COMPILATION ERRORS, WARNINGS, AND BUGS

JEFFREY HE

1. Introduction

This guide overviews how to tackle compilation errors, compilation warnings, and bugs in OCaml. What are errors, warnings, and bugs?

- A compilation error occurs when code doesn't compile or doesn't run.
- A compilation warning occurs when code can compile, but may present issues when you run the code.
- A bug occurs when the code compiles but doesn't work as expected.

In this directory, you will find two files: `errors.ml` and `example.ml`. `errors.ml` provides practice fixing compilation errors and warnings, while `example.ml` will be used to illustrate testing.

2. Compilation Errors and Warnings

To fix a compilation error or warning, start by following the general steps outlined below. We include some of the more common errors and warnings to get you acquainted with their format. But when you encounter a different error, begin by carefully reading the error message and searching for documentation online before posting to Piazza or emailing a TF. To practice, open up `errors.ml`, a file with some (intentional!) compilation errors. By the end of this guide, you should be equipped to resolve all of the errors and warnings in `errors.ml`. To complete this exercise:

(1) Open up a terminal.
(2) `cd` to the correct directory.
(3) Run `ocamlbuild errors.byte`.
(4) Read the error message, taking note of the line and character numbers.
(5) Fix the error (see below or search online).
(6) Repeat until the code compiles and gives no warnings.

3. Common Compilation Errors

3.1. **Type Mismatch.** A type mismatch occurs when a value is used in a manner inconsistent with its type, such as passing a function an input of the wrong type, or returning a value of the wrong type.

Date: February 1, 2018.

3.1.1. *Floating Arithmetic.* One common mistake is using integer operations for floating arithmetic. For example, to add 3 and 4, we use the + operator. However, to add 3.0 and 4.0, we must use the +. operator. Finally, to add 3 and 4.0, we must convert either the 3 to a float or the 4.0 to an integer.

This is because OCaml is a strongly typed language, which means that values of one type cannot be used as though they were values of another type. This eliminates certain kinds of bugs at runtime, such as adding together the string "1" and the integer 1, which evaluates to the string "11" in JavaScript (a weakly typed language).

```
# 3.0 + 4.0;;
Error: This expression has type float but an expression was expected of type
         int
```

3.1.2. *Option Types.* Another common error occurs when working with option types. A variable with the type `int option` can have the value `None` or `Some 3`, but not `3`. `3` has type `int`, not `int option`. To fix this, prefix integer values with `Some` to make them option values.

```
# let get_first (lst: int list) : int option =
    match lst with
    | [] -> None
    | head :: _ -> head;;
Error: This expression has type int but an expression was expected of type
         int option
```

3.2. **Unbound Value.** This error occurs when an identifier is used before its definition, so the compiler cannot tell what the identifier refers to.

3.2.1. *Value hasn't been defined.* Look in the error message to determine which value is unbound.

```
# let y = 3 + x;;
Error: Unbound value x
```

3.2.2. *Rec Flag.* When writing a recursive function, it is easy to forget the `rec` flag that allows a function to be called within its own body. When you omit it, the compiler thinks that the function just hasn't been defined, and hence will throw an unbounded value error. To fix this error, simply add in 'rec' after 'let' in the function signature!

```
# let fib x = if x <= 1 then 1 else fib (x - 1) + fib (x - 2);;
Error: Unbound value fib
```

4. Common Warnings

While warnings do not prevent your code from compiling, they can alert you to potential bugs or bad style. Therefore, it is worthwhile to address all compiler warnings.

4.1. **Pattern match not exhaustive.** If you write a match statement that doesn't match some values, you'll get this warning. When you miss a case in the pattern match, you can still compile the code, but you run the risk of encountering a fatal `match_failure` error. The compiler will give you an example of what case(s) you missed, so you can go back to the code and include the case that's missing.

```
# let rec increment_lst (lst : int list) : int list =
    match lst with
    | head :: tail -> (head + 1) :: (increment_lst tail);;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val increment_lst : int list -> int list = <fun>
```

4.2. **Unused variable.** The compiler will throw an unused variable warning if you define a variable but never use it. This is considered bad style as it is misleading to define a value and never use it. Other programmers (or you reading your own code in the future) may be confused and wonder whether you intended to use the value somewhere in your code.

The fix for this warning depends on its cause. You may have a `let` statement in your code that you don't need – simply delete it. In other cases, you may define a variable in a pattern match but never use it. To fix this, you should replace your variable name with an underscore. The meaning of the underscore is unintuitive, you can optionally add text after the underscore to make things clear to a potential reader of your code. For example, you could write _head. With this underscore, you can't call the variable again.

```
Warning 27:  unused variable y.
```

5. Bugs and Testing

To solve bugs, begin by trying to explain your code to a friend (or a rubber duck, or a wall). Speaking about your code can be surprisingly useful. But ultimately, to avoid bugs and ensure code correctness, you should write tests. We've provided the file `example.ml` to illustrate how testing works. In `functions_tests.ml`, we present three methods of testing.

5.1. **Method 1: Evaluating assertions (line by line).** See the Method 1 section of `example_tests.ml`. The most convenient way to run these tests is to execute the file.

(1) Open up a terminal and `cd` to the directory containing `example_tests.ml`.
(2) Run `ocamlbuild example_tests.byte`.
(3) Run `./example_tests.byte`.

5.2. **Method 2: Aggregating assertions into a testing function.** See the Method 2 section of example_tests.ml. This method is similar to the first method, but it groups the assertions together into a function. One way to run to run these tests is to execute the file as in the first method. If you do this, make sure to actually call the function in the file (currently commented out).

Alternatively, you can run these tests in an OCaml interpreter (`utop` or `ocaml`), as you now have a function that you can call. This gives you the ability to run specific groups of tests as you wish.

(1) Open up a terminal and `cd` to the directory containing `example_tests.ml`.
(2) Open `utop` (or `ocaml`).
(3) Run `#mod_use "example.ml";;` (because we call functions in our original example.ml file, we must import this module).
(4) Run `#mod_use "cS51.ml";;` (because we call functions in the cS51.ml file, we must import this module as well).
(5) Run `#use "example_tests.ml";;` (this is the file that we ultimately want to use in `utop`, normally you skip directly to this step but because `example_tests.ml` depends on example.ml and cS51.ml, we have to run the previous two steps first).
(6) Run `test_last ();;`.
(7) Run `test_increment_lst ();;`.

Note: Remember to include the () in steps 6 and 7! This is necessary because the functions are of type `unit -> unit`; they need the input of unit to actually run the tests.

5.3. **Method 3: Using the CS51 "verify" function.** See the Method 3 section of example_tests.ml. For the final method, we utilize a helper function from the CS51 module that works similarly to `assert`, but without throwing an exception on failure. Instead, if the given test evalutes to false, it prints a user-supplied message using Printf.printf. If the test evalutes to true, it silently returns (). This way, you can continue to run all your tests even if one fails. You can also still group these tests together into functions, if you want to. In the example below, the "alphabet list" test case is written incorrectly, to illustrate how testing continues after a failure.

To run these tests, you can compile and execute the file as in the first method.

6. Recap of Guide

6.1. **Common Mistakes.**

- Forgetting the `rec` flag for a recursive function.
- Forgetting to define a variable before using it.
- Forgetting to use the float operation when dealing with floats (`+` versus `+.`).
- Forgetting Some when dealing with option types.
- Forgetting to open files when using functions from other modules.
- Forgetting any match cases.

6.2. **MiscellaneousOther Mistakes.**

- Running `ocamlbuild example.ml` instead of `ocamlbuild example.byte`.
- Using commas instead of semicolons in lists when testing.
- Forgetting to add two semicolons at the end.
- Being in the wrong directory when trying to run commands.