (a) a list data type, as in exercise 3.9 of Friedman and Wand, and (b) a unary element operator, with the syntax

\[
\text{element ( <expression> )}
\]

that generates the elements of a given list. (For instance, the expression

\[
\text{every print(element(ls)) do fail}
\]

should print out the elements of ls.)