Appendix E
GHCi errors

This appendix examines some of the more common programming errors in Haskell, and shows the error messages to which they give rise in GHCi.

The programs we write all too often contain errors. On encountering an error, the system either halts, and gives an **error message**, or continues, but gives a **warning message** to tell us that something unusual has happened, which might signal that we have made an error. In this appendix, we look at a selection of the messages output by GHCi; we have chosen the messages which are both common and require some explanation; messages like

```
*** Exception: Prelude.head: empty list
```

are self-explanatory. The messages are classified into roughly distinct areas. Syntax errors show up malformed programs, while type errors show well-formed programs in which objects are used at the wrong types. In fact, an ill-formed expression can often show itself as a type error and not as a syntax error, so the boundaries are not clear.

Syntax errors

A Haskell system attempts to match the input we give to the syntax of the language. Commonly, when something goes wrong, we type something *unexpected*.

- Typing '2==3)' will provoke the error message

  ```
  <interactive>:1:4: parse error on input `)'
  ```

- If a part of a definition is missing, as in

  ```
  fun x
  fun 2 = 34
  ```

  we receive the message

  ```
  Errors.hs:3:0: Not in scope: `fun'
  Errors.hs:3:4: Not in scope: `x'
  ```

  The problem here is that the system tries to understand `fun` and `x` and these are not (yet) defined.

- A similar error results when the two lines are reversed, as in

```
fun 2 = 34
fun x
```

when the error message is

```
Errors.hs:4:4: Not in scope: 'x'
```

- The inclusion of a type definition in a `where` clause, like this

```
fun x = x+1
        where
          type MyInt = Int
```

is signalled by

```
Errors.hs:8:10: parse error on input 'type'
```

- The syntax of patterns is more restricted than the full expression syntax, and so we get error messages like

```
Errors.hs:6:5:
    Conflicting definitions for 'x'
    Bound at: Errors.hs:6:5
              Errors.hs:6:7
    In the definition of 'fun'
```

when we use the same variable more than once within a pattern, as in the definition

```
fun (x,x) = x+1
```

- In specifying constants, we can make errors: floating-point numbers can be too large, and characters specified by an out-of-range ASCII code: for example, on typing `'\9999999999999999'` as input we get

```
<interactive>:1:18:
    lexical error in string/character literal at character '\''
```

- Not every string can be used as a name; some words in Haskell are **keywords** or **reserved identifiers**, and will give an error if used as an identifier. The keywords are

```
case class data default deriving do else if import in infix
infixl infixr instance let module newtype of then type where
```

For example, including the definition

```
do (x,y) = x+1
```

gives this error message

```
Errors.hs:6:0: Parse error in pattern
```

and the definition

```
data (x,y) = x+1
```

gives this message

```
Errors.hs:6:13: Not a data constructor: 'x'
```

As you can see from the two examples, the error message in a case like this reflects the meaning of the keyword.

- The special identifiers `as`, `qualified` and `hiding` have special meanings in certain contexts but can be used as ordinary identifiers.
- The final restriction on names is that names of constructors and types must begin with a capital letter; nothing else can do so, and hence we get error messages like

```
Errors.hs:6:0: Not in scope: data constructor 'Montana'
```

if we try to define a function called `Montana`.

Type errors

In this section we look at various different type errors that we can provoke in GHCi.

- As we have seen in the body of the text, the main type error we meet is exemplified by the response to typing `'c' && True` to the GHCi prompt:

```
Couldn't match expected type 'Bool' against inferred type 'Char'
In the first argument of '(&&)', namely 'c'
In the expression: 'c' && True
In the definition of 'it': it = 'c' && True
```

which is provoked by using a `Char` where an `Bool` is expected.

- Other type errors, such as

```
True + 4
```

provoke the error message

```
No instance for (Num Bool)
  arising from a use of '+' at <interactive>:1:0-7
Possible fix: add an instance declaration for (Num Bool)
In the expression: True + 4
In the definition of 'it': it = True + 4
```

This comes from the class mechanism: the system attempts to make `Bool` an instance of the class `Num` of numeric types over which '+' is defined. The error results since there is no such instance declaration making `Bool` belong to the class `Num`.

- As we said before, we can get type errors from syntax errors. For example, writing `abs -2` instead of `abs (-2)` gives the error message

```
No instance for (Num (a -> a))
  arising from a use of '-' at <interactive>:1:0-5
Possible fix: add an instance declaration for (Num (a -> a))
In the expression: abs - 2
In the definition of 'it': it = abs - 2
```

  because it is parsed as 2 subtracted from `abs::a->a`, and the operator '`-`' expects something in the class `Num`, rather than a function of type `a->a`. Other common type errors come from confusing the roles of '`:`' and '`++`' as in `2++[2]` and `[2]:[2]`.

- We always give type declarations for our definitions; one advantage of this is to spot when our definition does not conform to its declared type. For example,

```
myCheck :: Char -> Bool
myCheck n = toEnum n == 6
```

  gives the error message

```
Couldn't match expected type 'Int' against inferred type 'Char'
In the first argument of 'toEnum', namely 'n'
In the first argument of '(==)', namely 'toEnum n'
In the expression: toEnum n == 6
```

  Without the type declaration the definition would be accepted, only to give an error (presumably) when it is used.

- A definition like

```
asc x y
  | x <= y  = x y
```

  will give this error:

```
Occurs check: cannot construct the infinite type: a = a -> t
Probable cause: 'x' is applied to too many arguments
In the expression: x y
In the definition of 'asc': asc x y | x <= y = x y
```

  The problem here is that `x` is *compared with* `y` in the guard (type `a`), but *applied to* `y` in the body (type `a->t`): the unification required here produces an infinite type, which is not allowed.

- A final error related to types is given by definitions like

```
type Fred = (Fred,Int)                                    (Fred)
```

  a **recursive** type synonym; these are signalled by

```
Cycle in type synonym declarations:
  Errors.hs:9:0-21: type Fred = (Fred, Int)
```

  The effect of `(Fred)` can be modelled by the algebraic type definition

```
data Fred = Node Fred Int
```

  which introduces the **constructor** `Node` to identify objects of this type.

Program errors

Once we have written a syntactically and type correct script, and asked for the value of an expression which is itself acceptable, other errors can be produced during the **evaluation** of the expression.

- The first class of errors comes from missing cases in definitions. If we have written a definition like

  ```
  bat [] = 45
  ```

  and applied it to `[34]` we get the response

  ```
  *** Exception: Errors.hs:11:0-10: Non-exhaustive patterns
                 in function bat
  ```

  which shows the point at which evaluation can go no further, since there is no case in the definition of `bat` to cover a non-empty list. Similar errors come from built-in functions, such as `head`.
- Other errors happen because an **arithmetical constraint** has been broken. These include an out-of-range list index, division by zero, using a fraction where an integer is expected and floating-point calculations which go out of range; the error messages all have the same form. For example, suppose that we evaluate

  ```
  3 `div` 0
  ```

  then the error is

  ```
  *** Exception: divide by zero
  ```

- If we make a conformal definition, like

  ```
  [a,b] = [1 .. 10]
  ```

  this will fail with the message

  ```
  *** Exception: Errors.hs:13:0-16: Irrefutable pattern failed
                 for pattern [a, b]
  ```

  when either `a` or `b` is evaluated.

Module errors

The `module` and `import` statements can provoke a variety of error messages: files may not be present, or may contain errors; names may be included more than once, or an alias on inclusion may cause a name clash. The error messages for these and other errors are self-explanatory.

System messages

In response to some commands and interrupts, the system generates messages, including

- `^C ... Interrupted`
  signalling the interruption of the current task by typing `Ctrl-C`.

- The message

```
<interactive>: memory allocation failed (requested 2097152 bytes)
```

  which shows that the space consumption of the evaluation exceeds that available. One way around this is to increase the size of the heap.

A measure of the space complexity of a function, as described in Chapter 20, is given by the size of the smallest heap in which the evaluation can take place; how this 'residency' is measured is described in that chapter.