Unit 5c: Control Abstractions

1 Introduction

Most high level programming languages support a range of control abstractions.

Abstraction involves hiding complexity – control abstractions allow client code to make use of complex functionality without needing to "know" all the details of how it works.

Subroutines/functions abstract (possibly complex) sequences of functionality and can be called by client code using their interface (name and parameters).

Control structures abstract the details of how flow of execution works at the machine language level and allow these to controlled by named structures with convenient syntax.

For example repetition of instructions is abstracted into loop constructs (*for*, while) which are easy to program, and to understand when reading code.

Other common control abstractions built into many languages include *if-else*, *switch*, *try-catch*.

HO functions provide extensive scope for abstracting computations into functions and composing these

Curried functions make it possible to define control structures that look like they are part of the language, but are in fact just functions.

2 .curried

If we want to call a function that is not curried as if it is curried, for example, if we want to use it as a control abstraction, and we don't want to, or can't, rewrite the function itself, we can create a curried version using .curried.

For example, instead of rewriting filter we can do this:

```
val filter_curried = (filter _).curried
```

The function can then be called in curried style:

```
filter_curried(List.apply(1,2,3,4,5,6)){
  (x:Int) => x > 3
}
```

In some functional languages, such as Haskell, all functions are curried by default.