Foundations of Computer Science

Glossary

1. Introduction to programming

Programming The process of designing and building computer programs to solve a particular problem. A program is a sequence of instructions that is executed by the computer hardware, accomplishing a variety of tasks, such as data processing, input/output, computation, generation, etc.

Abstraction barrier The abstract (or sometimes concrete) barrier between adjacent program components, or different levels of abstraction.

Functions In a functional programming sense, a function is a piece of code that takes one or more inputs and returns a single value. *Pure* functions do not perform any side effects and their return value is solely dependent on the input – they are an approximation of mathematical functions. *Impure* functions may perform side effects in addition to (or instead of) returning values: while impure functions are required to write any kind of practical program that involves user interaction, their behaviour is much harder to predict and reason about.

Functional programming A programming paradigm that treats functions as the fundamental building block of programs. Functional programs are often evaluated expressions instead of executed statements, and their mathematical nature means that purity is a common desired property. The style is also often *declarative*: programs are written in a higher level of abstraction, expressing *what* the computer should do, not *how* to do a task.

Object-oriented programming A programming paradigm that treats objects as the fundamental building block of programs. Objects encapsulate the state and behaviour of concrete entities that we model in our code, and programs are based around the interaction of various objects. OOP style is usually *imperative*: programs are written as sequences of instructions that are executed in a linear manner, at a lower level of abstraction than functional programs.

Persistence If a piece of data (e.g. a number or a string) "outlives" the program that created it, we refer to it as persistent data. This includes writing values to memory, saving files to persistent storage, and many forms of interaction. Imperative programs heavily rely on persistent data, such as variables and in-memory data structures. Pure functional programs discourage the us e of persistence (and mutability) in favour of program safety: if data cannot be saved, it cannot be meddled with accidentally (or with malicious intent).

Recursion The standard way of achieving self-reference in mathematics, and repetition in functional programs. A function (or value, or data type) is recursive if its definition refers to itself. Standard examples are mathematical functions such as factorial or differentiation (e.g. the product rule), various list processing operations in functional programming, and recursive data structures such as lists and trees.

Iteration Iteration is the standard way of achieving repetition in imperative programs: it refers to the repeated execution of a sequence of instructions until some condition is met. Constructs such as `for`- and `while`-loops are the usual way of achieving iteration.

Overloading Using the same name or symbol for several (related) operations. The correct implementation is usually determined from the context or other clues (such as types). Examples in OCaml are the overloaded mathematical operators such as `+` and `*` that work on integers and real numbers.

Data types A means of structuring data (e.g. binary data) in a meaningful way that indicates the computer how we intend to use the particular piece of data – for example, whether a sequence of bytes should be treated as a character or an integer. Data types are also very useful for the programmer to represent and model real-world data, and get notified by the compiler if some value is not used in an intended way.

Type inference Deducing the type of a piece of data based on its structure or contextual clues. For example, in a piece of code such as `x *. 2.0`, the OCaml compiler can infer the type of `x` to be `float`, as it is being multiplied by another floating point number (using floating point multiplication).

Type constraints Sometimes the type inference cannot deduce the type of a value due to ambiguity. In that case, the programmer needs to add explicit type constraints to the code to guide the typechecker.

Numeric types Types that represent numbers, such as `int`, `float`, etc.

Boolean types Types that represent truth values, usually `bool`.

Declarations Stating the name (and type) of a value or a function.

Definitions Giving an implementation for a value or a function. Definitions and declarations are often performed together.

2. Recursion and efficiency

Expressions A recursive, tree-like syntactic structure consisting of constants, variables and operations. An expression can itself contain subexpressions which are combined with some operator, which gives expressions their hierarchical nature. This is in contrast with *statements* in imperative programming, which are sequenced in a mostly linear way.

Evaluation Reducing an expression to a single value, e.g. a mathematical expression to a real number. The evaluation usually proceeds by recursively evaluating the subexpressions, then applying the top-level operator to the values (e.g. $(5+1)*(4+3)$ reduces first to $6*7$ by evaluating the two factors, then 42 by applying the operator). This is in contrast with *execution* in imperative programming, which performs the side-effects of the linear statements in order.

Values The irreducible form of an expression: evaluation maps an expression to such a value.

Tail-recursion A way to simulate iteration in a functional language: instead of creating a lot of stack

frames with nested recursive calls (which could result in a stack overflow), we reuse the same stack frame and move changing data into accumulator arguments.

Asymptotic complexity A (very rough) estimate of the runtime or space usage of a program as a function of the input size. It ignores constant terms and multipliers and only takes the most significant term into account. It can be used to compare program efficiency in a fully algorithmic, machine-independent way.

***O*-notation** A notation for expressing the asymptotic space- or time-complexity of an algorithm, usually written as $O(f(n))$ where n is the input size and f is the complexity function (such as n^2 for quadratic algorithms).

Recurrence relations A technique for expressing and calculating the asymptotic complexity of recursive algorithms. We declare an estimate for the time/space usage of a recursive function by "induction" on the input, i.e. giving a separate expression for the base case(s) and another recursive expression for the recursive case. Solving the recurrence relation gives us a closed-form expression for the runtime estimate, which can then be rewritten in *O*-notation.

3. Lists

Lists The basic collection data type used in functional programming. A list of values can either be empty, or consist of a head element and a tail list. Lists can have arbitrary length but must be homogeneous, i.e. have elements of the same type.

Tuples Data types for "packaging" multiple values into a single value. For example, a tuple `(5, 'c')` of type `int * char` contains both an integer value `5`, and a character `'c'`. Tuples can contain values of different types, but their size is fixed.

Pattern-matching Analysing the structure of a value of some data type to extract the arguments to its constructors and/or provide different behaviour based on the shape of the value. For example, we can pattern-match on a list argument to provide different return expressions for when the list is empty and when it has a head and a tail.

Polymorphism Writing functions that can act on values of any type (or more precisely, functions whose type can contain an universally quantified type variable). Polymorphism lets us write *generic* code instead of giving several (or potentially infinite number of) declarations for each possible type that the function should handle. For example, many list processing operations are polymorphic in the type of the list elements, as e.g. a list reversal operation would be defined the same way for lists of integers, characters, or any arbitrary type.

Characters A single textual symbol such as a letter, number, punctuation mark, or any other ASCII character. Character literals in OCaml are written as `"a"`.

Strings A sequence of textual symbols – can be seen as just a list of characters (which is how strings are implemented in some languages). For efficiency reasons, OCaml implements strings as a primitive type.

4. More on lists

Wildcard pattern If we want to perform a pattern-match but don't actually care about one or more of the components of the pattern, we can avoid giving it a name by using `_`. For example, in the pattern for the `tail` function, we don't need to give the head element a name, so we can simply write the pattern as `(_::xs)`.

Equality polymorphism Polymorphism can be restricted to types whose values can be checked for equality. Most of the types we know support equality, the notable exception being functions. Whenever our function performs an equality check on a polymorphic argument (e.g. the `member` function), the inferred type variable will be an equality type. This means that the `member` function wouldn't work for functions.

Local declarations Assigning an expression or function to a name that can only be used within the body of a local declaration and not outside. They are used to giving a name to large expressions and declaring auxiliary functions that would otherwise pollute the namespace.

Stepwise refinement Making a function more efficient, smaller, faster, or simpler by refactoring and slightly changing it step-by-step. Instead of writing the "best" implementation right away, we can write one that works, and then use stepwise refinement to make it as good as possible.

List utilities Polymorphic functions that act on a list.

- `take`: Get the given number of elements from the beginning of the list.
- `drop`: Remove the given number of elements from the beginning of the list.
- `member`: Check whether an element appears in a list.
- `inter`: Intersection of lists (list of their common elements).
- `zip`: Convert a pair of lists into a list of pairs by matching up the elements pairwise and creating a tuple. If the two lists are of different length, the remaining elements of the longer list will be dropped.
- `unzip`: Convert a list of pairs into a pair of lists. The "opposite" of `zip`.

5. Sorting

Sorting algorithm An algorithm used to sort the elements of an ordered container in some order.

- **Insertion sort**: The head element of a list is inserted in the correct place in the list of sorted elements.
- **Quicksort**: The list is partitioned around a pivot element, creating a sublist of elements smaller than the pivot, and a sublist of elements greater than the pivot. The sublists are recursively sorted and combined with the pivot.
- **Mergesort**: The list is separated in two halves, which are recursively sorted then merged together maintaining the order.

Divide and conquer An algorithm design approach where the algorithm is (recursively) executed on small parts on the input first, which are then combined to give the final result. Examples are mergesort and quicksort.

Pivot The element selected in quicksort, around which the list is partitioned. Preferably the pivot should be an element in the middle of the list, but this cannot be enforced in general. After the two partitions are sorted, the pivot element will be the element between the two sorted sublists. One generally successful heuristic is to pick three (or five, or seven) elements from the list, and using the middle one as the pivot – this ensures that the pivot will never be the first or last element, but of course it doesn't guarantee a perfectly even split every time. To get an even split, we would need to sort the elements first!

Partitioning The step of the quicksort algorithm where the list is split into two sublists containing elements less than and greater than the pivot, respectively. Ideally the two partitions should be about the same size, but this depends on our choice of the pivot element.

6. Datatypes and trees

Datatype declarations An declaration that introduces a custom type into our program. The general syntax is

```
type <ty_vars> <ty_name> = <con1_name> of <con1_args>
                         | ...
                         | <conK_name> of <conK-args>
```

Constructors Special functions that return a value of our custom type. The type of constructor `<conK_name>` is `<conK_args> -> <ty_vars> <ty_name>`. Constructors can take any number of arguments as a tuple, but their return type will always be the type they belong to.

Enumeration types Types with nullary constructors (without arguments). Enumeration types effectively list all the possible values the type can take. Examples would be suits in a deck of cards, days of the week, etc.

Recursive datatypes Datatypes with a constructor that takes a value of the defined type as an argument. Standard examples are lists, whose tails are lists themselves, and trees, whose two subtrees are also trees.

- **Lists**: A recursive datatype that can either be an empty list (nullary constructor) or the cons (construction) of a list from a head (first element) and the tail (the rest of the list).
- **Binary trees**: A recursive datatype that can either be empty (nullary constructor) or a branch consisting of some value, a left subtree and a right subtree.

Exceptions A way of signalling the program or user of some error or unexpected control flow. Exceptions are often caused by missing resources, invalid input, etc. – they are generally different from *bugs* in that they are specifically declared and raised by the programmer who has considered the ways in which the program may fail. In practice, exceptions are values of a special `exception` type, and whenever we `raise` an exception in a function, it stops evaluation and propagates the exception to the calling function. Wrapping a piece of code in a `try` block lets us capture the error and gracefully resolve it; if that is not possible, the exception can propagate

to the top level of the program and cause a crash.

Comments Text in the source code of the program that is ignored by the compiler. Used to document and explain the code for human readers, such as specifying what a function does, what are its arguments, what are some exceptions that it may throw, etc. Documentation is a very important software engineering practice, as the code is written only once, but can be read dozens of times – without documentation, the reader has to go through the source code line-by-line to understand what it is doing and debug or refactor it. In, comments are written like `(* this *)` – anything between the two parentheses is ignored.

7. Dictionaries and functional arrays

Abstract types Type declarations with associated operations. The implementation of the type is hidden/abstracted away, so the only way to interact with a value of an abstract type is with the use of the declared operations.

- **Dictionaries**: Containers of key-value pairs. Keys and values can have any type, but the keys have to be unique (i.e. no repeated keys in one dictionary). Examples are dictionaries of names and unique identifiers, college names and their founding dates, etc.
- **Arrays**: Homogeneous containers (elements of the same type) indexed by the natural numbers, with a constant-time subscripting operation to access any element in the array.

Concrete types Implementations of abstract types, explicitly defining the type and the associated operations. An abstract type can have many concrete implementations, differing in efficiency, space use, simplicity, etc.

- **Association lists**: A very simple implementation of the dictionary type, maintaining the key-value pairs as a list of tuples. The operations are straightforward list traversals, but they do not result in the most efficient implementation.
- **Binary search trees**: A general, efficient container for ordered elements, such as key-value pairs (ordered by the key). Elements are stored in a binary tree with the invariant that for any node n, the root of the left subtree is ordered before n, and the root of the right subtree is ordered after n.
- **Functional arrays**: An efficient implementation of the array type in a functional language without references (or first-class arrays). Elements are stored in a binary tree (/not/ binary search tree) following an indexing convention that keeps all even elements in the left subtree and all odd elements in the right subtree (which holds recursively, with multiples of the subscript). The subscripting operation then traverses the tree based on the binary representation of the index.

Traversal Visiting every element of a tree in some order (preorder, inorder, postorder). By accumulating the visited nodes in a different container, we can convert a binary tree into another type, such as a list of elements.

Lookup Finding a value based on its key.

Update, insert, delete Updating or deleting a key-value pair at some specific key. Insertion creates a new pair (or possibly overwrites the existing one if the key already appears in the dictionary).

Subscripting Lookup of an array element based on its index (location within the array).

8. Functions as values

Higher-order functions Functions that can take other functions as arguments. This is an example of *behaviour parameterisation*: while in *data parameterisation* we write a single function instead of several values (e.g. a function `length` instead of value declarations `let lengthOfEmpty = 0`, `let lengthOfSingleton = 1` etc. for each possible list), in behaviour parameterisation we write a single higher-order function instead of several functions (e.g. a function `map` instead of functions `let doubleListElems`, `let incrementListElems`, `let convertListElemsToString` etc. for each possible transformation or behaviour that can be applied to the elements of a list).

Currying Functions that return other functions as results. Currying allows us to write functions of multiple arguments in a more idiomatic and flexible manner: a curried function `f : 'a -> ('b -> 'c)` of two arguments takes the first argument (of type `'a`) and returns a new function that takes the second argument (of type `'b`) and returns the result (of type `'c`). In fact, the term *currying* refers to the transformation of a function of type `('a * 'b) -> 'c` to a function of type `'a -> 'b -> 'c` which can be traced back to a very abstract and general duality between functions from a pair (product) type and functions into a function type.

First-class functions Higher-order functions and currying are two independent concepts, but they often appear together in programming languages under the term *first-class functions* or *functions as values*. Essentially, functions are not some special construct, they are normal values just as numbers, characters and strings are. And since they are values, they can be given as arguments to functions and returned from functions as results, or even put into a list or any other generic collection.

Anonymous functions When working with higher-order functions, we often want to parameterise a very small, simple behaviour, such as incrementing or doubling a number. It would be overkill to define a top-level function for such simple tasks, which could pollute our namespace or cause name clashes. Instead, we can define the function *anonymously*, as a single expression that can be passed to a higher-order function as an argument. For example, the OCaml expression `fun x -> x * x` represents the squaring function that takes a single argument `x` and returns the expression `x * x`. This expression (which is, in fact, a value) can then be bound to a value declaration (i.e. the declaration `let square = fun x -> x * x` is the same as the declaration `let square x = x * x`) or passed as an argument, e.g. as `map (fun x -> x * x) [1;2;3]`.

Partial application Curried functions return a new function when given their first argument. This means that, unlike for multivariable functions of type `'a * 'b -> 'c`, we do not have to pass in all of the arguments at the same time – passing in only the first one (or more, if there are more arguments) just results in a smaller function of fewer variables. This is called *partially*

applying a curried function to an argument. For example, given a curried function `cons : 'a -> 'a list -> 'a list` that acts as the `::` operator in OCaml, we can transform it into a new single-variable function by passing in its first argument, e.g. `5`. Then, `cons 5 : int list -> int list` is a new function that conses the number `5` to the beginning of an input list. This technique is often used to specialise higher-order functions to some specific behaviour: for example, given a generic sorting function, we can partially apply it to an ordering to create a function that sorts in the given (e.g. increasing) order.

9. Sequences and laziness

Laziness In computer science, laziness refers to "computing only what is required": a lazy program does exactly the amount of processing needed to give a result, no more, no less. A clear advantage of this approach is that programs don't do unnecessary computation: if something is not required by the rest of the program, the value is not computed. However, laziness leads to unpredictable space and time usage, as the amount of computation required might only be known at runtime.

Delayed evaluation Laziness often comes hand-in-hand with delayed evaluation: a lazily evaluated expression is not evaluated at all, until its value is required by some other expression.

Forcing When a lazily evaluated expression *does* need to be evaluated, we force its evaluation. This might be implicit (in most lazy functional languages), or explicit (e.g. applying the tail of a sequence to `()` in OCaml). The aim of lazy evaluation is to force as late as possible, only when the program definitely needs the value of an expression (e.g. `5`) and not just the expression itself (e.g. `2 + 3`).

Lazy lists A common lazy data structure is a lazy list, which acts as a normal list but whose tail is actually lazily evaluated. This means that every list processing operation happens lazily, and evaluation actually happens only when we need to look at the list elements. For example, `take 2 (map isEven [1;2;3;4;5])` only applies `isEven` to the first two elements of the list, as `take` returns only these and drops the rest. As those elements will not be seen, the function application on them is not evaluated. This also means that lazy lists can represent infinite lists: the function `let l x = x : l x` would be a perfectly valid lazy list, as the recursive call `l x` is only evaluated if more than the first element are needed.

Sequences OCaml is an *eager* language, i.e. it does not use lazy evaluation (unlike e.g. Haskell). Hence, OCaml lists are all fully evaluated and kept in memory, which obviously forbids infinite lists. Sequences (at least in the wording of the notes) are a way to encode lazy lists in OCaml with the trick of making the tail of the list a function from the unit. As functions are values, the tail function is never evaluated by default, but we can *force* evaluation by applying it to `()`. This way, we encode the desirable properties of lazy lists into our strict, eager language.

10. Queues and search strategies

Abstract data type A data type with associated operations, such as a stack with operations for push, pop, peek, etc. An abstract data type can have several concrete implementations, differing in simplicity, efficiency, and so on – however, the implementation details are hidden since we only interact with the abstract type through its interface, not the internal representation.

Depth-first traversal Tree traversal that visits the subtrees of a node before its siblings. This is a more natural operation for recursive data structures like trees, and much easier to implement. However, it is unsuitable if the subtrees need to be visited in a "fair" way: if the left subtree is very deep, the traversal will not reach even the root of the right subtree for a long time. For example, if we are searching for a specific node in the tree, we might end up wasting a lot of time and resources trying to find it in a deep left subtree even though it is near the root of the right subtree.

Breadth-first traversal Tree traversal that visits the sibling nodes before the subtrees of a node. Not as natural to implement than depth-first search, but it eliminates the problem of "fairness" and is therefore much more suitable for search.

Queues Abstract collection data type that allows easy access to the "front" of the queue for element removal, and easy access to the "back" of the queue for element addition. Characteristic operations are `enqueue` for addition to the back, `dequeue` for removal from the front.

FIFO First-in-first-out strategy for queues: the first element to enter the queue will also be the first element to exit it.

Stacks Abstract collection data type that allows easy access to the "top" of the stack for both element addition and removal. Characteristic operations are `push` for addition to the top, `pop` for removal from the top.

LIFO Last-in-first-out strategy for stacks: the last element to get pushed on the stack will also be the first element to get popped off.

Amortised cost Asymptotic time/space complexity calculated over the "lifetime" of an algorithm or structure. Typically, if some operation has low cost most of the time and a high cost once in a while, the overall cost "averaged" over all operations will still be the same complexity as the cheap operations. In general, amortised cost analysis can be valuable if the frequency of an operation is inversely proportional to its cost. Examples are the OCaml queue operations: most of the time they are simple list head access, but sometimes the queue has to be normalised, which is linear. However, the larger the queue, the less frequently it has to be normalised, so overall these higher-cost operations can be ignored.

Iterative deepening A breadth-first search strategy that performs BFS repeatedly with increasing depth bounds, each time discarding the result of the previous search. This is not as inefficient as it may seem, as the number of nodes at level d is exponentially greater than the number of nodes at level $d-1$ (in fact, for full trees, it is greater than the total number of nodes up to level $d-1$). Iterative deepening is more space-efficient than normal depth-first search, as it

recomputes the nodes at a depth d rather than storing them, so the space usage is linear in the depth instead of exponential.

Case expression A general expression that can be used for pattern-matching: case-splitting on an expression `e` evaluates the expression, then allows us to pattern-match on the results and handle each of the possible cases. Given that match-expressions can split on any expression and not just an input, they are more general than pattern-matching on arguments. For example, the following function returns the head and tail of the (non-contiguous) sublist of even numbers in a list; note how the match-expression lets us pattern-match on the result of the call to `List.filter`.

```
let unconsEvens xs = match List.filter isEven xs with
                     | []      -> failwith "No evens"
                     | (e::es) -> (e,es)
```

11. Elements of procedural programming

State Persistently stored values that can be saved, retrieved and changed.

Procedural programming An imperative programming paradigm that places importance on procedure calls (also called functions, methods, subroutines, etc.) and sequential execution of statements. Procedural programs are generally based on manipulation of (possibly shared) state, instead of evaluating expressions.

Input/output Most common "side effects" in imperative languages: input (key presses from the user, temperature values from sensors, etc.) and output (text on the terminal, images on screen, sound, file output, etc.) that enables the user to interact with the program.

References Special values that refer to (or point to) a memory location that stores another value. In imperative languages we can change the stored value by assigning to a reference, so there is no guarantee of immutability (unlike in pure functional languages).

Dereferencing Getting the value from memory that a reference is pointing to. Can be implicit (in most imperative languages) or explicit (the `!` operator in OCaml).

Assignment Putting a value into the memory location that a reference points to. Usually done with an assignment operator such as `=` in imperative languages, or `:=` in OCaml.

Commands, statements The instruction that the program has to execute one after the other. Commands are often executed purely for the side-effects they produce (e.g. assignment or printing), so the value of an assignment is often meaningless (such as `unit`). Commands and statements are usually sequenced with the `;` operator, which denotes that the first command must be executed before the second – this gives imperative programs their linear execution structure.

While loops The basic imperative iteration structure: execution of a block of commands as long as some Boolean condition holds. While loops only make sense in a stateful context, because the Boolean condition would never change otherwise: if there is no state at play, the condition

would have to evaluate to the same value every time. While loops can be used to implement other iteration constructs such as do-while loops, do-until loops, for loops and foreach loops.

Arrays The basic imperative data structure: a contiguous sequence of memory locations that store a collection of values. Arrays allow constant-time access to all of its elements, but they have a known, fixed length (as a sequence of locations must be "reserved" beforehand).