5.2.2 EXERCISE [⋆⋆]: Find another way to define the successor function on Church numerals. □

$c_0 = \lambda s.\ \lambda z.\ z \qquad c_1 = \lambda s.\ \lambda z.\ s\ z \qquad \ldots$

$succ = \lambda n.\ \lambda s.\ \lambda z.\ n\ s\ (s\ z)$

$n\ s \rightarrow_0 \lambda z.\ ss \rightarrow s\ z$

5.2.3 EXERCISE [⋆⋆]: Is it possible to define multiplication on Church numerals without using `plus`? □

$times = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ (n\ s)\ z$

5.2.4 EXERCISE [RECOMMENDED, ⋆⋆]: Define a term for raising one number to the power of another. □

$Power = \lambda m.\ \lambda n.\ m\ (times\ n)\ c_1$

5.2.6 EXERCISE [⋆⋆]: Approximately how many steps of evaluation (as a function of n) are required to calculate `prd` c_n? □

```
zz = pair c₀ c₀;
ss = λp. pair (snd p) (plus c₁ (snd p));
prd = λm. fst (m ss zz);
```

Because it builds from $(Pair\ c_0\ c_0)$ to $(pair\ c_{n-1}\ c_n) \rightarrow O(n)$

5.2.7 EXERCISE [⋆⋆]: Write a function `equal` that tests two numbers for equality and returns a Church boolean. For example,

```
equal c3 c3;
▸ (λt. λf. t)
equal c3 c2;
▸ (λt. λf. f)
```

□

tru

fls

equal = λm. λn. and (iszero (m prd n)) (iszero (n prd n))

5.2.8 EXERCISE [RECOMMENDED, ⋆⋆⋆]: A list can be represented in the lambda-calculus by its `fold` function. (OCaml's name for this function is `fold_left`; it is also sometimes called `reduce` .) For example, the list `[x,y,z]` becomes a function that takes two arguments `c` and `n` and returns `c x (c y (c z n))`. What would the representation of `nil` be? Write a function `cons` that takes an element `h` and a list (that is, a fold function) `t` and returns a similar representation of the list formed by prepending `h` to `t`. Write `isnil` and `head` functions, each taking a list parameter. Finally, write a `tail` function for this representation of lists (this is quite a bit harder and requires a trick analogous to the one used to define `prd` for numbers). □

nil = λx. true list = λs. s h t

cons = λh. λt. λs. s h t

isnil = λl. l (λh. λt. false)

head = λl. l true tail = λl. l false

5.2.11 EXERCISE [RECOMMENDED, ⋆⋆]: Use `fix` and the encoding of lists from Exercise 5.2.8 to write a function that sums lists of Church numerals. □

sum = λL. isnil (L) (C0) (Plus (head L) (sum (tail L)))

F = λf. λL. isnil (L) (C0) (Plus (head L) (f (tail L)))

sum = fix F

4.3 Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda x.\lambda y.xy)(\lambda x.xy)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

$(\lambda x.\lambda y.xy)(\lambda x.xy) = (\lambda v.\lambda w.vw)(\lambda x.xy)$

$= \lambda w.(\lambda x.xy)w = \lambda w.wy$

If we don't rename the variables, we get in trouble distinguishing free and bound variables.

4.4 Symbolic Evaluation

The Algol-like program fragment

```
function f(x)
    return x+4
end;
function g(y)
    return 3-y
end;
f(g(1));
```

can be written as the following lambda expression:

$$\left(\underbrace{(\lambda f.\lambda g.\, f(g\ 1))}_{\text{main}}\ \underbrace{(\lambda x.x+4)}_{f} \right)\ \underbrace{(\lambda y.3-y)}_{g}.$$

Reduce the expression to a normal form in two different ways, as described below.

(a) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *left* as possible.

(b) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *right* as possible.

a) $\left((\lambda f.\ \lambda g.\ f(g\ 1))(\lambda x.\ x+4)\right)(\lambda y.\ 3-y)$

$= \left(\lambda g.\ (\lambda x.\ x+4)(g\ 1)\right)(\lambda y.\ 3-y)$

$= (\lambda x.\ x+4)((\lambda y.\ 3-y)\ 1) = ((\lambda y.\ 3-y)\ 1) + 4$

$= (3-1) + 4 = 6$

b) $\left((\lambda f.\ \lambda g.\ f(g\ 1))(\lambda x.\ x+4)\right)(\lambda y.\ 3-y)$

$= \left(\lambda g.\ (\lambda x.\ x+4)(g\ 1)\right)(\lambda y.\ 3-y)$

$= \left(\lambda g.\ ((g\ 1)+4)\right)(\lambda y.\ 3-y) = ((\lambda y.\ 3-y)\ 1) + 4$

$= (3-1) + 4 = 6$

4.5 Lambda Reduction with Sugar

Here is a "sugared" lambda expression that uses let declarations:

$$\begin{array}{l}\text{let } compose = \lambda f.\, \lambda g.\, \lambda x.\ f(g\,x) \text{ in}\\ \quad \text{let } h = \lambda x.\, x + x \text{ in}\\ \quad\quad compose\, h\, h\, 3\end{array}$$

The "desugared" lambda expression, obtained when each let $z = U$ in V is replaced with $(\lambda z.\, V)\, U$ is

$$\begin{array}{l}(\lambda compose.\\ \quad (\lambda h.\ compose\ h\ h\ 3)\ \ \lambda x.\, x + x)\\ \quad \lambda f.\, \lambda g.\, \lambda x.\, f(g\,x).\end{array}$$

This is written with the same variable names as those of the let form to make it easier to read the expression.

Simplify the desugared lambda expression by using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

$$(\lambda \text{compose}.\ (\lambda h.\ \text{compose}\ h\ h\ 3)\ \lambda x.\ x+x)\ \lambda f.\ \lambda g.\ \lambda a.\ f(g\,a)$$

$$= (\lambda h.\ (\lambda f.\ \lambda g.\ \lambda a.\ f(g\,(a)))\ h\ h\ 3)\ \lambda x.\ x+x$$

$$= (\lambda f.\ \lambda g.\ \lambda a.\ f(g\,(a)))(\lambda x.\ x+x)\ (\lambda x.\ x+x)\ 3$$

$$= (\lambda g.\ \lambda a.\ (\lambda x.\ x+x)\ (g\,(a)))\ (\lambda x.\ x+x)\ 3$$

$$= (\lambda a.\ (\lambda x.\ x+x)((\lambda x.\ x+x)\ (a)))\ 3$$

$$= (\lambda x.\ x+x)((\lambda x.\ x+x)\ 3) = (\lambda x.\ x+x)\ (6) = 12$$

4.6 Translation into Lambda Calculus

A programmer is having difficulty debugging the following C program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, as extra space is required every time a function call is made.)

```
int f(int (*g)(...)){ /* g points to a function that returns an int */
    return g(g);
}
int main(){
  int x;
  x = f(f);
  printf("Value of x = %d\n", x);
  return 0;
}
```

Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application f(f). This program assumes that the type checker does not check the types of arguments to functions.

$f = \lambda g.\ g\ g$

$f\ f = (\lambda g.\ g\ g)(\lambda g.\ g\ g) = (\lambda g.\ g\ g)(\lambda g.\ g\ g) = f\ f$

This is omega which diverges and can not be reduced to a normal form and loops forever.

Exercise:

sum = $\lambda f.\ \lambda m.\ \lambda n.$ test iszero (n) m succ (f m pred n)

Z = $\lambda f.\ (\lambda x.\ f(\lambda y.\ x\ x\ y))(\lambda x.\ f(\lambda y.\ x\ x\ y))$

Plus = Z sum

Plus 1 1 = Z sum 1 1 =

$(\lambda f.\ (\lambda x.\ f(\lambda y.\ x\ x\ y))(\lambda x.\ f(\lambda y.\ x\ x\ y)))$ sum 1 1 =

$(\lambda x.\ sum(\lambda y.\ x\ x\ y))(\lambda x.\ sum(\lambda y.\ x\ x\ y))$ 1 1 =

(iszero 1) 1 scc (z sum 1 (prd 1)) =

scc (z sum 1 (prd 1)) =

scc (z sum 1 0) =

scc $((\lambda f.\ (\lambda x.\ f(\lambda y.\ x\ x\ y))(\lambda x.\ f(\lambda y.\ x\ x\ y)))$ sum 1 0) =

scc $((\lambda x.\ sum\ (\lambda y.\ x\ x\ y))(\lambda x.\ sum\ (\lambda y.\ x\ x\ y))$ 1 0) =

scc ((iszero 0) 1 scc (z sum 1 (prd 0))) = scc 1 = 2