4) Show that:

```
λx.(succ (pred x))
```

and:

```
λx.(pred (succ x))
```

are equivalent for arbitrary non-zero integer arguments. Explain why they are not equivalent for a zero argument.

4. Recursion and arithmetic

4.1. Introduction

In this chapter we are going to look at how recursion is used for repetition in functional programming.

To begin with, we will see that our existing definition notation cannot be used to introduce recursion because it leads to infinite substitution sequences. We will then see that this can be overcome through absraction in individual functions.

Next, we will discuss the introduction of recursion using a general purpose construct based on function self application.

Finally, we will look at the use of recursion to build a wide variety of arithmetic operations.

4.2. Repetition, iteration and recursion

Repetition involves doing the same thing zero or more times. It is useful to distinguish **bounded repetition**, where something is carried out a fixed number of times, from the more general **unbounded iteration**, where something is carried out until some condition is met. Thus, for bounded repetition the number of repetitions is known in advance whereas for unbounded repetition it is not.

It is important to relate the form of repetition to the structure of the item to be processed. Bounded repetition is used where a **linear** sequence of objects of known length is to be processed, for example to process each element of an array. Here, the object sequence can be related to a consecutive range of values. For example, arrays have addresses which are linear sequences of integers.

Unbounded repetition is used where a **nested** sequence of objects is to be processed and the number of layers of nesting is unkown. For example, a filing system might consist of a nested hierarchy of directories and files. Processing such a filing system involves starting at the root directory and then processing the files and sub-directories. Processing the sub-directories involves processing their files and sub-directories, and so on. In general, the depth of directory nesting is unknown. For unbounded repetition, processing ends when the end of the nesting is reached. For example, processing a filing system ends when all the files at every level of directory nesting have been processed.

Bounded repetition is a weaker form of unbounded repetition. Carrying out something a fixed number of times is the same as carrying it out until the last item in a sequence has been dealt with.

In imperative languages repetition is based primarily on **iterative** constructs for repeatedly carrying out structured assignment sequences. For example, in Pascal, `FOR` statements provide bounded iteration over a range of integers and `WHILE` or `REPEAT` statements provide unbounded iteration until a condition is met. Here, repetition involves repeatedly inspecting and changing variables in common memory.

In functional languages, programs are based on structured nested function calls. Repetition requires such nesting to be continued until some condition is met. There is no concept of a changing shared memory. Instead, each function passes its result to the next in the function call nesting. Repetition involves deciding whether or not to carry out another layer of function call nesting with the same nested sequence of function calls.

RBpetition in functional programming is based on *recursion:* the definition of something in terms of itself. The nested function call sequence which is to be repeated is given a name. If some condition is met within the sequence then that name is invoked to repeat the nested function call sequence within itself. The condition often checks whether or not the end of a linear or nested object sequence has been reached.

Let us compare iteration and recursion through another contrived example. Suppose we want to eat some sweets. If we know that there are `N` sweets then we might write an iterative algorithm as:

```
EAT N = FOR COUNT := N DOWNTO 1 DO
         gobble a sweet
```

or:

```
EAT N = COUNT := N
        WHILE COUNT > 0 DO
        BEGIN
             gobble a sweet
             COUNT := COUNT - 1
        END
```

For example, for 3 sweets we would:

```
EAT 3 sweets =>

gobble a sweet and
gobble a sweet and
gobble a sweet and
stop
```

An equivalent recursive algorithm would be:

```
EAT N = IF N > 0 THEN
        BEGIN
             gobble a sweet
             EAT N-1
        END
```

For example, for 3 sweets we would:

```
EAT 3 sweets =>

gobble a sweet and
EAT 2 sweets =>

gobble a sweet and
gobble a sweet and
EAT 1 sweet =>

gobble a sweet and
gobble a sweet and
gobble a sweet and
EAT 0 sweets =>
```

```
gobble a sweet and
gobble a sweet and
gobble a sweet and
stop
```

Note that eating sweets iteratively involves gobbling `1` sweet `N` times whereas eating sweets recursively involves gobbling `1` sweet and then eating the remaining `N-1` sweets recursively.

It is useful to distinguish **primitive recursion** where the number of repetitions is known from **general recursion** here the number of repetitions is unknown. Primitive recursion is weaker than general recursion. Primitive recursion involves a finite depth of function call nesting so it is equivalent to bounded repetition through iteration with a finite memory. For general recursion the nesting depth is unknown so it is equivalent to unbounded repetition through iteration with an infinite memory.

Note that imperative languages often provide repetition through recursive procedures and functions as well as through iteration.

4.3. Recursion through definitions?

It might appear that our definition notation enables recursion and we could just use the name from the left of the definition in the expression on the right. For example, two numbers may be added together by repeatedly incrementing the first and decrementing the second until the second is `zero:`

```
def add x y =
 if iszero y
 then x
 else add (succ x) (pred y)
```

Thus, to add one and two, for example:

```
add one two => ... =>

add (succ one) (pred two) => ... =>

add (succ (succ one)) (pred (pred two)) => ... =>

(succ (succ one)) ==

three
```

However, in chapter 2 we required all names in expressions to be replaced by their definitions before the expression is evaluated.

In the above example:

```
λx.λy.
 if iszero y
 then x
 else add (succ x) (pred y) ==

λx.λy.
 if iszero y
 then x
 else
  ((λx.λy.
    if iszero y
    then x
```

```
        else add (succ x) (pred y))
       (succ x) (pred y)) ==

    λx.λy.
     if iszero y
     then x
     else
      ((λx.λy.
        if iszero y
        then x
        else
         ((λx.λy.
           if iszero y
           then x
           else add (succ x) (pred y))
          (succ x) (pred y))
       (succ x) (pred y)) == ...
```

Replacement will never terminate!

We want the replacement to take place a finite number of times depending on particular uses of the function with particular arguments but, of course, there is no way of knowing what argument values are required when the function is defined. If we did know then we could construct specific functions for specific cases rather than a general purpose function. This was not a problem in earlier examples because we knew replacement would always be finite. For recursion, though, we need some means of delaying the repetitive use of the function until it is actually required.

4.4. Passing a function to itself

Function use always occurs in an application and may be delayed through abstraction at the point where the function is used. For an arbitrary function, the application:

```
<function> <argument>
```

is equivalent to:

```
λf.(f <argument>) <function>
```

The original function becomes the argument in a new application.

In our addition example we could introduce a new argument:

```
def add1 f x y =
 if iszero y
 then x
 else f (succ x) (pred y)
```

to remove recursion by abstraction at the point where recursion is required. Now we need to find an argument for `add1` with the same effect as `add.` Of course, we cannot just pass `add` to `add1` as we end up with the non-terminating replacement again. What is needed is to pass `add1` into itself but this just pushes the problem down a level. If we try:

```
def add = add1 add1
```

then the definition expands to:

```
(λf.λx.λy.
  if iszero y
```

```
    then x
    else f (succ x) (pred y)) add1 =>

λx.λy.
 if iszero y
 then x
 else add1 (succ x) (pred y)
```

We have failed to pass `add1` down far enough. In the original definition for `add1`, the application:

```
f (succ x) (pred y)
```

has only two arguments. Thus, after substitution:

```
add1 (succ x) (pred y)
```

has no argument corresponding to the bound variable `f.`

We need the effect of:

```
add1 add1 (succ x) (pred y)
```

so that `add1` may be passed on to subsequent recursions.

Let us define an `add2`, this time passing the argument for `f` to the argument itself as well:

```
def add2 f x y =
 if iszero y
 then x
 else f f x y
```

As before, `add` is:

```
def add = add2 add2
```

The definition expands and evaluates as:

```
(λf.λx.λy.
   if iszero y
   then x
   else f f (succ x) (pred y)) add2 =>

λx.λy.
 if iszero y
 then x
 else add2 add2 (succ x) (pred y)
```

Note that we do not strictly need to replace other occurrences of `add2` as its definition contains no references to itself.

Now, we have inserted two copies of `add2` - one as function and another as argument - to continue recursion. Thus, every time the recursion point is reached another copy of the whole function is passed down.

For example:

```
add one two ==

(λx.λy.
   if iszero y
```

```
    then x
    else add2 add2 (succ x) (pred y)) one two  => ... =>

  if iszero two
  then one
  else add2 add2 (succ one) (pred two)  => ... =>

  (λf.λx.λy.
    if iszero y
    then x
    else f f (succ x) (pred y)) add2 (succ one) (pred two)  => ... =>

  if iszero (pred two)
  then (succ one)
  else add2 add2 (succ (succ one)) (pred (pred two)) => ... =>

  (λf.λx.λy.
    if iszero y
    then x
    else f f (succ x) (pred y)) add2 (succ (succ one))
                                     (pred (pred two))  => ... =>

  if iszero (pred (pred two))
  then (succ (succ one))
  else add2 add2 (succ (succ (succ one)))
                 (pred (pred (pred two))) => ... =>

  succ (succ one)) ==

  three
```

4.5. Applicative order

From now on, to simplify the presentation of some examples we will evaluate them partially in applicative order; that is some cases we will evaluate arguments before passing them to functions. We will indicate the applicative order reduction of an argument with:

```
  ->
```

and the applicative order reduction of a sequence of arguments with:

```
  -> ... ->
```

Note that argument evaluation will generally involve other reductions which won't be shown.

We will consider the relationship between applicative and normal order evaluation in chapter 8 but note now that the result of a terminating applicative order reduction of an expression is the same as the result of the equivalent terminating normal order reduction. As we will see in chapter 8, the reverse is not true because there are expressions with terminating normal order reductions but non-terminating applicative order reductions. Nonetheless, provided evaluation terminates, applicative and normal order are equivalent.

As we will also see in chapter 8, a major source of non-termination results from our representation of conditional expressions. It turns out that the strict applicative order evaluation of conditional expressions embodying recursive calls in a function body won't terminate. Thus until chapter 8, the use of the applicative order indicators:

```
  ->
```

and:

```
-> ... ->
```

will still imply the normal order evaluation of conditional expressions.

4.6. Recursion function

A more general approach to recursion is to find a constructor function to build a recursive function from a non recursive function, with a single abstraction at the recursion point.

For example, we might define multiplication recursively. To multiply two numbers, add the first to the product of the first and the decremented second. If the second is zero then so is the product:

```
def mult x y =
 if iszero y
 then zero
 else add x (mult x (pred y))
```

For example:

```
mult three two => ... =>

add three (mult three (pred two)) -> ... ->

add three (add three (mult three (pred (pred two)))) -> ... ->

add three (add three zero) -> ... ->

add three three => ... =>

six
```

We can remove self-reference by abstraction at the recursion point:

```
def mult1 f x y =
 if iszero y
 then zero
 else add x (f x (pred y))
```

We would like to have a function `recursive` which will construct recursive functions from non-recursive versions, for example:

```
def mult = recursive mult1
```

The function `recursive` must not only pass a copy of its argument to that argument but also ensure that self application will continue: the copying mechanism must be passed on as well. This suggests that `recursive` should be of the form:

```
def recursive f = f <’f’ and copy>
```

If `recursive` is applied to `mult1:`

```
recursive mult1 ==

λf.(f <’f’ and copy>) mult1 =>
```

```
mult1 <'mult1' and copy> ==

(λf.λx.λy.
  if iszero y
  then zero
  else add x (f x (pred y))) <'mult1' and copy> =>

λx.λy.
 if iszero y
 then zero
 else add x (<'mult1' and copy> x (pred y))
```

In the body we have:

```
<'mult1' and copy> x (pred y)
```

but we require:

```
mult1 <'mult1' and copy> x (pred y)
```

so that:

```
<'mult1' and copy>
```

gets passed on again through `mult1`'s bound variable `f` to the next level of recursion. Thus, the copy mechanism must be such that:

```
<'mult1' and copy> => ... => mult1 <'mult1' and copy>
```

In general, from function `f` passed to `recursive,` we need:

```
<'f' and copy> => ... => f <'f' and copy>
```

so the copy mechanism must be an application and that application must be self-replicating.

We know that the self-application function:

```
λs.(s s)
```

will self-replicate when applied to itself but the replication never ends.

Self-application may be delayed through abstraction with the construction of a new function:

```
λf.λs.(f (s s))
```

Here, the self-application:

```
(s s)
```

becomes an argument for `f.` This might, for example, be a function with a conditional expression in its body which will only lead to the evaluation of its argument when some condition is met.

When this new function is applied to an arbitrary function, we get

```
λf.λs.(f (s s)) <function> =>

λs.(<function> (s s))
```

If this function is now applied to itself:

```
λs.(<function> (s s)) λs.(<function> (s s)) =>

<function> (λs.(<function> (s s)) λs.(<function> (s s)))
```

then we have a copy mechanism which matches our requirement.

Thus, we can define `recursive` as:

```
def recursive f = λs.(f (s s)) λs.(f (s s))
```

For example, in:

```
def mult = recursive mult1
```

the definition evaluates to:

```
λf.(λs.(f (s s)) λs.(f (s s))) mult1 =>

λs.(mult1 (s s)) λs.(mult1 (s s)) =>

mult1 (λs.(mult1 (s s)) λs.(mult1 (s s))) ==

(λf.λx.λy.
  if iszero y
  then zero
  else add x (f x (pred y))) (λs.(mult1 (s s)) λs.(mult1 (s s))) =>

λx.λy.
 if iszero y
 then zero
 else add x ((λs.(mult1 (s s)) λs.(mult1 (s s)))
             x (pred y))
```

Again, note that we don't strictly need to replace other occurrences of `mult1` as its definition contains no references to itself.

For example, we will try:

```
mult three two => ... =>

(λx.λy.
  if iszero y
  then zero
  else add x ((λs.(mult1 (s s)) λs.(mult1 (s s)))
             x (pred y))) three two  => ... =>

if iszero two
then zero
else add three ((λs.(mult1 (s s)) λs.(mult1 (s s)))
                three (pred two))  => ... =>

add three ((λs.(mult1 (s s)) λs.(mult1 (s s)))
           three (pred two)) ->

add three (mult1 (λs.(mult1 (s s)) λs.(mult1 (s s)))
                  three (pred two)) ==
add three ((λx.λy.
             if iszero y
             then zero
             else add x ((λs.(mult1 (s s)) λs.(mult1 (s s)))
                         x (pred y))) three (pred two))  -> ... ->

add three if iszero (pred two)
          then zero
          else add three ((λs.(mult1 (s s)) λs.(mult1 (s s)))
                          three (pred (pred two)))  -> ... ->

add three (add three ((λs.(mult1 (s s)) λs.(mult1 (s s)))
                      three (pred (pred two)))) ->

add three (add three (mult1 (λs.(mult1 (s s)) λs.(mult1 (s s)))
                            three (pred (pred two)))) ==

add three (add three ((λx.λy.
                        if iszero y
                        then zero
                        else add x ((λs.(mult1 (s s)) λs.(mult1 (s s)))
                                    x (pred y))
                      three (pred (pred two))))  -> ... ->

add three (add three if iszero (pred (pred two))
                     then zero
                     else add three ((λs.(mult1 (s s)) λs.(mult1 (s s)))
                                     three (pred (pred (pred two))))) -> ... ->

add three (add three zero) -> ... ->

add three three => ... =>

six
```

4.7. Recursion notation

The function `recursive` is known as a **paradoxical combinator** or a **fixed point finder**, and is called **Y** in the λ calculus literature.

Rather than always defining an auxiliary function with an abstraction and then using `recursive` to construct a recursive version, we will allow the defined name to appear in the defining expression but use a new definition form:

```
rec <name> = <expression>
```

This is to indicate that the occurrence of the name in the definition should be replaced using abstraction and the paradoxical combinator should then be applied to the whole of the defining expression. For example, for addition, we will write:

```
rec add x y =
 if iszero y
 then x
 else add (succ x) (pred y)
```

instead of:

```
def add1 f x y =
 if iszero y
 then x
 else f (succ x) (pred y)

def add = recursive add1
```

and for multiplication we will write:

```
rec mult x y =
 if iszero y
 then zero
 else add x (mult x (pred y))
```

When we expand or evaluate a recursive definition we will just leave the recursive reference in place.

4.8. Arithemtic operations

We will now use recursion to define arithmetic operations for raising to a power, subtraction, equality and inequalities, and division.

4.8.1. Power

To raise one number to the power of another number, multiply the first by the first to the power of the decremented second. If the second is zero then the power is one:

```
rec power x y =
 if iszero y
 then one
 else mult x (power x (pred y))
```

For example:

```
power two three => ... =>

mult two
     (power two (pred three)) -> ... ->

mult two
     (mult two
           (power two (pred (pred three)))) -> ... ->

mult two
     (mult two
           (mult two
                 (power two (pred (pred (pred three)))))) -> ... ->

mult two
     (mult two
           (mult two one)) -> ... ->

mult two
     (mult two two) -> ... ->

mult two four => ... =>
```

```
eight
```

4.8.2. Subtraction

To find the difference between two numbers, find the difference between the numbers after decrementing both. The difference between a number and zero is the number:

```
rec sub x y =
 if iszero y
 then x
 else sub (pred x) (pred y)
```

For example:

```
sub four two => ... =>

sub (pred four) (pred two) => ... =>

sub (pred (pred four)) (pred (pred two)) => ... =>

(pred (pred four)) => ... =>

two
```

Notice that this version of subtraction will return zero if the second number is larger than the first, for example:

```
sub one two => ... =>

sub (pred one) (pred two) => ... =>

sub (pred (pred one)) (pred (pred two)) => ... =>

pred (pred one) -> ... ->

pred zero => ... =>

zero
```

This is because `pred` returns zero from decrementing zero.

This form of subtraction is known as **natural subtraction**.

4.8.3. Comparison

There are a number of ways of defining equality between numbers. One approach is to notice that the difference between two equal numbers is zero. However, if we subtract a number from a smaller number we also get zero so we need to find the **absolute difference** between them; the difference regardless of the order of comparison.

To find the absolute difference between two numbers, add the difference between the first and the second to the difference between the second and the first:

```
def abs_diff x y = add (sub x y) (sub y x)
```

If they are both the same then the absolute differences will be zero because the result of taking each from the other will be zero. If the first is greater than the second then the absolute difference will be the first minus the second because the second minus the first will be zero. Similarly, if the second is greater than the first then the difference will be the second minus the first because the first minus the second will be zero.

Thus, we can define:

```
def equal x y = iszero (abs_diff x y)
```

For example:

```
equal two three => ... =>

iszero (abs_diff two three) -> ... ->

iszero (add (sub two three) (sub three two)) -> ... ->

iszero (add zero one) -> ... ->

iszero one => ... =>

false
```

We could equally well be explicit about the decrementing `sub` carries out and define equality recursively. Two numbers are equal if both are zero, they are unequal if one is zero or equal if decrementing both gives equal numbers:

```
rec equal x y =
 if and (iszero x) (iszero y)
 then true
 else
  if or (iszero x) (iszero y)
  then false
  else equal (pred x) (pred y)
```

For example:

```
equal two two => ... =>

equal (pred two) (pred two) -> ... ->

equal one one => ... =>

equal (pred one) (pred one) -> ... ->

equal zero zero => ... =>

true
```

We can also use subtraction to define arithmetic inequalities. For example, a number is greater than another if subtracting the second from the first gives a non-zero result:

```
def greater x y = not (iszero (sub x y))
```

For example, for

```
3 > 2
```

we use:

```
greater three two => ... =>

not (iszero (sub three two)) -> ... ->

not (iszero one) -> ... ->

not false => ... =>

true
```

Similarly, a number is greater than or equal to another if taking the first from the second gives zero:

```
def greater_or_equal x y = iszero (sub y x)
```

For example, for:

```
2 >= 3
```

we use:

```
greater_or_equal two three => ... =>

iszero (sub three two) -> ... ->

iszero one => ... =>

false
```

4.8.4. Division

Division, like decrementation, is problematic because of zero. It is usual to define division by zero as undefined but we do not have any way of dealing with undefined values. Let us define division by zero to be zero and remember to check for a zero divisor. For a non-zero divisor, we count how often it can be subtracted from the dividend until the dividend is smaller than the divisor:

```
rec div1 x y =
 if greater y x
 then zero
 else succ (div1 (sub x y) y)

def div x y =
 if iszero y
 then zero
 else div1 x y
```

For example:

```
div nine four => ... =>

div1 nine four => ... =>

succ (div1 (sub nine four) four)) -> ... ->

succ (div1 five four) -> ... ->

succ (succ (div1 (sub five four) four)) -> ... ->
succ (succ (div1 one four)) -> ... ->

succ (succ zero) -> ... ->

two
```

4.9. Summary

In this chapter we have:

- considered recursion as a means of repetition
- seen that recursion through function definitions leads to non-terminating substitution sequences
- introduced recursion by abstracting at the place where recursion takes place in a function and then passing the function to itself
- met applicative order β reduction
- generalised recursion through a recursion function which substitutes a function at its own recursion points
- introduced notation for defining recursive functions
- used recursion to develop standard arithmetic operations

Some of these topics are summarised below.

Recursion by passing a function to itself

For:

```
def <name> = ... (<name> ...) ...
```

write:

```
def <name1> f  = ... (f f ... ) ...

def <name> = <name1> <name1>
```

Applicative order β reduction

For `(<function expression> <argument expression>)`

i) applicative order β reduce `<argument expression>` to `<argument value>`

ii) applicative order β reduce `<function expression>` to `<function value>`

iii) if `<function value>` is λ`<name>.<body>`
then replace all free occurences of `<name>` in `<body>` with `<argument value>` and applicative order β reduce the new `<body>`

or

iv) if `<function value>` is not a function
then return `(<function value> <argument value>)`

Applicative order reduction notation

`->` - applicative order β reduction

`-> ... ->` - multiple applicative order β reduction

Recursion function

```
def recursive f = λs.(f (s s)) λs.(f (s s))
```

For:

```
def <name> = ... (<name> ... ) ...
```

write:

```
def <name1> f = ... (f ... ) ...

def <name> = recursive <name1>
```

Note that:

```
recursive <name1> => ... => <name1> (recursive <name1>)
```

Recursion notation

```
rec <name> = <expression using ‘<name>’> ==

def <name> = recursive λf.<expression using ‘f’>
```

4.10. Exercises

1) The following function finds the sum of the numbers between `n` and `zero:`

```
def sum1 f n =
 if iszero n
 then zero
 else add n (f (pred n))

def sum = recursive sum1
```

Evaluate:

```
sum three
```

2) Write a function that finds the product of the numbers between `n` and `one:`

```
def prod1 f n = ...

def prod = recursive prod1 (*)
```

so that:

```
prod n
```

in lambda calculus is equivalent to:

```
n * n-1 * n-2 * ... * 1
```

in normal arithmetic.

Evaluate:

```
prod three
```

3) Write a function which finds the sum of applying a function `fun` to the numbers between `n` and `zero:`

```
def fun_sum1 f fun n = ...

def fun_sum = recursive fun_sum1
```

For example, given the ‘squaring’ function:

```
def sq x = mult x x
```

then:

```
fun_sum sq three
```

in the λ calculus is equivalent to:

$0^2 + 1^2 + 2^2 + 3^2$

in arithmetic.

Evaluate:

```
fun_sum double three
```

given the ‘doubling’ function:

```
def double x = add x x
```

4) Define a function to find the sum of applying a function `fun` to the numbers between `n` and `zero` in steps of `s:`

```
def fun_sum_step1 f fun n s = ...

def fun_sum_step = recursive fun_sum_step1
```

so, for example:

```
fun_sum_step sq six two
```

in the λ calculus is equivalent to:

$6^2 + 4^2 + 2^2 + 0^2$

in normal arithmetic.

Evaluate:

(*) There’s no escape from ‘factorial’...

```
i)  fun_sum_step double five two
ii) fun_sum_step double four two
```

5) Define functions to test whether or not a number is less than, or less than or equal to another number:

```
def less x y = ...

def less_or_equal x y = ...
```

Evaluate:

```
i)   less three two
ii)  less two three
iii) less two two
iv)  less_or_equal three two
v)   less_or_equal two three
vi)  less_or_equal two two
```

6) Define a function to find the remainder on dividing one number by another:

```
def mod x y = ...
```

Evaluate:

```
i)   mod three two
ii)  mod two three
iii) mod three zero
```

5. TYPES

5.1. Introduction

In this chapter we are going to consider how types can be added to our functional notation to ensure that only meaningful arguments are passed to functions.

To begin with, we will consider the role of types in programming in general and how types may be characterised. We will then introduce functions for constructing and manipulating typed values, using the pair manipulation functions to represent typed objects as type/value pairs.

Next, we will introduce the error type for error objects which are returned after type errors. We will then develop typed representations for booleans, numbers and characters.

Finally, we will introduce new notations to simplify function definitions through case definitions and structure matching.

5.2. Types and programming

We are working with a very simple language. As we exclude single names as expressions, the only objects are functions which take function arguments and return function results. (For the moment, we won't consider non-terminating applications.) We have constructed functions which we can interpret as boolean values, boolean operations, numbers, arithmetic operations and so on but particular functions have no intrinsic interpretations other than in terms of their effects on other functions. Because functions are so general, there is no way to restrict the application of functions to specific other functions, for example we cannot restrict 'arithmetic' functions to 'numeric' operands. We can carry out function applications which are perfectly valid but have results with no relevant meaning