Coping with Errors

It is very hard to write even small programs correctly the first time. An unfortunate but inevitable part of programming is the location and fixing of mistakes. OCaml has a range of messages to help you with this process.

Here are descriptions of the common messages OCaml prints when a program cannot be accepted or when running it causes a problem (a so-called "run-time error"). We also describe warnings OCaml prints to alert the programmer to a program which, though it can be accepted for evaluation, might contain mistakes.

Errors

These are messages printed when an expression could not be accepted for evaluation, due to being malformed in some way. No evaluation is attempted. You must fix the expression and try again.

Syntax error

This error occurs when OCaml finds that the program text contains things which are not valid words (such as **`if`**, **`let`** etc.) or other basic parts of the language, or when they exist in invalid combinations – this is known as *syntax*. Check carefully and try again.

```
        OCaml

#1 +;;
Error: syntax error
```

OCaml has underlined where it thinks the error is. Since this error occurs for a wide range of different mistakes and problems, the underlining may not pinpoint the exact position of your mistake.

Unbound value ...

This error occurs when you have mentioned a name which has not been defined (technically "bound to a value"). This might happen if you have mistyped the name.

```
        OCaml

# x + 1;;
Error: Unbound value x
```

In our example x is not defined, so it has been underlined.

This expression has type ... but an expression was expected of type ...

You will see this error very frequently. It occurs when the expression's syntax is correct (i.e. it is made up of valid words and constructs), and OCaml has moved on to type-checking the expression prior to evaluation. If there is a problem with type-checking, OCaml shows you where a mismatch between the expected and actual type occurred.

```
        OCaml

# 1 + true;;
Error: This expression has type bool but an expression was expected of type
         int
```

In this example, OCaml is looking for an integer on the right hand side of the + operator, and finds something of type **bool** instead.

It is not always as easy to spot the real source of the problem, especially if the function is recursive. Nevertheless, a careful look at the program will often shine light on the problem – look at each function and its arguments, and try to find your mistake.

This function is applied to too many arguments

Exactly what it says. The function name is underlined.

```
        OCaml

# let f x = x + 1;;
val f : int -> int = <fun>
# f x y;;
Error: This function is applied to too many arguments;
maybe you forgot a `;'
```

The phrase "maybe you forgot a ';' " applies to imperative programs where accidently missing out a ';' between successive function applications might commonly lead to this error.

Unbound constructor ...

This occurs when a constructor name is used which is not defined.

```
        OCaml

# type t = Roof | Wall | Floor;;
type t = Roof | Wall | Floor
# Window;;
Error: Unbound constructor Window
```

OCaml knows it is a constructor name because it has an initial capital letter.

The constructor … expects … argument(s), but is applied here to … argument(s)

This error occurs when the wrong kind of data is given to a constructor for a type. It is just another type error, but we get a specialised message.

```
        OCaml

# type p = A of int | B of bool;;
type p = A of int | B of bool
# A;;
Error: The constructor A expects 1 argument(s),
       but is applied here to 0 argument(s)
```

Run-time Errors

In any programming language powerful enough to be of use, some errors cannot be detected before attempting evaluation of an expression (until "run-time"). The exception mechanism is for handling and recovering from these kinds of problems.

Stack overflow during evaluation (looping recursion?)

This occurs if the function builds up a working expression which is too big. This might occur if the function is never going to stop because of a programming error, or if the argument is just too big.

```
        OCaml

# let rec f x = 1 + f (x + 1);;
val f : int -> int = <fun>
# f 0;;
Stack overflow during evaluation (looping recursion?).
```

Find the cause of the unbounded recursion, and try again. If it is really not a mistake, rewrite the function to use an accumulating argument (and so, to be tail recursive).

Exception: Match_failure …

This occurs when a pattern match cannot find anything to match against. You would have been warned about this possibility when the program was originally entered. For example, if the following function `f` were defined as

```
   let f x = match x with 0 -> 1
```

then using the function with `1` as an argument would produce:

```
        OCaml
```

```
# f 1;;
Exception: Match_failure ("//toplevel//", 1, 10).
```

In this example, the match failure occurred in the top level (i.e. the interactive OCaml we are using), at line one, character ten.

Exception: ...

This is printed if an un-handled exception reaches OCaml.

```
        OCaml

# exception Exp of string;;
exception Exp of string
# raise (Exp "Failed");;
Exception: Exp "Failed".
```

This can occur for built-in exceptions like `Division_by_Zero` or `Not_found` or ones the user has defined like `Exp` above.

Warnings

Warnings do not stop an expression being accepted or evaluated. They are printed after an expression is accepted but before the expression is evaluated. Warnings are for occasions where OCaml is concerned you may have made a mistake, even though the expression is not actually malformed. You should check each new warning in a program carefully.

This pattern-matching is not exhaustive

This warning is printed when OCaml has determined that you have missed out one or more cases in a pattern match. This could result in a `Match_failure` exception being raised at run-time.

```
        OCaml

# let f x = match x with 0 -> 1;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val f : int -> int = <fun>
```

Helpfully, it is able to generate an example of something the pattern match does not cover, so this should give you a hint about what has been missed out. You may ignore the warning if you're sure that, for other reasons, this case can never occur.

This match case is unused

This occurs when two parts of the pattern match cover the same case. In this situation, the second one could never be reached, so it is almost certain the programmer has made a mistake.

OCaml

```
# let f x = match x with _ -> 1 | 0 -> 0;;
Warning 11: this match case is unused.
val f : int -> int = <fun>
```

In this case, the first case matches everything, so the second cannot ever match.

This expression should have type unit

Sometimes when writing imperative programs, we ignore the result of some side-effect-producing function. However, this can indicate a mistake.

OCaml

```
# f 1; 2;;
Warning 10: this expression should have type unit.
- : int = 2
```

It is better to use the built-in `ignore` function in these cases, to avoid this warning:

OCaml

```
# ignore (f 1); 2;;
- : int = 2
```

The ignore function has type $\alpha \to$ **unit**. It has no side-effect.