Hints for Questions

Chapter 1
Starting Off

1

Try to work these out on paper, and then check by typing them in. Remember that the type of an expression is the type of the value it will evaluate to. Can you show the steps of evaluation for each expression?

2

Type each expression in. What number does each evaluate to? Can you work out which operator (`mod` or `+`) is being calculated first?

3

Type it in. What does OCaml print? What is the evaluation order?

7

What if a value of 2 appeared? How might we interpret it?

Chapter 2
Names and Functions

1

The function takes one integer, and returns that integer multiplied by ten. So what must its type be?

2

What does the function take as arguments? What is the type of its result? So what is the whole type? You can use the `<>` and `&&` operators here.

3

This will be a recursive function, so remember to use **`let rec`**. What is the sum of all the integers from $1 \ldots 1$? Perhaps this is a good base case.

4

This will be a recursive function. What happens when you raise a number to the power 0? What about the power 1? What about a higher power?

5

Can you define this in terms of the `isvowel` function we have already written?

6

Try adding parentheses to the expression in a way which does not change its meaning. Does this make it easier to understand?

7

When does it not terminate? Can you add a check to see when it might happen, and return 0 instead?

Chapter 3
Case by Case

1

We are pattern matching on a boolean value, so there are just two cases: `true` and `false`.

2

Convert the **`if ... then ... else`** structure of the `sum` function from the previous chapter into a pattern matching structure.

3

You will need three cases as before – when the power is 0, 1 or greater than 1 – but now in the form of a pattern match.

5

Consider where parentheses might be added without altering the expression.

6

There will be two cases in each function – the special range pattern `x..y`, and _ for any other character.

Chapter 4
Making Lists

1

Consider three cases: (1) the argument list is empty, (2) the argument list has one element, (3) the argument list has more than one element `a::b::t`. In the last case, which element do we need to miss out?

2

The function will have type **bool list→ int**. Consider the empty list, the list with `true` as its head, and the list with `false` as its head. Count one for each `true` and zero for each `false`.

3

The function to make a palindrome is trivial; to detect if a list is a palindrome, consider the definition of a palindrome – a list which equals its own reverse.

4

Consider the cases (1) the empty list, (2) the list with one element, and (3) the list with more than one element. For the tail recursive version, use an accumulating argument.

5

Can any element exist in the empty list? If the list is not empty, it must have a head and a tail. What is the answer if the element we are looking for is equal to the head? What do we do if it is not?

6

The empty list is already a set. If we have a head and a tail, what does it tell us to find out if the head exists within the tail?

7

Consider in which order the `@` operators are evaluated in the reverse function. How long does each append take? How many are there?

Chapter 5
Sorting Things

1

Consider adding another **`let`** before **`let`** `left` and **`let`** `right`.

2

Consider the situations in which `take` and `drop` can fail, and what arguments `msort` gives them at each recursion.

3

This is a simple change – consider the comparison operator itself.

4

What will the type of the function be? Lists of length zero and one are already sorted – so these will be the base cases. What do we do when there is more than one element?

6

You can put one **`let rec`** construct inside another.

Chapter 6
Functions upon Functions upon Functions

1

The function `calm` is simple recursion on lists. There are three cases – the empty list, a list beginning with `'!'` and a list beginning with any other character. In the second part of the question, write a function `calm_char` which processes a single character. You can then use `map` to define a new version of `calm`.

2

This is the same process as Question 1.

3

Look back at the section on anonymous functions. How can `clip` be expressed as an anonymous function? So, how can we use it with `map`?

4

We want a function of the form **`let rec`** `apply f n x =` ... which applies `f` to `x` a total of `n` times. What is the base case? What do we do in that case? What otherwise?

5

You will need to add the extra function as an argument to both `insert` and `sort` and use it in place of the `<=` operator in `insert`.

6

There are three possibilities: the argument list is empty, `true` is returned when its head is given to the function `f`, or `false` is returned when its head is given to the function `f`.

7

If the input list is empty, the result is trivially true – there cannot possibly be any elements for which the function does not hold. If not, it must hold for the first one, and for all the others by recursion.

8

You can use `map` on each α **list** in the α **list list**.

Chapter 7
When Things Go Wrong

1

Make sure to consider the case of the empty list, where there is no smallest positive element, and also the non-empty list containing entirely zero or negative numbers.

2

Just put an exception handler around the function in the previous question.

3

First, write a function to find the number less than or equal to the square root of its argument. Now, define a suitable exception, and wrap up your function in another which, on a bad argument, raises the exception or otherwise calls your first function.

4

Use the **`try`** ... **`with`** construct to call your function and handle the exception you defined.

Chapter 8
Looking Things Up

1

The keys in a dictionary are unique – does remembering that fact help you?

2

The type will be the same as for the `add` function, but we only replace something if we find it there – when do we know we will not find it?

3

The function takes a list of keys and a list of values, and returns a dictionary. So it will have type α **list** $\rightarrow$ β **list** $\rightarrow$ (α $\times$ β) **list**. Try matching on both lists at once – what are the cases?

4

This function takes a list of pairs and produces a pair of lists. So its type must be (α $\times$ β) **list** $\rightarrow$ α **list** $\times$ β **list**.

For the base case (the empty dictionary), we can see that the result should be `([], [])`. But what to do in the case we have `(k, v) :: more`? We must get names for the two parts of the result of our function on `more`, and then cons `k` and `v` on to them – can you think of how to do that?

5

You can keep a list of the keys which have already been seen, and use the `member` function to make sure you do not add to the result list a key-value pair whose key has already been included.

6

The function will take two dictionaries, and return another – so you should be able to write down its type easily.

Try pattern matching on the first list – when it is empty, the answer is trivial – what about when it has a head and a tail?

Chapter 9
More with Functions

2

Try building a list of booleans, each representing the result of `member` on a list.

3

The `/` operator differs from the $*$ operator in an important sense. What is it?

4

The type of `map` is (α $\rightarrow$ β) $\rightarrow$ α **list** $\rightarrow$ β **list**. The type of `mapl` is (α $\rightarrow$ β) $\rightarrow$ α **list list** $\rightarrow$ β **list list**. So, what must the type of `mapll` be? Now, look at our definition of `mapl` – how can we extend it to lists of lists of lists?

5

Use our revised `take` function to process a single list. You may then use `map` with this (partially applied) function to build the `truncate` function.

6

Build a function `firstelt` which, given the number and a list, returns the first element or that number. You can then use this function (partially applied) together with `map` to build the main `firstelts` function.

Chapter 10
New Kinds of Data

1

The type will have two constructors: one for squares, requiring only a single integer, and one for rectangles, requiring two: one for the width and one for the height.

2

The function will have type rect $\rightarrow$ **int**. Work by pattern matching on the two constructors of your type.

3

Work by pattern matching on your type. What happens to a square. What to a rectangle?

4

First, we need to rotate the rectangles as needed – you have already written something for this. Then, we need to sort them according to width. Can you use our `sort` function which takes a custom comparison function for this?

5

Look at how we re-wrote `length` and `append` for the sequence type.

6

Add another constructor, and amend `evaluate` as necessary.

7

Handle the exception, and return `None` in that case.

Chapter 11
Growing Trees

1

The type will be $\alpha \rightarrow \alpha$ tree $\rightarrow$ **bool**. That is, it takes an element to search for, and a tree containing elements of the same type, and returns `true` if the element is found, and `false` if not. What happens if the tree is a leaf? What if it is a branch?

2

The function will have type α tree $\rightarrow \alpha$ tree. What happens to a leaf? What must happen to a branch and its sub-trees?

3

If the two trees are both `Lf`, they have the same shape. What if they are both branches? What if one is a branch and the other a leaf or vice versa? For the second part of the question, consider a devious way to use `map_tree` to produce trees of like type.

4

We have already written a function for inserting an element into an existing tree.

5

Try using list dictionaries as an intermediate representation. We already know how to build a tree from a list.

6

Consider using a list of sub-trees for a branch. How can we represent a branch which has no sub-trees?

Chapter 12
In and Out

1

You can use the `print_string` and `print_int` functions. Be careful about what happens when you

print the last number.

2

You can use the `read_int` function to read an integer from the user. Be sure to give the user proper instructions, and to deal with the case where `read_int` raises an exception (which it will if the user does not type an integer).

3

One way would be to ask the user how many dictionary entries they intend to type in first. Then we do not need a special code to signal the end of input.

4

Try writing a function to build a list of integers from 1 to n. Can you use that to build the table and print it? The `iter` and/or `map` functions may come in useful. Deal with a channel in your innermost function – the opening and closing of the file can be dealt with elsewhere.

5

The `input_line` function can be used – how many times can you call it until `End_of_file` is raised?

6

We can read lines from the file using `input_line` and write using `output_string` – make sure the newlines do not get lost! How do we know when we are done? Write a function to copy a line from one channel to another – we can deal with opening and closing the files separately.

Chapter 13
Putting Things in Boxes

1

Consider the initial values of the references, and then work through how each one is altered by each part of the expression. What is finally returned as the result of the expression?

2

Try creating a value for each list in OCaml. Now try getting the head of the list, which is a reference, and updating its contents to another integer. What has happened in each case?

3

Try writing a function `forloop` which takes a function to be applied to each number, and the start and end numbers. It should call the given function on each number. What should happen when the start number is larger than the end number?

4

Type them in if you are stuck. Can you work out why each expression has the type OCaml prints?

5

We want a function of type **int array** $\rightarrow$ **int**. Try a for loop with a reference to accumulate the sum.

6

Consider swapping elements from opposite ends of the array – the problem is symmetric.

7

To build an array of arrays, you will need a use `Array.make` to build an array of empty arrays. You can then set each of the elements of the main array to a suitably sized array, again created with `Array.make`. Once the structure is in place, putting the numbers in should be simple.

8

What is the difference between the codes for `'a'` and `'A'`? What about `'z'` and `'Z'`?

Chapter 14
The Other Numbers

1

Consider the built-in functions `ceil` and `floor`.

2

This is simple arithmetic. The function will take two points and return another, so it will have type **float** $\times$ **float** $\rightarrow$ **float** $\times$ **float** $\rightarrow$ **float** $\times$ **float**.

3

Consider the built-in function `floor`. What should happen in the case of a negative number?

4

Calculate the column number for the asterisk carefully. How can it be printed in the correct column?

5

You will need to call the `star` function with an appropriate argument at points between the beginning and end of the range, as determined by the step.

Chapter 15
The OCaml Standard Library

1

You can assume `List.rev` which is tail-recursive.

2

You might use `List.map` here, together with `List.mem`

3

The `String.iter` function should help here.

4

Try `String.map` supplying a suitable function.

5

Consider `String.concat`.

6

Create a buffer, add all the strings to it in order, and then return its contents.

7

`String.sub` is useful here. You can compare strings with one another for equality, as with any other type.

Chapter 16
Building Bigger Programs

1

You will need to alter the `Textstat` module to calculate the histogram and allow it to be accessed through the module's interface. Then, alter the main program to retrieve and print the extra information.

2

You will need functions to read and write the lines. You can read the required input and output filenames from `Sys.argv`. What should we do in case of an error, e.g. a bad filename?

3

Consider doing something a very large number of times. You should avoid printing information to the screen, because the printing speed might dominate, and the differing computation speeds may be hard to notice.

4

Start with a function to search for a given string inside another. You might find some functions from the `String` module in the OCaml Standard Library to be useful, or you can write it from first principles. Once this is done, the rest is simple.