```
i)   λx.λy.(λx.y λy.x)
ii)  λx.(x (λy.(λx.x y) x))
iii) λa.(λb.a λb.(λa.a b))
iv)  (λfree.bound λbound.(λfree.free bound))
v)   λp.λq.(λr.(p (λq.(λp.(r q)))) (q p))
```

6) Remove any name clashes in the expressions in exercise 5 above.

3. CONDITIONS, BOOLEANS AND INTEGERS

3.1. Introduction

In this chapter we are going to start to add layers to the λ calculus to develop a higher level functional notation.

First of all we will use the pair functions from chapter 1 to represent conditional expressions with truth values `true` and `false.` We will then use these to develop boolean operations like `not, and` and `or.`

Next we will use the pair functions to represent natural numbers in terms of the value `zero` and the successor function.

Finally, we will introduce notations for simplifying function definitions and λ expressions, and for an 'if .. then ... else' form of conditional expression.

For the moment we will be looking at untyped representations of truth values and functions. We will develop typed representations in chapter 5.

3.2. Truth values and conditional expression

Boolean logic is based on the truth values `TRUE` and `FALSE` with logical operations `NOT, AND, OR` and so on.

We are going to represent `TRUE` by `select_first` and `FALSE` by `select_second`, and use a version of `make_pair` to build logical operations. To motivate this, consider the C conditional expression:

```
<condition>?<expression>:<expression>
```

If the `<condition>` is `TRUE` then the first `<expression>` is selected for evaluation and if the `<condition>` is `FALSE` then the second `<expression>` is selected for evaluation.

For example, to set `max` to the greater of `x` and `y:`

```
max = x>y?x:y
```

or to set `absx` to the absolute value of `x:`

```
absx = x<0?-x:x
```

We can model a conditional expression using a version of the make pair function:

```
def cond = λe1.λe2.λc.((c e1) e2)
```

Consider `cond` applied to the arbitrary expressions `<expression1>` and `<expression2>`:

```
((cond <expression1>) <expression2>) ==
```

```
((λe1.λe2.λc.((c e1) e2) <expression1>) <expression2>) =>

(λe2.λc.((c <expression1>) e2) <expression2>) =>

λc.((c <expression1>) <expression2>)
```

Now, if this function is applied to `select_first`:

```
(λc.((c <expression1>) <expression2>) select_first) =>

((select_first <expression1>) <expression2>) => ... =>

<expression1>
```

and if it is applied to `select_second`:

```
(λc.((c <expression1>) <expression2>) select_second) =>

((select_second <expression1>) <expression2>) => ... =>

<expression2>
```

Notice that the `<condition>` is the last argument for `cond`, not the first.

Now, we will use the conditional expression and `cond` function with:

```
def true = select_first

def false = select_second
```

to model some of the logical operators.

3.3. NOT

`NOT` is a unary operator of the form:

```
NOT <operand>
```

which we will describe through a truth table with `X` standing for the single operand:

```
   X   | NOT X
 ------+------
 FALSE | TRUE
 TRUE  | FALSE
```

Note that if the operand is `TRUE` then the answer is `FALSE` and if the operand is `FALSE` then the answer is `TRUE`. Thus `NOT` could be written using an conditional expression as:

```
X ? FALSE : TRUE
```

We can describe this using selectors so if the operand is `TRUE` then `FALSE` is selected and if the operand is `FALSE` then `TRUE` is selected. This suggests using:

```
def not = λx.(((cond false) true) x)
```

Simplifying the inner body gives:

```
(((cond false) true) x) ==
(((λe1.λe2.λc.((c e1) e2) false) true) x) =>
((λe2.λc.((c false) e2) true) x) =>
(λc.((c false) true) x) =>
((x false) true)
```

so we will use:

```
def not = λx.((x false) true)
```

Let us try:

```
NOT TRUE
```

as:

```
(not true) ==
(λx.((x false) true) true) =>
((true false) true) ==
((λfirst.λsecond.first false) true) =>
(λsecond.false true) =>
false
```

and:

```
NOT FALSE
```

as:

```
(not false) ==
((λx.((x false) true) false) =>
((false false) true) ==
((λfirst.λsecond.second false) true) =>
(λsecond.second true) =>
true
```

which correspond to the truth table.

3.4. AND

`AND` is a binary operator of the form:

```
<operand> AND <operand>
```

which we will describe with a truth table with X standing for the left operand and Y standing for the right operand:

```
   X   |   Y   | X AND Y
-------+-------+--------
 FALSE | FALSE | FALSE
 FALSE | TRUE  | FALSE
 TRUE  | FALSE | FALSE
 TRUE  | TRUE  | TRUE
```

Note that if the left operand is TRUE then the final value depends on the right operand and if the left operand is FALSE then the final value is FALSE so AND could be modelled using the conditional expression as:

```
X ? Y : FALSE
```

Using selectors, if the left operand is TRUE then select the right operand and if the left operand is FALSE then select FALSE, so we will define AND as:

```
def and = λx.λy.(((cond y) false) x)
```

Simplifying the inner body gives:

```
(((cond y) false) x) ==

(((λe1.λe2.λc.((c e1) e2) y) false) x) =>

((λe2.λc.((c y) e2) false) x) =>

(λc.((c y) false) x) =>

((x y) false)
```

so we will now use:

```
def and = λx.λy.((x y) false)
```

For example, we could write:

```
TRUE AND FALSE
```

as:

```
((and true) false) ==

((λx.λy.((x y) false) true) false) =>

(λy.((true y) false) false) =>

((true false) false) ==

((λfirst.λsecond.first false) false) =>

(λsecond.false false) =>

false
```

3.5. OR

OR is a binary operator of the form:

```
<operand> OR <operand>
```

which we will again describe with a truth table using X for the left operand and Y for the right operand:

```
   X   |   Y   | X OR Y
-------+-------+--------
 FALSE | FALSE | FALSE
 FALSE | TRUE  | TRUE
 TRUE  | FALSE | TRUE
 TRUE  | TRUE  | TRUE
```

Note that if the first operand is TRUE then the final value is TRUE and otherwise the final value is the second operand, so we could describe this using the conditional expression as:

```
X ? TRUE : Y
```

Using selectors, if the first operand is TRUE then select TRUE and if the first operand is FALSE then select the second operand:

```
def or = λx.λy.(((cond true) y) x)
```

Simplifying the inner body:

```
(((cond true) y) x) ==

(((λe1.λe2.λc.((c e1) e2) true) y) x) =>

((λe2.λc.((c true) e2) y) x) =>

(λc.((c true) y) x) =>

((x true) y)
```

Now we will use:

```
def or = λx.λy.((x true) y)
```

For example, we could write:

```
FALSE OR TRUE
```

as:

```
((or false) true) ==

((λx.λy.((x true) y) false) true) =>

(λy.((false true) y) true) =>

((false true) true) =>

((λfirst.λsecond.second true) true) =>

(λsecond.second true) =>
```

```
true
```

3.6. Natural numbers

We tend to take numbers for granted in programming but we now have to consider how to represent them explicitly. Our approach will be based on the ability to define **natural numbers** - non-negative integers - as **successors** of **zero:**

1 = successor of 0

2 = successor of 1
  = successor of successor of 0

3 = successor of 2
  = successor of successor of 1
  = successor of successor of successor of 0

etc.

Thus, the definition of an arbitrary integer will be that number of successors of zero. We need to find a function `zero` to represent zero and a successor function `succ` so that we can define:

```
def one = (succ zero)

def two = (succ one)

def three = (succ two)
```

and so on.

Note that:

```
two == (succ (succ zero))

three == (succ (succ one)) == (succ (succ (succ zero)))
```

and so on.

There are a variety of ways of representing `zero` and `succ.` We will use:

```
def zero = identity

def succ = λn.λs.((s false) n)
```

so each time `succ` is applied to a number `n` it builds a pair function with `false` first and the original number second. For example:

```
one ==

(succ zero) ==

(λn.λs.((s false) n) zero) =>

λs.((s false) zero)
```

Similarly:

```
two ==

(succ one) ==

(λn.λs.((s false) n) one) =>

λs.((s false) one) ==

λs.((s false) λs.((s false) zero))
```

and:

```
three ==

(succ two) ==

(λn.λs.((s false) n) two) =>

λs.((s false) two) ==

λs.((s false) λs.((s false) one) ==

λs.((s false) λs.((s false) λs.((s false) zero)))
```

This representation enables the definition of a unary function `iszero` which returns `true` if its argument is `zero` and `false` otherwise. Remember that a number is a function with an argument which may be used as a selector. For an arbitrary number:

```
λs.((s false) <number>)
```

if the argument is set to `select_first` then `false` will be selected:

```
(λs.((s false) <number>) select_first) =>

((select_first false) <number>) ==

((λfirst.λsecond.first false) <number>) =>

(λsecond.false <number>) =>

false
```

If `zero`, which is the identity function, is applied to `select_first` then `select_first`, which is the same as `true` by definition, will be returned:

```
(zero select_first) ==

(λx.x select_first) =>

select_first ==

true
```

This suggests using:

```
def iszero = λn.(n select_first)
```

Notice that `iszero` applies the number to the selector rather than the selector to the number. This is because our number representation models numbers as functions with selector arguments.

We can now define the predecessor function `pred` so that:

```
(pred one) => ... => zero

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

(pred three) => ... => two
```

and so on.

For our number representation, `pred` should strip off a layer of nesting from an arbitrary number:

```
λs.((s false) <number>)
```

and return the:

```
<number>
```

This suggests using `select_second` because:

```
(λs.((s false) <number>) select_second) =>

((select_second false) <number>) ==

((λfirst.λsecond.second false) <number>) =>

(λsecond.second <number>) =>

<number>
```

so we might define a first version of `pred` as:

```
def pred1 = λn.(n select_second)
```

However, there is a problem with `zero` as we only have positive integers. Let us try our present `pred1` with `zero:`

```
(pred1 zero) ==

(λn.(n select_second) zero) =>

(zero select_second) ==

(λx.x select_second) =>

select_second ==

false
```

which is not a representation of a number.

We could define the predecessor of `zero` to be `zero` and check numbers to see if they are `zero` before returning their predecessor, using:

```
<number> = zero ? zero : predecessor of <number>
```

So:

```
def pred = λn.(((cond zero) (pred1 n)) (iszero n))
```

Simplifying the body gives:

```
(((cond zero) (pred1 n)) (iszero n)) ==

(((λe1.λe2.λc.((c e1) e2) zero) (pred1 n)) (iszero n)) =>

((λe2.λc.((c zero) e2) (pred1 n)) (iszero n)) =>

(λc.((c zero) (pred1 n)) (iszero n))  =>

(((iszero n) zero) (pred1 n))
```

Substituting for `pred1` and simplifying gives:

```
(((iszero n) zero) (λn.(n select_second) n)) ==

(((iszero n) zero) (n select_second)) ==
```

so now we will use:

```
def pred = λn.(((iszero n) zero) (n select_second))
```

Alternatively, we might say that the predecessor of `zero` is undefined. We won't look at how to handle undefined values here.

When we use `pred` we will have to be careful to check for a `zero` argument.

3.7. Simplified notations

By now you will have noticed that manipulating λ expressions involves lots of brackets. As well as being tedious and fiddley, it is a major source of mistakes due to unmatched or mismatched brackets. To simplify things, we will allow brackets to be omitted when it is clear what is intended. In general, for the application of a function `<function>` to `N` arguments we will allow:

```
<function> <argument1> <argument2> ... <argumentN>
```

instead of:

```
(...((<function> <argument1>) <argument2>) ... <argumentN>)
```

so in a function application, a function is applied first to the nearest argument to the right. If an argument is itself a function application then the brackets must stay. There must also be brackets round function body applications. For example, we could re-write `pred` as:

```
def pred = λn.((iszero n) n (n select_second))
```

We can also simplify name/function association definitions by dropping the λ and `.,` and moving the bound variable to the left of the = so:

```
def <names> = λ<name>.<expression>
```

where `<names>` is one or more `<name>s` becomes:

```
def <names> <name> = <expression>
```

We can now re-write all our definitions:

```
def identity x = x

def self_apply s = s s

def apply func = λarg.(func arg)
```

and hence:

```
def apply func arg = func arg

def select_first first = λsecond.first
```

and hence:

```
def select_first first second = first

def select_second first = λsecond.second
```

and hence:

```
def select_second first second = second

def make_pair e1 = λe2.λc.(c e1 e2)
```

and hence:

```
def make_pair e1 e2 = λc.(c e1 e2)
```

and hence:

```
def make_pair e1 e2 c = c e1 e2

def cond e1 e2 c = c e1 e2

def true first second = first

def false first second = second

def not x = x false true

def and x y = x y false

def or x y = x true y
```

For some functions there are standard equivalent notations. Thus, it is usual to write:

```
cond <true choice> <false choice> <condition>
```

in an `if ... then ... else ...` form. We will use:

```
if <condition>
then <true choice>
else <false choice>
```

For example, we could re-write `pred`'s definition as:

```
def pred n =
 if iszero n
 then zero
 else n select_second
```

Similarly, using our conditional derivation of booleans, we could rewrite `not` as:

```
def not x =
 if x
 then false
 else true
```

and `and` as:

```
def and x y =
 if x
 then y
 else false
```

and `or` as:

```
def or x y =
 if x
 then true
 else y
```

3.8. Summary

In this chapter we have:

- developed representations for conditional expressions and truth values, and used them to develop boolean operations
- developed a representation for natural numbers based on zero and the successor function
- introduced notations for removing brackets from expressions, simplifying function definitions and an 'if .. then ... else ...' form of conditional expression

Some of these topics are summarised below.

Removing brackets

```
( ... ((<function> <argument1>) <argument2>) ... <argumentN>) ==

<function> <argument1> <argument2> ... <argumentN>
```

Simplifying function definitions

```
def <names> = λ<name>.<expression> ==

def <names> <name> = <expression>
```

if ... then ... else ...

```
if <condition>
then <true choice>
else <false choice> ==

cond <true choice> <false choice> <condition>
```

3.9. Exercises

1) The boolean operation `implication` is defined by the following truth table:

```
   X   |   Y   | X IMPLIES Y
-------+-------+------------
 FALSE | FALSE |   TRUE
 FALSE | TRUE  |   TRUE
 TRUE  | FALSE |   FALSE
 TRUE  | TRUE  |   TRUE
```

Define a lambda calculus representation for `implication:`

```
def implies = λx.λy...
```

Show that the definition satisfies the truth table for all boolean values of `x` and `y.`

2) The boolean operation `equivalence` is defined by the following truth table:

```
   X   |   Y   | X EQUIV Y
-------+-------+------------
 FALSE | FALSE |   TRUE
 FALSE | TRUE  |   FALSE
 TRUE  | FALSE |   FALSE
 TRUE  | TRUE  |   TRUE
```

Define a lambda calculus representation for `equivalence:`

```
def equiv = λx.λy...
```

Show that the definition satisfies the truth table for all boolean values of `x` and `y.`

3) For each of the following pairs, show that functions `a)` and `b)` are equivalent for all boolean values of their arguments:

```
i)   a) λx.λy.(and (not x) (not y))
     b) λx.λy.(not (or x y))

ii)  a) implies
     b) λx.λy.(implies (not y) (not x))

iii) a) not
     b) λx.(not (not (not x)))

iv)  a) implies
     b) λx.λy.(not (and x (not y)))

v)   a) equiv
     b) λx.λy.(and (implies x y) (implies y x))
```

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.