# Lecture Notes: An Introduction to Computing[1]

Subhashis Banerjee S. Arun-Kumar

*Department of Computer Science and Engineering*
*Indian Institute of Technology Delhi*
*New Delhi, 110016*
*email: {suban,sak}@cse.iitd.ac.in*

March 21, 2022

# Contents

# Part I

# Models of computation

# Chapter 1

# Introduction

This course is about *computing*. The notion of computing is much more fundamental than the notion of a computer, because computing can be done even without one. In fact, we have been computing ever since we entered primary school, mainly using pencil and paper. Since then, we have been *adding, subtracting, multiplying, dividing*, computing *lengths, areas, volumes* and many many other things. In all these computations we follow some definite, unambiguous set of rules. This course is about *studying these rules* for a variety of problems and writing them down explicitly.

When we explicitly write down the rules (or instructions) for solving a given computing problem, we call it an *algorithm.* Thus algorithms are primarily vehicles for *communication*; for specifying solutions to computational problems, unambiguously, so that others (or even computers) can understand the solutions. When an algorithm is written according to a particular syntax of a language which can be interpreted by a digital computer, we call it a *program.* This last step is necessary when we wish to carry out our computations using a computer.

While writing down algorithms, it is important to choose an underlying model of computation, i.e., to choose appropriate primitives to describe an algorithm. This choice determines the kind of computations that can be carried out in the model. For example, if our computational model consists of only ruler and compass constructions, then we can write down explicit rules (algorithms) for bisecting a line segment, bisecting an angle, constructing lengths equal to given irrational numbers and a plethora of other things. We cannot, however, trisect an angle. For trisecting an angle, we require additional primitives like, for example, a protractor. For arithmetic computations we can use various computing models like calculators, slide rules or even an abacus (as we believe our ancestors have been using). With each of these models of computing, the rules for specifying a solution (algorithms) are different, and the precision of the solution also differs. Thus, in our study of *algorithms* and *programs*, it becomes important to first choose a reasonable model of computation. Soon in these notes we will describe our choice of computational models which are widely used in modern computing using digital computers.

Once a computational model is available, and we can specify an algorithm (or a program)

to solve a given problem, we have to ensure that the algorithm is *correct*. We would also wish that our algorithms are *efficient*. Correctness and efficiency of algorithm design are central issues in this course. In what follows, we will endeavour to develop methodologies for the design of correct algorithms.

Thus, the major vehicles for problem solving using computers are:

**Algorithm** An *Algorithm* is a finite specification of the solution to a given problem (definite input and output). It is unambiguous and specifies a solution in terms of a *finite* process (finite number of steps in execution).

**Effective Algorithm** When the solution is specified according to a model of computation (particular primitives).

**Program** Encoding of an effective algorithm in the syntax of a programming language. This is necessary for machine execution of an algorithm.

In the two subsequent chapters we will establish two widely used models of computation and develop methodologies for algorithm design and proving correctness of algorithms in these models. The two models that we will introduce are the *functional* and the *imperative* models of computation. We will use the programming languages `ML` and `Java` to write programs in these two models respectively. We will devote the remaining part of this chapter to discuss some mathematical preliminaries necessary for programming.

# Chapter 2

# Mathematical preliminaries

We will discuss *sets, relations* and *functions, Boolean logic* and *Principle of Mathematical Induction*. Most of these topics are covered in the high school curriculum and a confident reader may wish to skip this section. However, we urge the reader to definitely read the material on *Mathematical Induction* which forms the basis for programming.

## 2.1 Sets

A *set* is a collection of *distinct* objects. The class of CS120 is a set. So is the group of all first year students at the IITD. We will use the notation $\{a, b, c\}$ to denote the collection of the objects $a, b$ and $c$. The elements in a set are not ordered in any fashion. Thus the set $\{a, b, c\}$ is the same as the set $\{b, a, c\}$. Two sets are *equal* if they contain exactly the same elements.

We can describe a set either by enumerating all the elements of the set or by stating the properties that uniquely characterize the elements of the set. Thus, the set of all even positive integers not larger than 10 can be described either as $S = \{2, 4, 6, 8, 10\}$ or, equivalently, as $S = \{x \mid x \text{ is an even positive integer not larger than } 10\}$

A set can have another set as one of its elements. For example, the set $A = \{\{a, b, c\}, d\}$ contains two elements $\{a, b, c\}$ and $d$; and the first element is itself a set.

We will use the notation $x \in S$ to denote that $x$ is an element of ("belongs to") the set $S$.

A set $A$ is a *subset* of another set $B$, denoted as $A \subseteq B$, if $x \in B$ whenever $x \in A$.

An *empty set* is one which contains no elements and we will denote it with the symbol $\phi$. For example, let $S$ be the set of all students who fail the course CS120. $S$ might turn out to be empty (hopefully; if everybody studies hard). By definition, the empty set $\phi$ is a subset of all sets. We will also assume an *Universe* of discourse $\mathcal{U}$, and every set that we will consider is a subset of $\mathcal{U}$. Thus we have

1. $\phi \subseteq A$ for any set $A$
2. $A \subseteq \mathcal{U}$ for any set $A$

The *union* of two sets $A$ and $B$, denoted $A \cup B$ is the set whose elements are exactly the elements of either $A$ or $B$ (or both). The *intersection* of two sets $A$ and $B$, denoted $A \cap B$ is the set whose elements are exactly the elements of both $A$ and $B$. Thus, we have

1. $S = A \cup B = \{x \mid (x \in A) \text{ or } (x \in B)\}$
2. $S = A \cap B = \{x \mid (x \in A) \text{ and } (x \in B)\}$

We also have, for any set $A$

1. $A \cup \phi = A$
2. $A \cup \mathcal{U} = \mathcal{U}$
3. $A \cap \phi = \phi$
4. $A \cap \mathcal{U} = A$

The *Cartesian product* of two sets $A$ and $B$, denoted by $A \times B$, is the set of all ordered pairs $(a, b)$ such that $a \in A$ and $b \in B$. Thus,

$$A \times B = \{(a, b) \mid (a \in A) \text{ and } (b \in B)\}$$

$A^n$ is the set of all ordered $n$-tuples $(a_1, a_2, \ldots, a_n)$ such that $a_i \in A$ for all $i$. i.e.,

$$A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$$

We will use the following notation to denote some standard sets:

**The set of Natural Numbers** [1] $\mathbb{N} = \{0, 1, 2, \ldots\}$

**The set of positive integers** $\mathbb{P} = \{1, 2, 3, \ldots\}$

**The set of integers** $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

**The set of real numbers** $\mathbb{R}$

**The Boolean set** $\mathbb{B} = \{false, true\}$

## 2.2 Relations and Functions

A *binary relation* from $A$ to $B$ is a subset of $A \times B$. It is a characterization of the intuitive notion that some of the elements of $A$ are related to some of the elements of $B$. Familiar binary relations from $\mathbb{N}$ to $\mathbb{N}$ are $=$, $\neq$, $<$, $\leq$, $>$, $\geq$. Thus the elements of the set $\{(0,0), (0,1), (0,2), \ldots, (1,1), (1,2), \ldots\}$ are all members of the relation $\leq$ which is a subset of $\mathbb{N} \times \mathbb{N}$.

[1] we will include 0 in the set of Natural numbers. After all, it is quite natural to score a 0 in an examination

In general, an *n-ary relation* among the sets $A_1, A_2, \ldots, A_n$ is a subset of the set $A_1 \times A_2 \times \cdots \times A_n$.

A *function* from $A$ to $B$ is a binary relation $R$ from $A$ to $B$ such that for *every* element $a \in A$ there is a *unique* element $b \in B$ so the $(a, b) \in R$ $(R(a) = b)$. We will use the notation $R : A \to B$ to denote a function $R$ from $A$ to $B$. The set $A$ is called the *domain* of the function $R$ and the set $B$ is called the *co-domain* of the function $R$. The *range* of a function $R : A \to B$ is the set $\{b \in B \mid \text{for some } a \in A,\ R(a) = b\}$. Some familiar examples of functions are

1. $+$ and $*$ (addition and multiplication) are functions of the type $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
2. $-$ (subtraction) is a function of the type $f : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$.
3. *div* and *mod* are functions of the type $f : \mathbb{N} \times \mathbb{P} \to \mathbb{N}$. If $a = q * b + r$ such that $0 \leq r < b$ and $a, b, q, r \in \mathbb{N}$ then the functions *div* and *mod* are defined as $div(a, b) = q$ and $mod(a, b) = r$. We will often write these binary functions as $a * b$, $a\ div\ b$, $a\ mod\ b$ etc.
4. The binary relations $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ are also functions of the type $f : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ where $\mathbb{B} = \{false, true\}$.

## 2.3 Principle of Mathematical Induction

Anyone who has had a good grounding in school mathematics must be familiar with two uses of mathematical induction.

1. *Definition* of functions and relations by mathematical induction,
2. *Proofs* by the principle of mathematical induction.

We present below some familiar examples of *definitions by mathematical induction.*

**Example 2.1** The factorial function on natural numbers (of the type $f : \mathbb{N} \to \mathbb{N}$) is defined as follows

**Basis.** $0! = 1$

**Induction step.** $(n + 1)! = (n + 1) * n!$

**Example 2.2** The $n^{th}$ power (where $n$ is a natural number) of a positive number $x$ is often defined as

**Basis.** $x^0 = 1$

**Induction step.** $x^{n+1} = x^n * x$

This is a function of the type $f : \mathbb{P} \times \mathbb{N} \to \mathbb{P}$.

**Example 2.3** The set of $n$-tuples of natural numbers can be defined in terms of Cartesian products as

**Basis.** $\mathbb{N}^1 = \mathbb{N}$

**Induction step.** $\mathbb{N}^{n+1} = \mathbb{N}^n \times \mathbb{N}$

**Example 2.4** For binary relations $R$ and $S$ on $A$ we define their composition (denoted $R \circ S$) as follows.

$$R \circ S = \{(a, c) \mid \text{for some } b \in A,\ (a, b) \in R \text{ and } (b, c) \in S\}$$

We may extend this binary relational composition to an $n$-fold composition of a single relation $R$ as follows.

**Basis.** $R^1 = R$

**Induction step.** $R^{n+1} = R \circ R^n$

Similarly the principle of mathematical induction is the means by which we have often *proved* (as opposed to *defining*) properties about numbers, or statements involving the natural numbers. The principle may be stated as follows.

**Principle of Mathematical Induction (PMI) – Version 1**
A property **P** holds for all natural numbers provided

**Basis step:** **P** holds for 0, and

**Induction step:** If **P** holds for an arbitrary $n \geq 0$, **P** also holds for $n + 1$

The underlined portion, called the **Induction hypothesis**, is an assumption that is necessary for the conclusion to be proved. Intuitively, the principle captures the fact that in order to prove any statement involving natural numbers, it suffices to do it in two steps. The first step is the basis which needs to be proved. The proof of the induction step essentially tells us that the reasoning involved in proving the statement involving the other natural numbers is the same once the basis has been proved. Hence instead of an infinitary proof (one for each natural number) we have a compact finitary proof which exploits the similarity of the proofs for all the naturals except the basis.

**Example 2.5** We prove that all natural numbers of the form $n^3 + 2n$ are divisible by 3.
*Proof:*

**Basis.** For $n = 0$, we have $n^3 + 2n = 0$ which is divisible by 3.

**Induction hypothesis.** $n^3 + 2n$ is divisible by 3 for an arbitrary $n \geq 0$.

**Induction step.** Consider $(n+1)^3 + 2(n+1)$. We have

$$\begin{aligned} & (n+1)^3 + 2(n+1) \\ = \; & (n^3 + 3n^2 + 3n + 1) + (2n + 2) \\ = \; & (n^3 + 2n) + 3(n^2 + n + 1) \end{aligned}$$

which is divisible by 3.

□

Several versions of this principle exist. We state some of the most important ones. In each case, the underlined portion is the induction hypothesis. For example it is not necessary to consider 0 (or even 1) as the basis step. Any integer $k$ could be considered the basis, as long as the property is to be proved for all $n \geq k$.

**Principle of Mathematical Induction (PMI) – Version 2**
A property **P** holds for all natural numbers, $n \geq k \geq 0$ provided

**Basis step:** **P** holds for $k$, and

**Induction step:** If **P** holds for an arbitrary $n \geq k$, then **P** holds for $n+1$.

Such a version seems very useful when the property to be proved is not true or is undefined for 0 or 1. The following example illustrates this.

**Example 2.6** Suppose we have stamps of two different denominations, 3 paise and 5 paise. We want to show that it is possible to make up exactly any postage of 8 paise or more using stamps of these two denominations [Liu]. Thus we want to show that every positive integer $n \geq 8$ is expressible as $n = 3i + 5j$ where $i, j \geq 0$.
*Proof:*

**Basis.** For $n = 8$, we have $n = 3 + 5$, i.e. $i = j = 1$.

**Induction hypothesis.** $n = 3i + 5j$ for an $n \geq 8$, $i, j \geq 0$.

**Induction step.** Consider $n+1$. If $j = 0$ then clearly $i \geq 3$ and we may write $n+1$ as $3(i-3) + 5(j+2)$. Otherwise $n + 1 = 3(i+2) + 5(j-1)$.

□

In the next version we strengthen the hypothesis.

**Principle of Mathematical Induction (PMI) – Version 3**
A property **P** holds for all natural numbers provided

**Basis step:** **P** holds for 0, and

**Induction step:** If **P** holds for all $m$, $0 < m \leq n$ for an arbitrary $n \geq 0$, then **P** also holds for $n+1$.

**Example 2.7** Let $F_0 = 0,\ F_1 = 1,\ F_2 = 1, \ldots$ be the Fibonacci sequence where for all $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. Let $\phi = (1 + \sqrt{5})/2$. We now show that $F_n \leq \phi^{n-1}$ for all positive $n$.
*Proof:*

**Basis.** For $n = 1$, we have $F_1 = \phi^0 = 1$.

**Induction hypothesis.** $F_n \leq \phi^{n-1}$ for all $m$, $1 \leq m \leq n$.

**Induction step.**

$$\begin{array}{rcll} F_{n+1} & = & F_n + F_{n-1} & \\ & \leq & \phi^{n-1} + \phi^{n-2} & \text{(by the induction hypothesis)} \\ & = & \phi^{n-2}(\phi + 1) & \\ & = & \phi^n & \text{(since } \phi^2 = \phi + 1\text{)} \end{array}$$

□

Versions 1 and 2 of **PMI** rely on the fact that starting from 0 (or $k$) every integer larger than 0 (or $k$) may be obtained by successively adding 1 to the previous one, whereas version 3 is obtained by considering the natural numbers as being totally ordered by the $<$ relation.

Since the natural numbers are themselves defined as the smallest set $\mathbb{N}$ such that $0 \in \mathbb{N}$ and whenever $n \in \mathbb{N}$, $n + 1$ also belongs to $\mathbb{N}$. Therefore we may state yet another version of **PMI** from which the other versions previously stated may be derived. The intuition behind this version is that a property **P** may also be considered as defining a set $S = \{x \mid x \text{ satisfies } \mathbf{P}\}$. Therefore if a property **P** is true for all natural numbers the set defined by the property is the set of natural numbers. This gives us the last version of **PMI**.

**Principle of Mathematical Induction (PMI) – Version 0**
For a set $S$, $S = \mathbb{N}$ provided

**Basis step:** $0 \in S$, and

**Induction step:** If $\underline{n \in S}$, then $n + 1 \in S$.

## 2.4 Elements of logic

Please read up about Boolean algebra and Propositional calculus.

## Problems

1. GCD of two integers $a, b > 0$ is defined as $\max\{x : x \text{ is an integer},\ x > 0,\ x \mid a,\ x \mid b\}$, where the notation $x \mid a$ means $x$ divides $a$. Consider the following algorithm for

computing GCD using pencil and paper:

$$gcd(a,b) = \begin{cases} a & \text{if } a = b \\ gcd(a-b,b) & \text{if } a > b \\ gcd(a,b-a) & \text{if } b > a \end{cases}$$

Convince yourself that the above algorithmic specification (rule) is correct for computing GCD. Carry out the pencil and paper computation using the above algorithm for the special case of $a = 18$ and $b = 12$.

2. Suppose your algorithmic primitives are *ruler and compass* constructions.
   (a) Explain how you can compare two integer lengths to determine which is larger.
   (b) Explain how you can subtract one integer from the other.
   (c) Give a set of rules (algorithm) for computing the GCD of two integers using ruler and compass.

3. Prove the following formulae by using **PMI**.
   (a) $1 + 2 + \cdots + n = n(n+1)/2$
   (b) $1 + 3 + 5 + \cdots + (2n-1) = n^2$
   (c) $1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2$

4. Note that

$$\begin{aligned} 1 + \frac{1}{2} &= 2 - \frac{1}{2} \\ 1 + \frac{1}{2} + \frac{1}{4} &= 2 - \frac{1}{4} \\ 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} &= 2 - \frac{1}{8} \end{aligned}$$

   Guess the general law suggested and prove it by using **PMI**.

5. Prove the following statement using **PMI**: If a line of unit length is given, then a line of length $\sqrt{n}$ can be constructed using *ruler and compass* for every positive integer $n$.

6. Prove by **PMI**, that every integer $n > 1$ is either a *prime* or a *product of primes*.

7. Prove that versions 1, 2 and 3 of **PMI** are mutually equivalent.

8. Find the fallacy in the following proof by **PMI**. Rectify it and again prove using **PMI**.
   **Theorem**

$$\frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \cdots + \frac{1}{(n-1) \times n} = \frac{3}{2} - \frac{1}{n}$$

*Proof:* For $n = 1$ the $LHS$ is $1/2$ and so is the $RHS$. Assume that the theorem is *true* for an $n > 1$. We then prove the induction step.

$$\begin{aligned} LHS &= \frac{1}{1\times 2} + \frac{1}{2\times 3} + \cdots + \frac{1}{(n-1)\times n} + \frac{1}{n\times(n+1)} \\ &= \frac{3}{2} - \frac{1}{n} + \frac{1}{n\times(n+1)} \\ &= \frac{3}{2} - \frac{1}{n} + \frac{1}{n} - \frac{1}{n+1} \\ &= \frac{3}{2} - \frac{1}{n+1} \end{aligned}$$

□

9. Find the fallacy in the following proof by **PMI**.
**Theorem** Given any collection of $n$ blonde girls. If at least one of the girls has blue eyes, then all $n$ of them have blue eyes.
*Proof:* The statement is obviously true for $n = 1$. The step from $k$ to $k + 1$ can be illustrated by going from $n = 3$ to $n = 4$. Assume, therefore, that the statement is true for $n = 3$ and let $G_1, G_2, G_3, G_4$ be four blonde girls, at least one of which, say $G_1$, has blue eyes. Taking $G_1, G_2$, and $G_3$ together and using the fact that the statement is true when $n = 3$, we find that $G_2$ and $G_3$ also have blue eyes. Repeating the process with $G_1, G_2$ and $G_4$, we find that $G_4$ has blue eyes. Thus all four have blue eyes. A similar argument allows us to make the step from $k$ to $k + 1$ in general. □
**Corollary.** All blonde girls have blue eyes.
*Proof:* Since there exists at least one blonde girl with blue eyes, we can apply the foregoing result to the collection consisting of all blonde girls. □
*Note:* This example is from G. Pólya, who suggests that the reader may want to test the validity of the statement by experiment.

# Chapter 3

# A functional model of computation

In this chapter we will introduce the basics of a functional model of computation. The functional model is very close to mathematics; hence functional algorithms are easy to analyze in terms of correctness and efficiency. We will use the `ML` interactive environment to write and execute functional programs. This will provide an interactive mode of development and testing of our early algorithms. In the later chapters we will see how a functional algorithm can serve as a specification for development of algorithms in other models of computation.

In the functional model of computation every problem is viewed as an evaluation of a function. The solution to a given problem is specified by a complete and unambiguous functional description. Every reasonable model of computation must have the following facilities:

**Primitive expressions** which represent the simplest objects with which the model is concerned.

**Methods of combination** which specify how the primitive expressions can be combined with one another to obtain compound expressions.

**Methods of abstraction** which specify how compound objects can be named and manipulated as units.

In what follows we introduce the following features of our functional model:

1. The *Primitive* expressions.
2. Definition of one function in terms of another (substitution).
3. Definition of functions using conditionals.
4. Inductive definition of functions.

We will introduce the more advanced concepts in our functional model later in these notes.

## 3.1 The primitive expressions

The basic primitives of the functional model are *constants*, *variables* and *functions*.

Elements of the sets $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ are *constants*. Thus, numbers like -1, 0, 1, 5.26 are all constants. So are the elements of the set $\mathbb{B} = \{true, false\}$. We will introduce other kind of constants later in these notes. *Variables* are identifiers which refer to data objects (constants). We will use identifiers like $n$, $a$, $b$, $x$ etc. to refer to various data elements used in our functional algorithms. The variables are *bound* to values (constants) in pretty much the same way as in school algebra. Thus, the declarations $x = 5$ and $y = true$ bind the variables $x$ and $y$ to the values 5 and *true* respectively.

In what follows we describe an interactive session in the `ML` programming environment where we bind variables to constants.

Typing `val x = 5;` at the prompt `-` of the `ML` environment defines `x` to be 5. The `ML` interpreter returns `val x = 5 : int` and waits at the next `-` prompt.

```
Standard ML of New Jersey, Version 110.0.3, January 30, 1998
- val x = 5;
val x = 5 : int
-
```

If we now type `x;` at the `ML` prompt the interpreter returns `val it = 5 : int`. The word `it` is an `ML` keyword indicating the the value of the last expression that it has evaluated. Further, note that `ML` also informs you of the type of the value that it has returned. The word `int` is used to denote an integer.

```
- x;
val it = 5 : int
```

Similarly we can bind the variable `y` to the boolean constant `true` (*true*) as follows:

```
- val y = true;
val y = true : bool
-
```

Typing `y;` next gives us `val it = true : bool`.

```
- y;
val it = true : bool
-
```

The primitive functions of the type $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ which we assume to be available in our functional model are addition $(+)$, subtraction $(-)$, multiplication $(*)$. We will also assume the *div* and *mod* functions of the type $f : \mathbb{N} \times \mathbb{P} \to \mathbb{N}$. Note that if $a \in \mathbb{N}$ and $b \in \mathbb{P}$ and $a = q * b + r$ for some integers $q \in \mathbb{N}$ and $0 \leq r < b$ then $div(a, b) = q$ and $mod(a, b) = r$. The division function $/ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ will be assumed to be valid only for real numbers. In addition to the above, we will assume the functions $=$, $\leq$, $<$, $\geq$, $>$

and $\neq$ which are of the type $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ or $f : \mathbb{R} \times \mathbb{R} \to \mathbb{B}$ depending on the context; and the functions $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (and), $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (or) and $\neg : \mathbb{B} \to \mathbb{B}$ (not). These functions can be invoked in `ML` in the following way.

⟨*Example*⟩≡

```
- 5 * 2;
val it = 10 : int
- 5.0/2.0;
val it = 2.5 : real
- 5 - 2;
val it = 3 : int
- val a = true;
val a = true : bool
- val b = false;
val b = false : bool
- not a;
val it = false : bool
- a andalso b;
val it = false : bool
- a orelse b;
val it = true : bool
- 10 div 3;
val it = 3 : int
- 10 mod 3;
val it = 1 : int
- ~10 div 3;
val it = ~4 : int
- ~10 mod 3;
val it = 2 : int
-
```

In the above example the user types in the expressions after the `ML` prompt `-` and the `ML` interpreter prints the result in the next line. Note that `ML` distinguishes between the binary operation of subtraction `-` from the unary minus `~` used for negative numbers. Further notice that `ML` follows standard mathematical convention for the `div` and `mod` operators, viz. that the quotient and remainder on division must satisfy the condition that the remainder is non-negative.

$\langle Example \rangle \equiv$ is not a part of the `ML` code. It is our convention for stating that $\langle Example \rangle$ is the name we will use to denote the following Program codes. The actual `ML` code follows in the `type-writer font`.

## 3.2 Substitution of functions

In what follows, we give a few examples of definition of one function in terms of another and the evaluation of such functions through substitution.

**Example 3.1** Finding the square of a natural number.

We can directly specify the function *square*, which is of the type $square : \mathbb{N} \to \mathbb{N}$ in terms of the standard *multiplication* function $* : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as

$$square(n) = n * n$$

Here, we are assuming that we can *substitute* one function for another provided they both return an item of the same type. To evaluate, say, $square(5)$, we have to thus evaluate $5 * 5$. An `ML` program for this function can be described as

$\langle Square \rangle \equiv$

```
fun square(n):int = n * n;
```

`fun` is a special word in `ML` (called a *keyword*) that is used for defining new functions. In our example we use it to define `square n` to be `n * n`. An invocation of the `ML` function with `square(5)` returns 25.

Thus, we can build more complex functions from simpler ones. As an example, let us define a function to compute $x^2 + y^2$.

**Example 3.2** Finding the sum of two squares.

We can define a function $sum\_squares : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as follows

$$sum\_squares(x, y) = square(x) + square(y)$$

The function *sum_squares* is thus defined in terms of the functions + and *square*. The corresponding `ML` program can be written as

$\langle Sum\ of\ squares \rangle \equiv$

```
fun sum_squares (x, y) =  square (x) + square (y);
```

An invocation of the ML function as `sum_squares (3, 4)` results in 25.

As another example, let us consider the following

**Example 3.3**
Let us define a function $f : \mathbb{N} \to \mathbb{N}$ as follows

$$f(n) = sum\_squares((n+1),(n+2))$$

In ML we can define it as

⟨*The function f*⟩≡
```
fun f (n) = sum_squares (n+1, n+2);
```

Invocation of the function with `f(5)` results in evaluation of `sum_squares (5+1, 5+2)` which, in turn, results in the evaluation of `square (6) + square (7)` yielding the final answer `+ (6 * 6) + (7 * 7)` which is 85.

### 3.2.1 Substitution using `let`

We often need local variables in our functions other than those defined as formal parameters. The ML function `let` allows for definition and substitution of local variables. We illustrate its use through the following example.

**Example 3.4** *Using* `let` *to define local variables.*
Suppose we wish to compute the function

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

We could also express this as

$$\begin{aligned} a &= 1+xy \\ b &= 1-y \\ f(x,y) &= xa^2 + yb + ab \end{aligned}$$

Thus we can avoid multiple computations of $1+xy$ and $1-y$ by using local variables $a$ and $b$.

In ML this can be achieved using the primitive `let` as follows

⟨*using* `let`⟩≡
```
fun f (x, y) =
   let
      val a = 1 + x * y;
      val b = 1 - y
   in  x*a*a + y*b + a*b
   end;
```

The general form of `let` is

⟨`let`⟩≡

```
let
   val ⟨var 1⟩ = ⟨exp 1⟩;
   val ⟨var 2⟩ = ⟨exp 2⟩;
       .
       .
       .
   val ⟨var n⟩ = ⟨exp n⟩
in  ⟨body⟩
end;
```

which can be thought of as

⟨⟩≡

```
let
   ⟨var 1⟩ have the value defined by ⟨exp 1⟩ and
   ⟨var 2⟩ have the value defined by ⟨exp 2⟩ and
       .
       .
       .
   ⟨var n⟩ have the value defined by ⟨exp n⟩
in ⟨body⟩
end;
```

## 3.3 Definition of functions using conditionals

In this section we give a few examples of function definitions using conditionals.

**Example 3.5** *Finding the larger of two numbers.*
Let us define a function $max : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$.The domain set for this function is the Cartesian product of natural numbers representing a pair, and the co-domain is the set $\mathbb{N}$. Thus, the function accepts a pair of natural numbers as its input, and gives a single natural number as its output. We define this function as

$$max(a, b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{otherwise} \end{cases}$$

While defining the function $max$, we have assumed that we can compare two natural numbers, with the $\geq$ function and determine which is larger. The basic primitive used in this case is *if-then-else*. Thus if $a \geq b$, the function returns $a$ as the output, else it returns $b$. Note that for every pair of natural numbers as its input, $max$ returns a unique number as the output and hence it adheres to the definition of a function given in Section 2.2. In `ML` the above definition looks as follows.

⟨*max*⟩≡
```
fun max (a, b):int =
   if a >= b then a
   else b;
```

**Example 3.6** *Finding the absolute value of $x$.*
We define the function $abs : \mathbb{Z} \rightarrow \mathbb{N}$ as

$$abs(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

In `ML` , we may define the above function as

⟨*abs (first alternative)*⟩≡
```
fun abs (x) =
   if x > 0 then x
   else
      if x = 0 then 0
      else ~x;
```

or, alternatively as

⟨*abs (second alternative)*⟩≡
```
fun abs (x) =
   if x < 0 then ~x
   else x;
```

In general, the conditional function `if` is expressed in `ML` as

⟨*if expression*⟩≡
```
if ⟨predicate⟩ then ⟨consequent⟩ else ⟨alternative⟩;
```

To evaluate the `if` function, the ⟨*predicate*⟩ is evaluated first. The ⟨*predicate*⟩ must evaluate to either *true* or *false*. If the ⟨*predicate*⟩ evaluates to *true*, the value of the ⟨*consequent*⟩ is returned. Otherwise the value of the ⟨*alternative*⟩ is returned.

## 3.4 Functions as inductively defined computational processes

All the examples we have presented so far are of functions which can be evaluated by substitutions or by evaluation of conditions. In what follows we give an example of an inductively defined functional algorithm for computing the GCD of two positive integers. This algorithm was described in Chapter 1.

**Example 3.7** *Computing the GCD of two numbers.*
We can define the function $gcd : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ as

$$gcd(a,b) = \begin{cases} a & \text{if } a = b \\ gcd(a-b,b) & \text{if } a > b \\ gcd(a,b-a) & \text{if } b > a \end{cases}$$

It is a function because for every pair of positive integers as input, it gives a positive integer as the output. It is also a finite computational process, because given any two positive integers as input, the description tells us, unambiguously, how to compute the solution and the process terminates after a finite number of steps. For example for the specific case of computing $gcd(18,12)$, we have

$$gcd(18,12) = gcd(12,6) = gcd(6,6) = 6.$$

In `ML` we can write the above function as

⟨*gcd*⟩≡
```
fun gcd (a, b) =
   if a = b then a
   else
      if a>b then gcd (a-b, b)
      else gcd (a, b-a);
```

However not all mathematically valid specifications of functions are algorithms. For example,

$$sqrt(n) = \begin{cases} m & \text{if } m * m = n \\ 0 & \text{if } \nexists m : m * m = n \end{cases}$$

is mathematically a perfectly valid description of a function of the type $sqrt : \mathbb{N} \to \mathbb{N}$. However the mathematical description does not tell us how to evaluate the function, and hence it is not an algorithm. An algorithmic description of the function would have to start with $m = 1$ and check if $m * m = n$ for all subsequent increments of $m$ by 1 till either such an $m$ is found or $m * m > n$. We will soon see how to describe such functions as computational processes or algorithms such that the computational processes terminate in finite time.

As another example of a mathematically valid specification of a function which is *not* an algorithm, consider the following functional description of $f : \mathbb{N} \to \mathbb{N}$

$$f(n) = 0 \text{ for } n = 0 \quad \text{and} \quad f(n) = f(n+1) - 1 \quad \text{for all } n \in \mathbb{N}$$

$f(n) = n \quad \forall n \in \mathbb{N}$ is a solution (though not the only one) to the above specification. However, it is not a valid algorithm because in order to evaluate $f(1)$ we have to evaluate $f(n+1)$ for $n = 1, 2, 3, \ldots$ which leads to an infinite computational process. One can rewrite the specification of the above function, in an *inductive* form, as

$$g(n) = \begin{cases} 0 & \text{if } n = 0 \\ g(n-1) + 1 & \text{otherwise} \end{cases}$$

Now this indeed defines a valid algorithm for computing $f(n)$. Mathematically the specifications for $f(n)$ and $g(n)$ are equivalent in that they both define the same function. However, the specification for $g(n)$ constitutes a valid algorithm whereas that for $f(n)$ does not. For successive values of $n$, $g(n)$ can be computed as

$$\begin{aligned} g(0) &= 0 \\ g(1) &= g(0) + 1 = 1 \\ g(2) &= g(1) + 1 = g(0) + 1 + 1 = 2 \\ &\vdots \end{aligned}$$

Similarly, consider the definition

$$f(n) = f(n)$$

Every function is a solution to the above but it is computationally undefined.

Thus, we see that **a specification of a function is an algorithm** only if it actually defines a precise computational procedure to evaluate it. For instance, any of the following constitutes an algorithmic description:

1. It is directly specified in terms of a pre-defined function which is either primitive or there exists an algorithm to compute it.
2. It is specified in terms of the evaluation of a condition.
3. It is inductively defined and the validity of its description can be established through the Principle of Mathematical Induction.
4. It is obtained through any finite number of combinations of the above three steps using substitutions.

In what follows we further elaborate on how we can describe functions as computational processes. Complex functions can be algorithmically defined in terms of two main types of processes - *recursive* and *iterative*.

## 3.5 Recursive processes

Recursive computational processes are characterized by a chain of deferred operations. As an example, we will consider an algorithm for computing the factorial of an integer $n$ ($n!$).

**Example 3.8** *Factorial Computation: Given $n \geq 0$, compute factorial of $n$ ($n!$)*[1].
Recall the inductive definition of $n!$ given inExample 2.1. On the basis of the inductive definition, we can define a functional algorithm for computing $factorial(n)$, which is of the type $factorial : \mathbb{N} \to \mathbb{N}$ as

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times factorial(n-1) & \text{otherwise} \end{cases}$$

Here $factorial(n)$ is the function "name" and the description after the = sign is the "body" of the function. A `ML` program for the above algorithm looks as follows:

⟨*Factorial*⟩≡
```
fun factorial (n) =
    if n = 0 then 1
    else n * factorial (n-1);
```

[1]The factorial function was first defined by Euclid in his *Elements* during the course of his proof of the existence of infinitely many prime numbers. This was written around 300 B.C.

It is instructive to examine the computational process underlying the above definition. The computational process, in the special case of $n = 5$, looks as follows

$$\begin{aligned}
&factorial(5)\\
&\quad = (5 \times factorial(4))\\
&\quad = (5 \times (4 \times factorial(3)))\\
&\quad = (5 \times (4 \times (3 \times factorial(2))))\\
&\quad = (5 \times (4 \times (3 \times (2 \times factorial(1)))))\\
&\quad = (5 \times (4 \times (3 \times (2 \times (1 \times factorial(0))))))\\
&\quad = (5 \times (4 \times (3 \times (2 \times (1 \times 1)))))\\
&\quad = (5 \times (4 \times (3 \times (2 \times 1))))\\
&\quad = (5 \times (4 \times (3 \times 2)))\\
&\quad = (5 \times (4 \times 6))\\
&\quad = (5 \times 24)\\
&\quad = 120
\end{aligned}$$

A computation such as this is characterized by a *growing* and *shrinking* process. In the *growing* phase each "call" to the function is replaced by its "body" which in turn contains a "call" to the function with different arguments. In order to compute according to the inductive definition, the actual multiplications will have to be postponed till the base case of $factorial(0)$ can be evaluated. This results in a growing process. Once the base value is available, the actual multiplications can be carried out resulting in a shrinking process. Computational processes which are characterized by such "deferred" computations are called *recursive*. This is not to be confused with the notion of a *recursive procedure* which just refers to the syntactic fact that the procedure is described in terms of itself.

Note that by a computational process we require that a machine, which has only the capabilities provided by the computational model, be able to perform the computation. A human normally realizes that multiplication is commutative and associative and may exploit it so that he does not have to defer performing the multiplications. However if the multiplication operation were to be replaced by a non-associative operation then even the human would have to defer the operation. Thus it becomes necessary to perform all recursive computations through deferred operations.

**Exercise 3.1** Consider the following example of a function $f : \mathbb{N} \to \mathbb{Z}$ defined just like factorial except that multiplication is replaced by subtraction which is not associative.

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n - f(n-1) & \text{otherwise} \end{cases}$$

1. Unfold the computation, as in the example of $factorial(5)$ above, to show that $f(5) = 2$.

2. What properties will you use as a human computer in order to avoid deferred computations?

## 3.6 Analysis of correctness and efficiency

In this section we deal with the methodology for the analysis of correctness and efficiency of functional algorithms.

### 3.6.1 Correctness

The correctness of the above functional algorithm can be established by using the Principle of Mathematical Induction. The algorithm adheres to an inductive definition and, consequently, can be proved correct by using **PMI**. Even though the proof of correctness may seem obvious in this instance, we give the proof to emphasize and clarify the distinction between a mathematical specification and an algorithm that implements it.
**To show that:** For all $n \in \mathbb{N}$, $factorial(n) = n!$ (i.e., the function $factorial$ implements the factorial function defined in Example 2.1).
*Proof:* By **PMI** on $n$.

**Basis.** When $n = 0$, $factorial(n) = 1 = 0!$ by definitions of $factorial$ and $0!$.

**Induction hypothesis.** For $k = n - 1$, $k \geq 0$, we have that $factorial(k) = k!$.

**Induction step.** Consider $factorial(n)$.

$$\begin{array}{lcll} factorial(n) & = & n \times factorial(n-1) & \\ & = & n \times (n-1)! & \text{by the induction hypothesis} \\ & = & n! & \text{by the definition of } n! \end{array}$$

Hence the function $factorial$ implements the factorial function $n!$. □

### 3.6.2 Efficiency

The other important aspect in the analysis of an algorithm is the issue of efficiency - both in terms of *space* and *time*. The efficiency of an algorithm is usually measured in terms of the space and time required in the execution of the algorithm (the space and time complexities). These are functions of the input size $n$.

A careful look at the above computational process makes it obvious that in order to compute $factorial(n)$, the $n$ integers will have to be remembered (or stacked up) before the actual multiplications can begin. Clearly, this leads to a space requirement of about $n$. We will call this the *space complexity*.

The time required to execute the above algorithm is directly proportional (at least as a first approximation) to the number of multiplications that have to be carried out and the number of function calls required. We can evaluate this in the following way. Let $T(n)$ be the number of multiplications required for a problem of size $n$ (when the input is $n$). Then, from the definition of the function $factorial$ we get

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases} \tag{3.1}$$

$T(0)$ is obviously 0, because no multiplications are required to output $factorial(0) = 1$ as the result. For $n > 0$, the number of multiplications required is one more than that required for a problem of size $n-1$. This is a direct consequence of the recursive specification of the solution. We can solve Equation 3.1 by telescoping, i.e.,

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= T(n-2)+2 \\ &\vdots \\ &= T(0)+n \\ &= n \end{aligned} \tag{3.2}$$

Thus $n$ is the number of multiplications required to compute $factorial(n)$ and this is the time complexity of the problem.

To estimate the space complexity, we have to estimate the number of deferred operations which is about the same as the number of times the function $factorial$ is invoked.

**Exercise 3.2** Show, in a similar way, that the number of function calls required to evaluate $factorial(n)$ is $n+1$.

Equation 3.1 is called a recurrence equation and we will use such equations to analyze the time complexities of various algorithms in these notes. Note that the measure of space and time given above are independent of how fast a computing machine is. Rather, it is given in terms of the amount of space required and the number of multiplications and function calls that are required. The measures are thus independent of any computing machine.

### 3.6.3 Efficiency, Why and How?

Modern technological advances in silicon have seen processor sizes fall and computing power rise dramatically. The microchip one holds in the palm today packs more computing power – both processor speed and memory size – than the monster monoliths that occupied whole rooms in the 50's. A sceptic is therefore quite entitled to ask: who cares about efficient algorithms anyway? If it runs too slow, just throw the processor away and get a larger one. If it runs out of space, just buy a bigger disk! Let's perform some simple back-of-the-envelope calculations to see if this scepticism is justified.

Consider the problem of computing the *determinant* of a matrix, a problem of fundamental importance in numerical analysis. One method is to evaluate the determinant by the well known formula:

$$\det A = \sum_{\sigma} (-1)^{\mathrm{sgn}\sigma} A_{1,\sigma(1)} \cdot A_{2,\sigma(2)} \cdots A_{n,\sigma(n)}.$$

Suppose you have implemented this algorithm on your laptop to run in $10^{-4} \times 2^n$ seconds when confronted with any $n \times n$ matrix (it will actually be worse than this!). You can solve an instance of size 10 in $10^{-4} \times 2^{10}$ seconds, i.e., about a tenth of a second. If you double

the problem size, you need about a thousand times as long, or, nearly 2 minutes. Not too bad. But to solve an instance of size 30 (not at all an unreasonable size in practice), you require a thousand times as long again, i.e. even running your laptop the whole day isn't sufficient (the battery would run out long before that!). Looking at it another way, if you ran your algorithm on your laptop for a whole year (!) without interruption, you would still only be able to compute the detrminant of a $38 \times 38$ matrix!

Well, let's buy new hardware! Let's go for a machine that's a hundred times as fast – now this is getting almost into supercomputing range and will cost you quite a fortune! What does it buy you in computing power? The same algorithm now solves the problem in $10^{-6} \times 2^n$ seconds. If you run it for a whole year non–stop (let's not even think of the electricity bill!), you can't even compute a $45 \times 45$ determinant! In practice, we will routinely encounter much larger matrices. What a waste!

**Exercise 3.3** *In general, show that if you were previously able to compute $n \times n$ determinants in some given time (say a year) on your laptop, the fancy new supercomputer will only solve instances of size $n + \log 100$ or about $n + 7$ in the same time.*

Suppose that you've taken this course and invest in algorithms instead. You discover the method of *Gaussian elimination* (we will study it later in these notes) which, let us assume, can compute a $n \times n$ determinant in time $10^{-2}n^3$ on your laptop. To compute a $10 \times 10$ determinant now takes 10 seconds, and a $20 \times 20$ determinant now requires between one and two minutes. But patience! It begins to pay off later: a $30 \times 30$ determinant takes only four and a half minutes and in a day you can handle $200 \times 200$ determinants. In a years's computation, you can do monster $1500 \times 1500$ determinants.

### 3.6.4 In the long run: Asymptotic analysis and Orders of growth

You may have noticed that there was something unsatisfactory about our way of doing things – the calculation was tuned too closely to our machine. The figure of $10^{-4} \times 2^n$ seconds is a bit arbitrary – the time to execute on one particular laptop – and has no other absolute significance for an analysis on a different machine. We would like to remedy this situation so as to have a mode of analysis that has *absolute significance* applicable to *any* machine. It should tell us precisely how the problem *scales* – how does the resource requirement grow as the size of the input increases – on any machine.

We now introduce one such machine independent measure of the resources required by a computational process – the *order of growth*. If $n$ is a parameter that measures the size of a problem then we can measure the resources required by an algorithm as a function $R(n)$. We say that the function $R(n)$ has an order of growth $O(f(n))$ (of order $f(n)$), if there exist constants $K$ and $n_0$ such that $R(n) \leq Kf(n)$ whenever $n \geq n_0$.

In our example of the computation of $factorial(n)$, we found that the space required is $n$, whereas the number of multiplications and function calls required are $n$ and $n + 1$ respectively. We see, that according to our definition of order of growth, each of these are $O(n)$. Thus, we can say that the *space complexity* and the *time complexity* of the algorithm

are both $O(n)$. In the example of determinant computation, regardless of the particular machine and the corresponding constants, the algorithm based on Gaussian elimination has time complexity $O(n^3)$.

Order of growth is only a crude measure of the resources required. A process which requires $n$ steps and another which requires $1000n$ steps have both the same order of growth $O(n)$. However, On the other hand, the $O(\cdot)$ notation has the following advantages:

- It *hides constants*, thus it is *robust across different machines*.
- It gives fairly precise indication of how the algorithm *scales* as we increase the size of the input. For example, if an algorithm has an order of growth $O(n)$, then doubling the size of the input will very nearly double the amount of resources required, whereas with a $O(n^2)$ algorithm will *square* the amount of resources required.
- It tells us which of two competing algorithm will win out eventually *in the long run*: for example, however large the constant $K$ may be, it is always possible to find a break point above which $Kn$ will always be smaller than $n^2$ or $2^n$ giving us an indication of when an algorithm with the former complexity will start working better than algorithms with the latter complexities.
- Finally the very fact that it is a crude analysis means that it is frequently much easier to perform than an exact analysis! And we get all the advantages listed above.

In what follows we will give examples of algorithms which have different orders of growth.

**Exercise 3.4** What does $O(1)$ mean? Are $O(1)$ and $O(n)$ different?

In Figure 3.1 we show the relative scaling of some order functions with respect to $n$. In Figure 3.2 we plot the $O(n^2)$ and the $O(2^n)$ curves with an increased $y$-axis range. Clearly any algorithm with a time complexity of $O(2^n)$ is computationally infeasible. In order to solve a problem of size 100 roughly $2^{100} \approx 10^{30}$ steps will be required.

**Exercise 3.5** Assuming that a single step may be executed in, say, $10^{-9}$ seconds, obtain a rough estimate to solve a problem of size 100 using an algorithm with a time complexity of $O(2^n)$.

## 3.7 More examples of recursive algorithms

Now that we have established methods for analyzing the correctness and efficiency of algorithms, let us consider a few more examples of fundamental recursive algorithms.

**Example 3.9** *Computing $x^n$:* Given an integer $x > 0$, compute $x^n$, where $n \geq 0$.

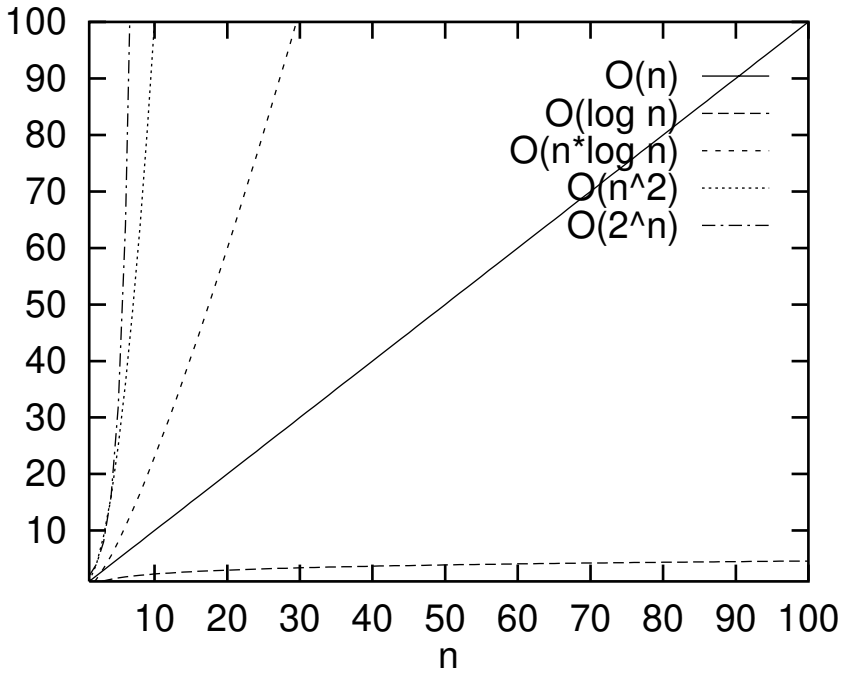


**Figure 3.1**: A comparison of various orders of growth.

We seek a function of the type $power : \mathbb{P} \times \mathbb{N} \to \mathbb{N}$. Let us develop this algorithm using **PMI- version 1** according to Example 2.2. Clearly, the base case specification can be given as $power(x, n) = 1$ if $n = 0$. If we assume, as the induction hypothesis, that we can compute $power(x, n-1) = x^{n-1}$ for an $n \geq 1$, then the induction step to compute $power(x, n) = x^n$ would be $x * power(x, n-1)$. Thus, an obvious algorithmic specification for this problem is

$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x * power(x, n-1) & \text{otherwise} \end{cases}$$

The correctness of the algorithm can be established by the **PMI**. See Example 2.2.

**Exercise 3.6** Show that the space and time complexities of the above algorithm are both $O(n)$.

An `ML` program for this function can be given as

$\langle Power \rangle \equiv$

```
fun power (x, n) =
    if n = 0 then 1
    else x * power (x, n-1)
```

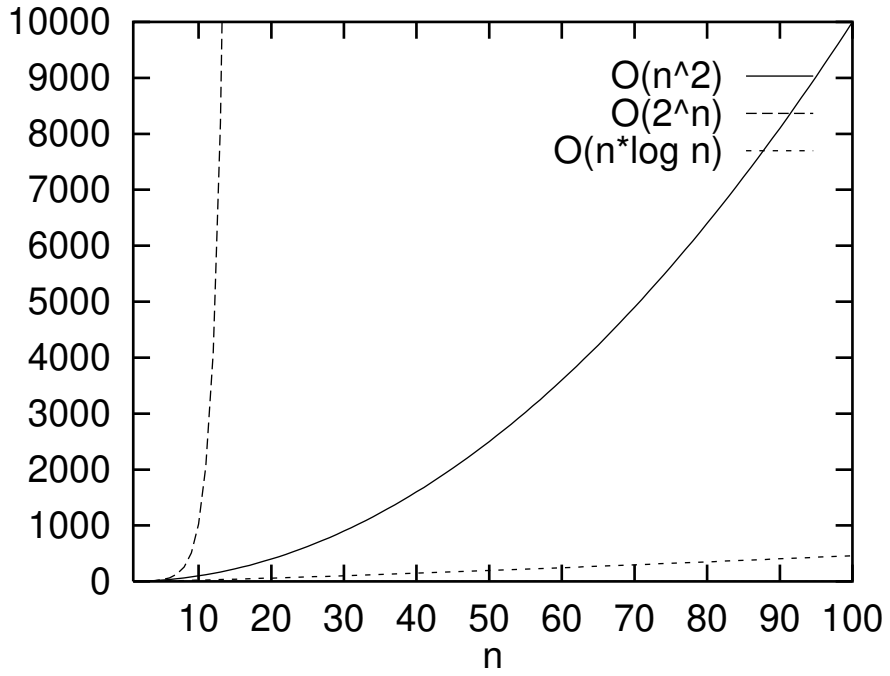


**Figure 3.2**: A comparison of $O(n^2)$, $O(n\ logn)$ and $O(2^n)$.

We can, however, significantly reduce the number of multiplications required by adopting the following strategy. Note that once we have computed $x^2$, we can compute $x^4$ by simply squaring it with only one multiplication, instead of the two required by the above scheme. Thus, we can compute $x^n$ by successive squaring.

We can again develop this algorithm according to the Principle of Mathematical Induction on $n$. The base can again be given as $power(x, n) = 1$ if $n = 0$. Let us assume that we can compute $x^{n\ div\ 2} = power(x, n\ div\ 2)$ as the inducuction hypothesis (we use **PMI** - version 3). Then $sqr(power(x, n\ div\ 2))$ would give us $x^{n-1}$ or $x^n$ depending on whether $n$ is odd or even. Thus the induction step to compute $x^n$ would be $x * sqr(power(x, n\ div\ 2))$ if $n$ is odd and $sqr(power(x, n\ div\ 2))$ if $n$ is even. This leads to the following algorithm specification[2]

$$fast\_power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x * square(fast\_power(x, (n\ div\ 2))) & \text{if } odd(n) \\ square(fast\_power(x, (n\ div\ 2))) & \text{otherwise} \end{cases}$$

where $odd(n) = ((n\ mod\ 2) = 1)$ and $square(x) = x * x$.

The correctness of the fast algorithm can be established as follows:

[2]The idea behind this algorithm is ancient. It appears in the Hindu *Chandah-sutra* by Acharya Pingala, written before 200 B.C. See Knuth 1969, section 4.6.3, for a more detailed discussion.

**Correctness**
**To show that:** $fast\_power(x, n) = x^n$ for all $x \in \mathbb{P}, n \in \mathbb{N}$.
*Proof:* By induction on $n$ using **PMI − version 3**.

**Basis.** for $n = 0$ we have $fast\_power(x, n) = 1 = x^0$ for any $x \in \mathbb{P}$.

**Induction hypothesis.** $fast\_power(x, m) = x^m$ for all $0 \leq m \leq (n-1)$ and for all $x \in \mathbb{P}$.

**Induction step.** Consider $power(x, n)$ for any $x \in \mathbb{P}$.

1. If $n$ is odd. Then $n = 2k + 1$ for some $k \geq 0$ and $n\ div\ 2 = k$.

$$\begin{array}{rcll} fast\_power(x,n) & = & x * (fast\_power(x, n\ div\ 2))^2 & \\ & = & x * x^{n\ div\ 2} * x^{n\ div\ 2} & \text{by induction hypothesis} \\ & = & x * x^{n-1} & \text{by the fact that } n \text{ is odd} \\ & = & x^n & \end{array}$$

2. If $n$ is even. Then $n = 2k$ for some $k \geq 0$ and $n\ div\ 2 = k$.

$$\begin{array}{rcll} fast\_power(x,n) & = & (fast\_power(x, n\ div\ 2))^2 & \\ & = & x^{n\ div\ 2} * x^{n\ div\ 2} & \text{by induction hypothesis} \\ & = & x^n & \text{by the fact that } n \text{ is even} \end{array}$$

□

**Efficiency**
To see that the successive squaring method is more efficient than our previous method, let us compute the number of multiplications required by the method of recurrence. For simplicity, we assume that $n$ is a power of 2 ($n = 2^m$). The recurrence is given by

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{for } n > 1 \end{cases}$$

we solve the recurrence equation to obtain

$$\begin{array}{rcl} T(n) & = & T(2^{m-1)} + 1 \\ & = & T(2^{m-2)} + 2 \\ & & \vdots \\ & = & T(2^{0)} + m \\ & = & m + 1 \\ & = & \log_2 n + 1 \end{array}$$

Thus, instead of $O(n)$ multiplications, the new algorithm requires only $O(\log_2 n)$ multiplications. (we will write this as $O(\lg n)$) multiplications. To see how significant this improvement is, we compare $n$ and $\lg n$ in the following table.

| $n$ | 2 | 4 | 8 | 16 | 32 | 64 | ... |
|---|---|---|---|---|---|---|---|
| $\lg n$ | 1 | 2 | 3 | 4 | 5 | 6 | ... |

The ML function corresponding to the fast powering algorithm is

⟨*Fast power*⟩≡
```
fun fast_power (x, n) =
    let fun odd (m) = (m mod 2 = 1)
     in if n=0 then 1
        else if odd (n) then x * square (fast_power (x, n div 2))
             else square (fast_power (x, n div 2))
    end;
```

`square` is the function we have previously defined and `div` is standard function in ML .[3]

**Exercise 3.7** For the fast method of powering –

1. Show that for any value of $n$ the number of multiplications required cannot be more than $\lceil 2 \lg n \rceil$. Hence conclude that the number of multiplications is $O(\lg n)$. For what values of $n$ do you require $\lceil 2 \lg n \rceil$ multiplications exactly.
2. Evaluate the number of function calls required.
3. Evaluate the space requirement.

**Example 3.10** *Fibonacci:* Computation of the $n^{th}$ Fibonacci number, $n \geq 1$.

The first few numbers in the Fibonacci sequence are

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

Each number beyond the first two is derived from the sum of its two nearest predecessors.

We can give a straightforward functional description for computing the $n^{th}$ Fibonacci number. It is a function of the type $fib : \mathbb{P} \to \mathbb{P}$

$$fib(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

The correctness of the algorithm is obvious from the inductive definition. We can write an ML function for the above as

⟨*Fibonacci*⟩≡
```
fun fib (n) =
    if (n=0) orelse (n=1) then 1
    else fib (n-1) + fib (n-2);
```

[3]Note that the most recent version of ML (version 110.0.3) assumes by default that all arithmetic variables and operations like `+`, `\*` are integer operations unless specified explicitly as real

It is instructive to look at the computational process underlying the computation of $fib(n)$. Let us consider the computation for the specific case of $n = 5$ (see Figure 3.3). Note that, unlike our previous examples which use one recursive call, $fib(n)$ is defined in terms of two recursive calls. This is an example of *nonlinear recursion* whereas all our previous examples were of *linear recursion.* As a consequence of the two recursive calls, in order to evaluate $fib(5)$ we have to evaluate $fib(4)$ and $fib(3)$. In turn, to evaluate $fib(4)$, we have to evaluate $fib(3)$ and $fib(2)$. Thus, we have to evaluate $fib(3)$ twice, which leads to inefficiency. In fact, the number of times $fib(1)$ or $fib(2)$ will have to be computed is $fib(n)$ itself.

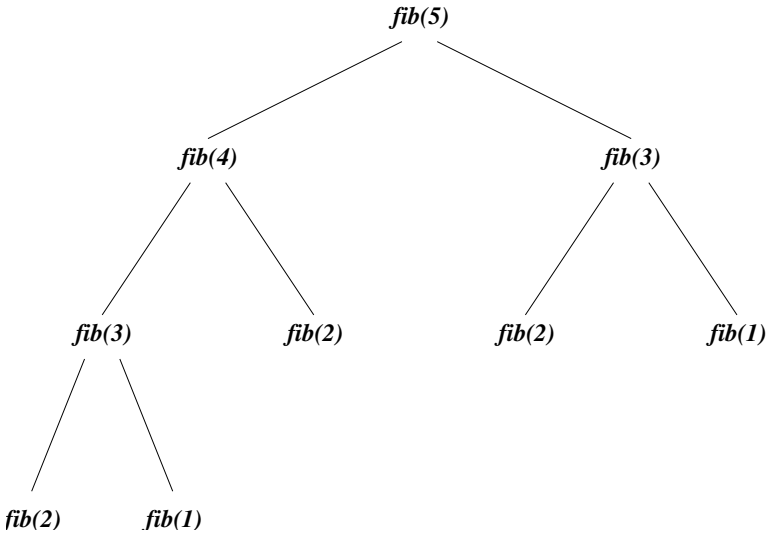


**Figure 3.3**: The unfolding of the computation of $fib(5)$

**Exercise 3.8** Show that the number of times $fib(1)$ or $fib(2)$ will have to be computed by the above algorithm while computing $fib(n)$ is equal to $fib(n)$ itself.

**Exercise 3.9** Verify, by induction, that $fib(n) = (\phi^n - \psi^n)/\sqrt{5}$, where $\phi = (1 + \sqrt{5})/2 = 1.618$ and $\psi = (1 - \sqrt{5})/2$. $\phi$ is called the *golden ratio*[4]

From the above exercises it is obvious that the time complexity of the above algorithm is clearly $O(\phi^n)$. Thus, the number of steps required to compute $fib(n)$ grows exponentially with $n$, and the computation is intractable for large $n$. $\phi^{100}$ is of the order of $10^{20}$, and, consequently, the evaluation of $fib(n)$ using the above algorithm will require of the order of $10^{20}$ function calls. This is a very large number indeed, and may take several years of computation even on the fastest of computers. In the Section 3.9 we will see how the computation of $fib(n)$ can be speeded up by designing an *iterative process.*

[4]Many of the ancient Greek monuments (including the Parthenon) had an elevation where the ratio of the base of the monument to its height was a close approximation of $\phi$. It was considered the most majestic proportion for temples. Can you give a ruler and compass construction of the golden ratio?

**Example 3.11** *Counting the number of primes between integers a and b (both inclusive).*

We will assume the availability of a function $prime(n)$ which returns *true* if $n$ is a prime and returns *false* otherwise. The function we are seeking is of the type $count\_primes : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. We can give an inductive definition of this function as

$$count\_primes(a,b) = \begin{cases} 0 & \text{if } a > b \\ count\_primes(a, b-1) + 1 & \text{if } prime(b) \\ count\_primes(a, b-1) & \text{otherwise} \end{cases}$$

We can establish the correctness of the above algorithms as follows.

**Correctness**
**To show that:** The function $count\_primes(a,b)$ returns the count of the number of primes between $a$ and $b$ assuming the function $prime(n)$ to be correct.
*Proof:* By **PMI** – Version 2 on $(b - a + 1)$.

**Basis.** If $a > b$, the interval is empty and $count\_primes(a,b)$ returns 0.

**Induction hypothesis.** $count\_primes(a, b-1)$ returns the count of the number of primes between $a$ and $b-1$ for $a, b$ such that $(b - a + 1) \geq 0$.

**Induction step.** If $b$ is a prime then $count\_primes(a,b)$ returns $count\_primes(a, b-1)+1$. Otherwise, it returns $count\_primes(a, b-1)$.

□

**Exercise 3.10** Show that the number of additions required and number of function calls to $prime(n)$ required are both $O(n)$ where $n = b - a$. Note that it is not possible to determine the time and space complexities of this algorithm without the knowledge of the complexities of the function $prime(n)$.

An `ML` function for the above can be written as

⟨*Count*⟩≡
```
fun count_primes (a, b) =
    if a > b then 0
    else if prime (b) then 1 + count_primes (a, b-1)
         else count_primes (a, b-1);
```

**Example 3.12** *Computing* $\sum_{n=a}^{b} f(n)$.

We will assume that the function $f(n)$ is available. We can then define the function $sum : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, inductively, as

$$sum(a, b) = \begin{cases} 0 & \text{if } a > b \\ f(b) + sum(a, b-1) & \text{otherwise} \end{cases}$$

**Exercise 3.11** For the algorithm described above

1. Establish the correctness by **PMI**.
2. Show that both the time and the space complexities of the algorithm are $O(n)$ where $n = b - a$. Assume that the function $f(n)$ can be computed using $O(1)$ time and space.

An `ML` function for the above can be written as

$\langle Sum \rangle \equiv$

```
fun sum (a, b) =
    if a > b then 0
    else sum (a, b-1) + f(b);
```

**Example 3.13** *Determining whether a positive integer is a perfect number.*
A positive integer is called a *perfect number* if the sum of its proper divisors add up to the number itself. $a$ is a proper divisor of $b$ if $a$ is a divisor of $b$ and $a \neq b$. The smallest examples of perfect numbers are 6 $(1 + 2 + 3 = 6)$ and 28 $(1 + 2 + 4 + 7 + 14 = 28)$ [5]. The next few perfect numbers are 496, 8128 and 33550336. Euclid devotes a chapter to perfect numbers in his *Elements*. There he proves that any number of the form $2^{p-1}(2^p - 1)$ is perfect, provided the odd factor $(2^p - 1)$, is prime. A few values of $p$ for these perfect numbers are $p = 2, 3, 5, 7, 13, 17, 19, 61, 107, 127, 257$.

We define a function $perfect? : \mathbb{P} \rightarrow \{true, false\}$ for determining whether a number is perfect or not in the following way.

$$perfect(n) = (n = addfactors(n))$$

where the function $addfactors : \mathbb{P} \rightarrow \mathbb{N}$ computes the sum of the proper factors of $n$. We can define add-factors as

$$addfactors(n) = sum(1, n \; div \; 2)$$

where *sum* is as defined in Example 3.12 and $f : \mathbb{P} \rightarrow \mathbb{N}$ is defined as

$$f(i) = \begin{cases} i & \text{if } n \; mod \; i = 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that the $n$ used in the definition of $f(i)$ is the same as in the function $perfect?$.

**Exercise 3.12** For the above algorithms

1. Establish the correctness.
2. Evaluate the space and the time complexities.

We can write an `ML` function for the above as

⟨*Perfect*⟩≡
```
fun perfect (n) =
    let
    ⟨Code for add_factors⟩
     in  n = add_factors (n)
    end;
```

[5] The smallest perfect numbers 6 and 28 were known to the Hindus as well as the Hebrews. Some commentators of the bible regard 6 and 28 as the basic numbers of the Supreme Architect. They point to the 6 days of creation and the 28 days of the lunar cycle. Others go so far as to explain the imperfection of the second creation by the fact that eight souls, not six, were rescued in Noah's ark. Said St. Augustine: "Six is a number perfect in itself, and not because God created all things in six days; rather the converse is true; God created all things in six days because this number is perfect, and it would have been perfect even if the work of six days did not exist."

⟨*Code for add-factors*⟩≡
```
fun add_factors (n) =
    let
     ⟨Code for f(i)⟩;
     ⟨Code for sum⟩
     in sum (1, n div 2)
    end;
```

⟨*Code for* `f(i)`⟩≡
```
  fun f (i) =
      if n mod i = 0 then i
      else 0;
```

⟨*Code for sum*⟩≡
```
fun sum (a, b) =
    if a > b then 0
    else f(b) + sum (a, b-1);
```

Thus, the entire code can be given as

⟨*Entire code for* `perfect (n)`⟩≡
```
fun perfect (n) =
    let fun add_factors (n) =
            let fun f (i) =
                    if n mod i = 0 then i
                    else 0;
                fun sum (a, b) =
                    if a > b then 0
                    else f(b) + sum (a, b-1);
             in sum (1, n div 2)
             end;
     in  n = add_factors (n)
    end;
```

**Exercise 3.13** Using the property that if $i$ is a divisor of $n$ then $(n \; div \; i)$ is also a divisor of $n$, give an improved version of the above algorithm and thus improve the complexity from $O(n)$ to $O(\sqrt{n})$. What happens if $n$ is a perfect square? Write a `ML` program to implement your improved algorithm.

The above is a typical example of program development through *top down* design and *step-wise* refinement. We strongly recommend this method of program development and will adhere to this method for most examples in these notes.

It is instructive to note the nesting of the various `ML` functions declared above. The function `add-factors` is *local* to the function `perfect`. Hence it cannot be directly accessed from the level from which `perfect` can be invoked. The accessibility of various variables and functions from different parts of the `ML` code is guided by the *Scope rules* in functional programming. In what follows in the next section we formalize the notion.

## 3.8 Scope rules

In this section we introduce and formalize the notion of *scope* and the concepts of *free* and *bound* variables. As will be evident these concepts play quite an important role in programming. They also exist in mathematics as we illustrate by the following examples.

**Example 3.14** Consider the expression $\sum_{n=a}^{b} f(n)$ in Example 3.12. It contains the following names

$$a, b, n, f$$

Of these we do not know what $a$, $b$ and $f$ denote except that we assume that $a$ and $b$ are natural numbers and $f$ is a function on natural numbers. Hence the names $a$, $b$ and $f$ are called *free* in the expression $\sum_{n=a}^{b} f(n)$. However $n$ is said to be *bound* in the sense that the expression makes it clear that $n$ ranges over the interval $[a, b]$ and is used only in order to facilitate the definition of the summation function. Further the *scope* of $n$ is limited to the summation expression and we say that $n$ is *local* to the summation function.

**Example 3.15** Consider the following indefinite integral

$$\int_0^z \left( \int_0^y f(x)dx + \int_0^y g(u)du \right) dy$$

It contains as *free* the names $z$, $f$ and $g$. The other names $x$, $u$ and $y$ are *bound*. The scopes of the *bound* variables are shown below.

$$\underbrace{\int_0^z \left( \underbrace{\int_0^y f(x)dx}_{x} + \underbrace{\int_0^y g(u)du}_{u} \right) dy}_{y}$$

Note that an equivalent way of writing this indefinite integral is

$$\int_0^z \left( \int_0^y f(x)dx + \int_0^y g(x)dx \right) dy$$

where the two uses of $x$ in the two different integrals are meant to denote different variables. Further we may note that though $y$ is a bound variable of the complete expression, when we consider only the sub-expressions

$$\int_0^y f(x)dx \text{ and } \int_0^y f(x)dx$$

$y$ is *free* in both. It is also *free* in the sub-expression

$$\left( \int_0^y f(x)dx + \int_0^y g(x)dx \right)$$

However it is *bound* when the integral over $y$ is performed.

**Example 3.16** Now consider the complete `ML` code of Example 3.13 (perfect numbers).

- The name `perfect` is bound and has a scope which extends beyond the definition. This implies that in some later program in the same file or `ML` session one could use this name to mean exactly what we have defined it to be.
- The name `add-factors` is bound and has a scope which begins with its definition and extends right up to the end of the definition of `perfect (n)` but no further. Hence if after defining `perfect (n)` as given one types in, say, `add-factors (12)` in the same session then one would get an error. This is because `add-factors` has no meaning outside the scope of the definition of `perfect (n)`.
- Similarly the name `f` is bound and has a scope that extends up to the end of the definition of `add-factors` and no further. The name `sum` also has a scope similar to that of `f`.
- The variables `a` and `b` are bound and have scopes beginning at their first occurrence in the definition of `sum` and ending with the same definition.
- Similarly `i` in `(define (f i) ..)` has a scope that extends over the definition of `f` and no further.
- The name `n` in the definition of the function `f` has a scope that begins with its first occurrence in the definition of the function `add-factors (n)` and extends only up to the end of this definition of `add-factors` and no further. Thus within the scope of the function `f` the variable `n` is free. The variable `n` in the definition `perfect (n)` has a scope that extends up to the end of that definition. It is important to note that the variable `n` in the definition `perfect (n)` and the variable `n` in the definition of `add-factors` are actually different. We could, for example, replace all occurrences of `n` in the scope of `add-factors` with `m` without affecting the program in any way.

- There are a few other names used in the program like `div` and `mod`. At the initiation of the `ML` session these functions are automatically loaded by the `ML` interactive system and therefore they occur as bound names whose scopes extend right up to the end of the session. It is however possible to create a large "hole" in the scopes of these definitions by writing our own definition of `div` and `mod`

## 3.9 Tail-recursion and iterative processes

So far we have considered computations based on recursive processes which are characterized by deferred computations (see Section 3.5). The deferred computations invariably lead to a high space complexity for the algorithms. For example, the algorithm for computing $factorial(n)$ discussed in Example 3.8, has a space complexity of $O(n)$ as a consequence of the deferred computations. Also, in some cases like the computation of $fib(n)$, an algorithm described in terms of a recursive process leads to unacceptably high time complexities. In this section, we will see how such inefficiencies can be removed by describing alternative algorithms for these problems using tail-recursion which lead to iterative computational processes.

The crucial idea in iterative algorithms is to represent the *state* of the computation at each stage in terms of auxiliary variables so as to obtain the final result from the *final state* of these variables. We may think of the state of a computation as a collection of instantaneous values of certain quantities. This is analogous to the notion of the state of a particle in some good old problem in Physics - the state of a particle is described in terms of its mass, position, velocity and acceleration at any instant of time.

As an example of an iterative algorithm described through state changes, let us consider the problem of computation of $factorial(n)$ again and design an iterative algorithm for the problem.

**Example 3.17** *Iterative computation of factorial.*
We maintain the *state* of the factorial computation in terms of three auxiliary variables $f$, $c$ and $m$. We start with initial values $f = f_0$ when $c = c_0$, and successively increment the value of the counter $c$ by 1 while maintaining, at every stage, the following condition about the state of the computation *invariant*

$$(c_0 \leq c \leq m) \wedge (f = f_0 * \prod_{i=c_0+1}^{c} i) \wedge (f_0 * \prod_{i=c_0+1}^{m} = f * \prod_{i=c+1}^{m} i)$$

Then, when $c = m$ we can obtain $f = f_0 * \prod_{i=c_0+1}^{m}$ as the final result. This is the same as $factorial(n)$ if the initial values are $m = n$, $c_0 = 0$ and $f_0 = 1$ respectively. The resulting algorithm is described below.

$$factorial(n) = fact\_iter(n, 1, 0)$$

where, the auxiliary function $fact\_iter : \mathbb{N} \times \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{P}$ is given as

$$fact\_iter(m, f, c) = \begin{cases} f & \text{if } c = m \\ fact\_iter(m, f * (c+1), c+1) & \text{otherwise} \end{cases}$$

Note that the invariant condition (which is a boolean function of the state of the system described in terms of the variables $f$ and $c$) holds true every time the function $fact\_iter$ is invoked. We can write an `ML` program for this iterative version as follows

⟨*Iterative factorial*⟩≡
```
fun factorial (n) =
    let ⟨Code for fact_iter⟩
     in fact_iter (n, 1, 0)
    end;
```

⟨*Code for* `fact_iter`⟩≡
```
fun fact_iter (m, f, c) =
    if c=m then f
    else fact_iter (m, f*(c+1), c+1);
```

The function description of $fact\_iter$ is called *tail-recursive* because the "otherwise" clause in its description is a simple recursive call to the function itself. Contrast this with the "otherwise" clause of the recursive factorial (described in Example 3.8) which is given as $n * factorial(n-1)$ and involves the recursive call with the multiplication operation. A tail-recursive definition such as this leads to a computational process different from that of the recursive version for the same problem. The underlying computational process for the special case of $factorial(5)$ looks as follows

$$\begin{aligned} & factorial(5) \\ & \quad = fact\_iter(5,1,0) \\ & \quad = fact\_iter(5,1,1) \\ & \quad = fact\_iter(5,2,2) \\ & \quad = fact\_iter(5,6,3) \\ & \quad = fact\_iter(5,24,4) \\ & \quad = fact\_iter(5,120,5) \\ & \quad = 120 \end{aligned}$$

Contrast this with the recursive process for computing $factorial(n)$ in Example 3.8. The recursive process is characterized by a growing and shrinking due to deferred computations, where, in the growing process, the multiplicative constants 5,4,3,2 and 1 are stacked up before the results of $factorial(0)$, $factorial(1)$, $factorial(2)$, $factorial(3)$ and $factorial(4)$ become available. In the shrinking process the actual multiplications $n * factorial(n-1)$ are carried out to obtain $factorial(n)$ successively. In contrast, there is no growing process in the iterative version. The results of the successive stages are captured in the value of $f$ where the stage itself is indicated by the value of $c$. The values of these two variables, at any instant, give the state of the computation.

The time complexity of the iterative algorithm is clearly $O(n)$ which is same as that of the recursive one, whereas the space complexity in this case reduces to $O(1)$. This is because, at any stage, the instantaneous values of only three variables are required to be stored.

### 3.9.1 Correctness of an iterative process

The correctness of an iterative process can be established by an analysis of the invariant condition. In fact, the invariant condition is merely an encoding of the proof of correctness by mathematical induction.

To illustrate this, let us first give a proof of correctness of $fact\_iter$ using **PMI**.
**To show:** For all $m, f, c$ such that $0 \leq c \leq m$

$$fact\_iter(m,f,c) = f * \prod_{i=c+1}^{m} i$$

*Proof:* Using **PMI** – Version 1 on $(m-c)$.

**Basis.** $(m - c) = 0$ or $(m = c)$.

$$fact\_iter(m, f, c) = f = f * \prod_{i=c+1}^{m} i = f * 1$$

**Induction hypothesis.** For some $k = (m - c) \geq 0$,

$$fact\_iter(m, f, c) = f * \prod_{i=c+1}^{m} i$$

**Induction step.** Let $(m - c) = k + 1 > 0$. Then

$$\begin{array}{rcll} fact\_iter(m, f, c) & = & fact\_iter(m, f * (c + 1), c + 1) & \\ & = & f * (c + 1) * \prod_{i=c+2}^{m} i & \text{by Inductive hypothesis} \\ & = & f * \prod_{i=c+1}^{m} i & \end{array}$$

□

Then we can prove the correctness of the function $factorial(n)$ as follows:
*Proof:*

$$factorial(n) = fact\_iter(n, 1, 0) = 1 * \prod_{i=1}^{n} i = n!$$

□

On the other hand, the invariant condition

$$(c_0 \leq c \leq m) \wedge (f = f_0 * \prod_{i=c_0+1}^{c} i) \wedge (f_0 * \prod_{i=c_0+1}^{m} = f * \prod_{i=c+1}^{m} i)$$

encodes the above proof of correctness through a description of state changes. At the initial stage, when $c = c_0$, the invariant condition gives us that $f = f_0$. At the final stage when $c = m$, the invariant condition gives us that $f = f_0 * \prod_{i=c_0+1}^{m} i$ which is the final value that the function returns. According to the initial invocation of $fact\_iter$ from the function $factorial$, the initial values are $f_0 = 1$, $c_0 = 0$ and $m = n$. Thus the final value of $f$ is $f = \prod_{i=1}^{m} i = n!$.

Since iterative algorithms are described through state changes, for correct design of an iterative algorithm, it is helpful to first design the invariant condition such that the desired result can be obtained from the final state of the variables. The invariant condition can then act as a specification for the design of the algorithm. In what follows, we give some more examples of iterative processes.

## 3.10 More examples of iterative processes

**Example 3.18** *Iterative computation of* $\sum_a^b f(n)$.

As before, we assume that the function $f(n)$ is available. We can describe the iterative process in terms of the auxiliary variables $s$ and $c$. We can initialize the process with $c = c_0$ and $s = s_0 = 0$, keep computing the partial sum $s = \sum_{i=c_0}^{c-1} f(i)$, and continue the iterative process till $c$ reaches the final value $c_f + 1$ . An invariant capturing the above idea can be written as

$$INV = (c_0 \leq c \leq c_f + 1) \wedge (s = \sum_{i=c_0}^{c-1} f(i)) \wedge (s + \sum_{i=c}^{c_f} f(i) = \sum_{i=c_0}^{c_f} f(i))$$

In order to compute $\sum_a^b f(n)$ using the computational process described by the above invariant, we have to initialize the process with $c_0 = a$, $c_f = b$ and $s = 0$. We can describe the iterative algorithm for $sum : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as

$$sum(a, b) = sum\_iter(a, b, 0)$$

where, the auxiliary function $sum\_iter : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given as

$$sum\_iter(c, c_f, s) = \begin{cases} s & \text{if } c = c_f + 1 \\ sum\_iter(c + 1, c_f, s + f(c)) & \text{otherwise} \end{cases}$$

**Exercise 3.14** For the above algorithm

1. Establish the correctness, independently, using both **PMI** and the invariant property.
2. Estimate the time and space complexities assuming that $f(n)$ can be computed in $O(1)$ time using $O(1)$ space.

The `ML` function for the above can be written as

⟨*Iterative sum*⟩≡
```
fun sum (a, b) =
    let ⟨Code for sum_iter⟩
     in sum_iter (a, b, 0)
    end;
```

⟨*Code for* `sum_iter`⟩≡
```
fun sum_iter (c, cf, s) =
    if c = cf+1 then s
    else sum_iter (c+1, cf, s + f(c));
```

Note that in the above code `f` occurs *free* in the definition. Hence it is necessary to have already the function `f` previously in the `ML` session before using these definitions.

**Example 3.19** *Euclid's algorithm* [6] *for GCD.*
Euclid's algorithm for computing the GCD of two numbers can be expressed in a functional form as follows. It is a function of the type $Euclid\_gcd : \mathbb{P} \times \mathbb{N} \to \mathbb{P}$.

$$Euclid\_gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ Euclid\_gcd(b, (a \ mod \ b)) & \text{otherwise} \end{cases}$$

Note that the algorithm is tail-recursive, and, consequently, generates an iterative process.
**Correctness**
We will first prove the correctness by mathematical induction. Then we will construct an invariant for the algorithm and analyze the correctness using the invariant. In either case we require the following result which was proved by Euclid.
**Claim:** If $a = qb + r$, $0 < r < b$, then $gcd(a,b) = gcd(b,r)$
*Proof:* If $d = gcd(a,b)$ then $d \mid a$ ($d$ divides $a$) and $d \mid b$ which, in turn, implies that $d \mid (a - qb)$, or $d \mid r$. Thus $d$ is a common divisor of $b$ and $r$. If $c$ is any common divisor of $b$ and $r$, then $c \mid (qb + r)$ which implies that $c \mid a$. Thus $c$ is a common divisor of $a$ and $b$. Since $d$ is the largest divisor of both $a$ and $b$, it follows that $c \leq d$. It now follows from definition that $d = gcd(b,r)$. □

We will now prove using **PMI** that for all $b \geq 0$, for all $a > 0$, $Euclid\_gcd(a,b) = gcd(a,b)$.
*Proof:* By **PMI** – Version 3 on $b$.

**Basis.** $b = 0$. If $b = 0$ then for all $a > 0$, $Euclid\_gcd(a,b) = a = gcd(a,b)$.

**Induction hypothesis.** For all $b \leq k$ such that $0 \leq b$, for all $a > 0$, $Euclid\_gcd(a,b) = gcd(a,b)$.

**Induction step.** Consider $b = k + 1$, $a > 0$.

$$\begin{array}{rcll} Euclid\_gcd(a,b) & = & Euclid\_gcd(b, a \ mod \ b) & \\ & = & gcd(b, a \ mod \ b) & \text{by the inductive hypothesis} \\ & = & gcd(a,b) & \text{by the Claim above} \end{array}$$

□

If $a_0$ and $b_0$ are the initial values of $a$ and $b$ respectively, an invariant condition for the above is

$$INV = (gcd(a,b) = gcd(a_0,b_0)) \wedge (a > 0) \wedge (b \geq 0)$$

[6] So called because it appears in Euclid's *Elements*. This book was written around 300 B.C. According to Knuth (1969) it may be considered the oldest known non-trivial algorithm. The Egyptian method of multiplication (Problem 5) is definitely older, but, as Knuth explains, Euclid's algorithm is the oldest known to have been presented as a general algorithm, rather than as a set of illustrative examples.

**Exercise 3.15** Verify that the invariant condition is satisfied both at the initial and the final stage of the algorithm.

**Efficiency**
To analyze the efficiency of the Euclid's algorithm for *gcd* we need the following result.
**Lamé's Theorem:** If Euclid's algorithm requires $k$ steps to compute the *gcd* of some pair, then the smaller number in the pair must be greater than or equal to the $k^{th}$ Fibonacci number.

**Exercise 3.16** Verify the validity of Lamé's Theorem.

We can use this theorem to analyze the time complexity of the Euclid's algorithm. Let $n$ be the smaller of the two inputs to the function. If the process takes $k$ steps, then we must have $n \geq fib(k) \approx \phi^k$. Thus, the number of steps $k$ is $O(\log n)$.

The `ML` function that implements the algorithm can be written as

⟨*gcd*⟩≡
```
fun Euclid_gcd (a, b) =
    if b=0 then a
    else Euclid_gcd (b, a mod b);
```

**Example 3.20** *Iterative computation of Fibonacci numbers.*
In Example 3.10 we saw that the computation of $fib(n)$ by a purelyrecursive process requires an exponential number of operations. Thus the computation is intractable. However, the computation can be speeded up by designing an alternative iterative process. An easy way around the problem is to start with the two smallest Fibonacci numbers, and retain the values of the last two Fibonacci numbers as the *state* of the system, in two auxiliary variables $a$ and $b$ (say) and to compute the $n^{th}$ Fibonacci number from the previous two. We can use a counter *count*, to represent the stages of the computation. An invariant condition for the above process for $n \geq 3$ is

$$(n \geq 3) \wedge (3 \leq count \leq n) \wedge (a = fib(count - 2)) \wedge (b = fib(count - 1))$$

Then, when $count = n$, the process may terminate and we may obtain the value $a + b = fib(count - 2) + fib(count - 1) = fib(n)$ as the final answer. An algorithm based on this invariant condition can be described as

$$fib(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib\_iter(n, 1, 1, 3) & \text{otherwise} \end{cases}$$

where $fib\_iter(n, a, b, count) : \mathbb{P} \times \mathbb{P} \times \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ is an auxiliary function defined as

$$fib\_iter(n, a, b, count) = \begin{cases} a + b & \text{if } count = n \\ fib\_iter(n, b, a + b, count + 1) & \text{otherwise} \end{cases}$$

The function $fib\_iter(n, a, b, count)$ is invoked only if $n \geq 3$, and every time this function is invoked, the invariant condition holds. The above process obviously requires only $n - 2$ additions to compute $fib(n)$ for $n \geq 3$. Thus, the iterative algorithm requires $O(n)$ time and $O(1)$ space for computing the $n^{th}$ Fibonacci number. This is a significant improvement over the purely recursive version we considered earlier.

**Exercise 3.17** Establish the correctness of the above algorithm.

An `ML` function for the the iterative computation of Fibonacci is

⟨*Iterative Fibonacci*⟩≡
```
fun fib (n) =
    let ⟨Code for fib_iter⟩
     in if n<=2 then 1
        else fib_iter (n, 1, 1,3)
    end;
```

⟨*Code for* `fib_iter`⟩≡
```
fun fib_iter (n, a, b, count) =
    if count = n then a+b
    else fib_iter (n, b, a+b, count+1);
```

## Problems

For each of the problems given below, identify the types of the functions you need to define, establish their correctness using **PMI** and invariants (if the algorithm is iterative) and determine the space and time complexities. Finally, translate your functional algorithms into `ML` programs and execute them.

1. Construct both recursive and iterative algorithms for

   (a) Finding the number of digits (in base 10) in a given positive integer assuming there are no leading zeroes.

   (b) Reversing the digits of a positive integer in base 10.

2. Suppose we rewrite the fast powering algorithm described in Example 3.9 as follows

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x * power(x,(n\ div\ 2)) * power(x,(n\ div\ 2)) & \text{if } odd(n) \\ power(x,(n\ div\ 2)) * power(x,(n\ div\ 2)) & \text{otherwise} \end{cases}$$

   Do you foresee any problem?

3. Define a tail-recursive (iterative) algorithm for the function $f : \mathbb{N} \to \mathbb{Z}$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n - f(n-1) & \text{otherwise} \end{cases}$$

   Define an invariant property for the above algorithm.

4. Design an iterative process that uses successive squaring to compute $x^n$ and works in $O(\lg n)$ time.

5. The powering algorithm is based on successive multiplications. Similarly, one can devise an integer multiplication algorithm based on repeated additions. Design an iterative functional algorithm for integer multiplication in terms of adding, doubling and halving. The algorithm should work in $O(\lg n)$ time.[7]

6. If $a \geq 0$ and $b > 0$ are two integers, then there exists $q \geq 0$ and $0 \leq r < b$ such that $a = q*b+r$. The *div* and *mod* functions are defined as $div(a,b) = q$ and $mod(a,b) = r$. Develop iterative algorithms for *div* and *mod* using addition and subtraction.

7. Amicable numbers are pairs of numbers each of whose proper divisors add up to the other (1 is included as a divisor but the numbers are not included as their own

[7] This algorithm which is sometimes known as the 'Russian peasant method" of multiplication, is very old. Examples of its use are found in the Rhind Papyrus, one of the two oldest mathematical documents in existence, written about 1700 B.C. (and copied from an even older document) by an Egyptian scribe named A'h-mose.

divisors). The smallest pair of amicable numbers are 220 and 284 [8]. Develop a functional algorithm to determine whether a given pair of numbers are amicable or not.

8. Develop a functional algorithm to determine whether a given number is a prime or not.

9. Develop an algorithm to compute the sum of the first $n$ Fibonacci numbers. The algorithm should work in $O(n)$ time and $O(1)$ space.

10. Given that $t_n$ is the $n^{th}$ term in the expansion of $\sin(x)$, write a function to determine the $n + 1^{th}$ term.

11. Using your solution for the above, write a function to evaluate the value of $sin(x)$ up to $n$ terms.

12. Suppose you have an infinite supply of coins of denomination $50p$, $25p$, $10p$, $5p$ and $1p$. In how many ways can you generate change for a given amount, say Rs. $1/- = 100p$. Given that a function $d(n)$ is available which gives the denomination of the $n^{th}$ type of coin, develop a recursive algorithm to count the number of ways to generate change for a given amount. What can you say about the number of steps required for the computation?

13. Write an iterative function for the coin exchange problem. Show that a suitably defined iterative function will work faster for the given problem.

[8] Down through their quaint history, amicable numbers have been important in magic and astrology, and in casting of horoscopes, making talismans, and concocting love potions. The philosopher Iamblichus of Chalcis (A.D 250 - A.D 330) ascribed a knowledge of the pair 220 and 284 to the Pythagoreans. He wrote: "They [the Pythagoreans] call certain numbers amicable numbers, adopting virtues and social qualities to numbers, as 220 and 284; for the parts of each have the power to generate the other". See *Elementary number theory* by D. M. Burton for details.

# Chapter 4

# The Imperative model of computation

In the last chapter we studied the basics of a functional model of computation. Though the functional model is attractive from the point of view of ease of algorithm design and correctness analysis, *imperative models* of computation are more commonly used in practice mainly because of reasons of efficiency. In particular, imperative programming languages like `Fortran` and `C` have been thoroughly optimized through years of research, and programs compiled in these languages, in general, work faster. Hence, in this chapter, we will introduce the basics of an imperative model of computation. We will use the programming language `Python` as a representative language for writing programs for imperative algorithms. In the next chapter we will relate the two models of computation and show how an initial functional design of an algorithm helps in the development of an imperative algorithm using *step-wise refinement*.

In the imperative model of computation an algorithm is a specification of *what to do* in order to solve a given problem in terms of a sequence of instructions which have to be executed in the given order. In what follows we describe the primitives for the imperative model.

## 4.1 The primitives for the imperative model

In the last chapter we have seen in the cases of both the factorial and fibonacci computations that an iterative process is often more efficient than a recursive process for the same problem. However, we had used a functional model to describe iterative processes in which we introduced the notion of the state of the computation.

A careful look at the computation of iterative processes reveals that we have a starting state from which the desired final state is obtained. For describing iterative processes, it is often more convenient to use a model of computation which merely describes state changes.

In this chapter we consider the *imperative model* of computation which allows us to describe how a state should be changed. As we have mentioned in the last chapter, we

may think of the state of a computation as a collection of instantaneous values of certain quantities. A state change occurs if at least one of the quantities comprising the state is changed.

The imperative model of computation uses *instructions* or *commands* to make desired state changes. Hence the concept of a variable in the imperative model is really that it is a quantity whose value can be changed through an appropriate command. The primary command which does this in the imperative model is the 'assignment' instruction which we describe below.

### 4.1.1 Variables and the *assignment* instruction

The variables which we have been using in the functional model (e.g., $n, a, b, count$ etc.) have no special meaning other than that they can be instantiated with values of a given type. These variables are similar to those used in *algebra*. In contrast, in the *imperative* model we will use variables to store the *state* of a computation. In this model, a "declaration" of variables to be of a certain type with a statement like

```
int a,b
```

creates and reserves two *locations* (or boxes) with the names $a$ and $b$ in which integer values can be stored.


Initially, immediately after the declaration, the contents of the boxes with names $a$ and $b$ are *undefined*. The following assignment instruction stores the integer 2 in the location named $a$

```
a = 2
```

The above is read as "$a$ assigned 2". The content of the location $b$ is still undefined and the *state* of the variables is


A subsequent assignment instruction of the form

```
b = a
```

changes the state to


Note that the content of $a$ gets copied to $b$, and $a$ is unchanged. A further assignment instruction

```
a = a + b
```

changes the state to:


The contents of $a$ gets replaced by the sum of the present contents of $a$ and $b$, whereas the contents of $b$ remains unchanged.

On the left of the = operator must be a single variable whose state is being updated, and on the right must be an expression comprising operators, variables and values. The right side must evaluate to the same type as the variable specified in the left.

As a result of this, in the imperative style, state changes can be performed only one at a time. Consequently, simultaneous changes in the values of several variables have to be performed in some orderly fashion, one-at-a time, so as to ensure that the desired final state is obtained through several one-step changes.

The *state* of the computation at any instant is a snapshot of the contents of all the variables used in the algorithm.

As an example of an imperative style algorithm using the assignment instruction, let us consider the following problem of swapping the contents of two variables.

**Example 4.1** *Exchanging the values of two variables a and b (swapping).*
Given the initial state of two variables $a$ and $b$ of the same type, we have to construct an algorithm to exchange their values (contents). Assuming that the initial contents of $a$ and $b$ are $a0$ and $b0$ respectively, we can describe the initial and the final states as

$$\textbf{Pre-condition: } (a = a0) \wedge (b = b0)$$
$$\textbf{Post-condition: } (a = b0) \wedge (b = a0)$$

The 'pre-condition' and the 'post-condition' are logical properties of the initial and the desired final states of the computation respectively. Together they form the specification for the algorithm. Thus the objective of the algorithm is to change the state of the computation, through a sequence of steps, so that finally the post-condition is *true* given that the pre-condition is *true* to start with. We can achieve the transformation by using a variable *temp* for temporary storage. We can first copy the contents of $a$ to *temp*, then replace the contents of $a$ with the contents of $b$, and finally replace the content of $b$ with that of *temp*. We describe the complete imperative algorithm as

# assert $A : (a = a0) \wedge (b = b0)$

```
temp = a
a = b
b = temp
```

# assert $B : (a = b0) \wedge (b = a0)$

In the above example, the three assignment instructions have to be executed in the given order to achieve the exchange. An algorithm in the imperative model is a sequence of instructions which have to be executed in a step by step manner to carry out a desired computation. The instructions are separated by a line separator. The sequence of three instructions may be regarded as a single *compound* instruction. To enable us to regard certain sequences of instructions as a single instruction we indent them one level to the right. Hence the above algorithm could be rewritten as follows
# assert $A : (a = a0) \wedge (b = b0)$

```
    temp = a
    a = b
    b = temp
```

# assert $B : (a = b0) \wedge (b = a0)$

### 4.1.2 Assertions

The reader may have noticed that in the algorithm stated above there are statements labeled "assert" about the state of the computation. These are not instructions to be executed but are essential documentation necessary for correct design of imperative algorithms. These are *true* statements about the state of a computation. Such statements are called *assertions* or *logical propositions*. Throughout these notes we will make such assertions about the state of a computation in the imperative style description of algorithms. The pre-condition and the post-condition are assertions about the initial and the final states of the algorithm respectively. Very often they do not completely describe the state of each variable in the computation, but instead give us an abstract property of the state which should be *true*. The invariant properties described in the last chapter are also examples of such assertions. In special cases an assertion may completely describe the state as in the algorithm below in which between any two instructions there is an assertion.

# assert $A_0$: $(a = a_0) \wedge (b = b_0)$

```
    temp = a
```

# assert $A_1$: $(a = a_0 = temp) \wedge (b = b_0)$

```
    a = b
```

# assert $A_2$: $(a_0 = temp) \wedge (a = b = b_0)$

```
    b = temp
```

# assert $A_3$: $(a_0 = temp = b) \wedge (a = b_0)$

The assertions $A_1$, $A_2$ and $A_3$ completely describe the state of the computation at those points. In contrast, the post-condition, # assert $B$: $(a = b_0) \wedge (b = a_0)$, used earlier, does not completely describe the state and it is merely a *true* statement about the final state. It is obvious that from $A_3$ one may deduce $B$ and hence if $A_3$ is *true* then so is $B$.

In general, a complete description of the state may not be interesting, relevant or even known to us at any stage of a computation. Hence assertions, which are *true* statements about the state, are used to capture the essential properties of the state. Throughout these

notes we will use suitable assertions as algorithm/program documentations wherever there is a scope of ambiguity and definitely as the input/output specifications for the various modules which we will design. This will enable us to develop our algorithms in a modular fashion with the input-output specifications of the individual modules serving as the interfaces between the different modules.

**Exercise 4.1** Which of the following algorithms achieve the same result? Which of them achieves the same result as the swap algorithm defined above?

```
(a)              (b)              (c)
     temp = b         temp = b         temp = a
     b = a            a = temp         b = temp
     a = temp         b = a            a = b
```

**Exercise 4.2** Given the pre-condition $(a = a_0) \wedge (b = b_0) \wedge (c = c_0)$ write imperative algorithms to achieve the following post-conditions:

1. $(a = b_0) \wedge (b = c_0) \wedge (c = a_0)$
2. $(a = c_0) \wedge (b = a_0) \wedge (c = b_0)$

**Exercise 4.3** Given the pre-condition $(a = a_0) \wedge (b = b_0)$ where $a$ and $b$ are integers, indicate the state changes (using suitable assertions) that take place in the following algorithm. What is the final state?

```
a = a + b
b = a - b
a = a - b
```

In what follows we introduce the *if then else* and the *while do* instructions which provide the basic mechanisms for the flow of control in an imperative style algorithm.

### 4.1.3 The *if then else* instruction

The *if then else* instruction is the basic tool for decision making in imperative style programming. It is a directive for executing one out of two possible sequences of instructions depending on a logical condition. The structure of the *if then else* is

```
if (C):
   s1
else:
   s2
```

Here $s1$ and $s2$ may either be simple instructions (e.g., a single assignment or another *if then else* instruction or a *while-do* instruction which we will introduce shortly), or they may even be compound instructions indented together.

The condition ($C$) is a statement about the state of the computation at that point. However $C$ is not an assertion because it is not necessarily a *true* statement about the state. If the condition $C$ is *true* then the state changes proposed in $s1$ are performed and if $C$ is *false* then the state changes proposed in $s2$ are performed. Given that the pre-condition of the *if then else* instruction is an assertion $A$, we may rewrite it as follows by inserting appropriate assertions

# assert $A : \ldots$

```
if (C):
```
    # assert $A \wedge C$
```
    s1
else:
```
    # assert $A \wedge \neg C$
```
    s2
```

Note that if $C$ is *true* in the state before the *if then else* instruction is performed then we may assert that both $A$ and $C$ are *true*. However if $C$ is *false*, then we assert that both $A$ and $\neg C$ are *true* (as given after the "else").

A special case of the *if then else* is the *if then* instruction

```
if (C):
    s1
```

In this case $s1$ is executed if the condition $C$ is *true*, else nothing is done. The following two examples illustrate the use of such an instruction.

**Example 4.2** *Swap the values of variables a and b (which are of the same type) if* $a > b$. Let *temp* be a variable of the same type as $a$ and $b$. The algorithm can be given as

# assert $A : (a = a0) \wedge (b = b0)$

```
if (a > b):
```
    # assert $A \wedge (a > b)$
```
        temp = a
        a = b
        b = temp
```
# assert $B : ((a0 <= b0) \wedge (a = a0) \wedge (b = b0)) \vee ((a = b0) \wedge (b = a0))$

Note the use of indentation to ensure that the entire sequence is to be regarded as a single (compound) instruction if the condition $a > b$ is true.

**Exercise 4.4** Consider the following sequence of instructions
# assert $A : (a = a0) \wedge (b = b0)$

```
if (a > b):
      temp = a
      a = b
      b = temp
```

Write assertions after each step of the above sequence of instructions starting from the pre-condition.

**Example 4.3** *The following* if then *instruction ensures that $x$ is always non-negative.*

# assert $A : (x < 0) \vee (x \geq 0)$

```
if (x < 0):
```
   # assert $A \wedge (x < 0)$
```
   x = -x;
```
# assert $B : (x \geq 0)$

In what follows we give a few more examples of case analysis using the *if then else*.

**Example 4.4** *Finding the roots of a quadratic equation of the form $ax^2 + bx + c = 0$.*
Here $a$, $b$ and $c$ are real valued coefficients. We assume that $a \neq 0$. The variables $r1$ and $i1$ should contain the real and the imaginary parts of the first root, and the variables $r2$ and $i2$ should contain the real and the imaginary parts of the second root.
We describe the algorithm, in terms of the variables $a$, $b$, $c$, $d$ and $e$ of the type real as follows

# assert $A : (a = a0 \neq 0) \wedge (b = b0) \wedge (c = c0)$

```
d = b*b - 4*a*c;
if (d >= 0):
```
   # assert $A \wedge (d \geq 0)$
```
      e = sqrt(d)
      r1 = (-b+e)/(2*a)
      r2 = (-b-e)/(2*a)
      i1 = 0
      i2 = 0;
else:
```
   # assert $A \wedge (d < 0)$
```
      e = sqrt(-d)
      r1 = -b/(2*a)
      r2 = r1
      i1 = e/(2*a)
      i2 = -e/(2*a)
```

$$\begin{aligned}\#\text{ assert } B : [&(a = a0) \wedge (b = b0) \wedge (c = c0) \wedge (d = (b^2 - 4ac)] \bigwedge \\ &[\ ((d \geq 0) \wedge (r1 = (-b + \sqrt{d})/2a) \wedge \\ &\quad (r2 = (-b - \sqrt{d})/2a) \wedge (c1 = c2 = 0)) \bigvee \\ &\quad ((d < 0) \wedge (r1 = r2 = -b/2a) \wedge (c1 = \sqrt{|\ d\ |}/2a) \wedge \\ &\quad (c2 = -\sqrt{|\ d\ |}/2a)) \\ &]\end{aligned}$$

**Example 4.5** *Determining whether a given month and day represent a valid date.*
Given that $m$ and $d$ are integer type variables and the pre-condition $(m = m0) \wedge (d = d0)$, establish whether $m$ and $d$ together give a valid day of a year ($m$ gives the month and $d$ gives the day). Set an integer variable *valid* to 1 if they represent a valid day and set it to 0 otherwise.

We describe the algorithm as follows.

# assert $A : (m = m0) \wedge (d = d0)$

```
if ((m < 1) or (m > 12) or (d < 1) or (d > 31)):
   valid = 0
else:
```

# assert: $(1 \leq m \leq 12) \wedge (1 \leq d \leq 31)$

```
   if ((m = 1) or (m = 3) or (m = 5) or (m = 7) or (m = 8) or (m = 10)
       or (m = 12)):
```

# assert: $(m \in \{1, 3, 5, 7, 8, 10, 12\}) \wedge (1 \leq d \leq 31)$

```
      valid = 1
   else:
```

# assert: $m \in \{2, 4, 6, 9, 11\} \wedge (1 \leq d \leq 31)$

```
      if ((m = 4) or (m = 6) or (m = 9) or (m = 11)):
```

# ssert: $m \in \{4, 6, 9, 11\} \wedge (1 \leq d \leq 31)$

```
         if (d < 31):
```

/* assert: $m \in \{4, 6, 9, 11\} \wedge (1 \leq d \leq 30)$ */

```
            valid = 1
         else:
```

# assert: $m \in \{4, 6, 9, 11\} \wedge (d = 31)$

```
            valid = 0
      else:
```

# assert: $(m = 2) \wedge (1 \leq d \leq 31)\}$

```
         if (d < 30):
```

# assert: $(m = 2) \wedge (1 \leq d \leq 29)$

```
        valid = 1
    else:
        # assert: (m = 2) ∧ (30 ≤ d ≤ 31)
        valid = 0
```

# assert: $[(valid = 1) \wedge B] \bigvee [(valid = 0) \wedge \neg B]$
where,

$$\begin{aligned} B \quad = \quad & ((m \in \{1,3,5,7,8,10,12\}) \wedge (1 \leq d \leq 31)) \vee \\ & ((m \in \{4,6,9,11\}) \wedge (1 \leq d \leq 30)) \vee \\ & ((m = 2) \wedge (1 \leq d \leq 29)) \end{aligned}$$

**Exercise 4.5** Given that $y$, $m$ and $d$ are integer type variables and the pre-condition $(y = y0) \wedge (m = m0) \wedge (d = d0)$, establish whether $y$, $m$ and $d$ together give a valid date. Set an integer variable $valid$ to 1 if they represent a valid date and set it to 0 otherwise.

In our examples using the *if then else* instruction, we have made assertions at every stage of the decision making. However, in future, for the sake of brevity, we will make suitable assertions only at places where there is some scope of ambiguity.

### 4.1.4 The *while do* instruction

Apart from the assignment of values to variables and the *if then else* we need a construct for carrying out *iteration* in the imperative model. A careful look at the iterative process for computing $factorial(n)$ (Example 3.17) tells us that the final result is obtained through a sequence of state changes of the variables $f$ and $c$. The evolution of the states of these two variables can be described as

$$\begin{aligned} f = 1 \quad & \text{and} \quad c = 0 \\ f = 1 \quad & \text{and} \quad c = 1 \\ f = 2 \quad & \text{and} \quad c = 2 \\ f = 6 \quad & \text{and} \quad c = 3 \\ f = 24 \quad & \text{and} \quad c = 4 \\ & \vdots \end{aligned}$$

till $c = n$. To describe the process in the imperative model we would have to start with

```
f = 1
c = 0
```

and repeat the following instruction till $c = n$

```
if (c <> n):
   f = f * (c+1)
   c = c + 1
```

Thus we will have to put at least $n$ such *if then else* instructions in a sequence. Since $n$ is not known in general, we require a finite and compact representation of such repetitive operations. The primary iterative construct in this model which does this is the *while do*. The instruction

```
while (C):
    s1
```

repeats $s1$ while the boolean condition $C$ is *true*. Thus the instruction $s1$ is executed if the condition $C$ is *true* to start with. After each execution of the instruction $s1$ the condition $C$ is evaluated again to determine whether it is *true* or not. The process is repeated if $C$ is *true*; otherwise the *while do* instruction is terminated. As in the case of *if then else*, $s1$ may either be a simple instruction or a compound instruction. Each of these instructions, in turn, can be one of the three types - *assignment, if then else or while do.*

Since the purpose of the *while do* instruction is to represent iterative processes, we must associate an *invariant* condition with every *while do* loop. This invariant condition must hold *true* every time the condition $C$ is evaluated. We may rewrite the *while do* instruction with its associated assertions as

```
# assert: I
while (C):
    # assert I ∧ C
        s1
        # assert I
 # assert: B = I ∧ ¬C
```

Thus, the invariant assertion, $I$, must be *true* the first time the *while do* instruction is encountered. This has to be ensured though a proper initialization process. If the condition $C$ is *true*, then the assertion $I \wedge C$ must be *true* before the state changes proposed in $s1$ are carried out every time during the iterative process. The invariant condition $I$ along with the condition $C$ may thus be looked upon as the specification for the design of *while do* instruction. Finally, when the loop is terminated, the condition $I \wedge \neg C$ gives the final desired state.

To see an example of algorithm design using the *while do instruction*, let us consider an iterative version of the factorial computation in the imperative style.

**Example 4.6** *Iterative computation of $factorial(n)$ for $n \geq 0$ in the imperative style.* We can first describe an iterative algorithm for this problem in the functional style with the invariant condition $I = (0 \leq count \leq n) \wedge (fact = count\,!))$ as follows

$$factorial(n) = fact\_iter(n, 1, 0)$$

where,

$$fact\_iter(n, fact, count) = \begin{cases} fact & \text{if } count = n \\ fact\_iter(n, fact * (count + 1), count + 1) & \text{otherwise} \end{cases}$$

we can then translate the above algorithm in the imperative style using the *while do* instruction as

```
# assert A : n ≥ 0}
count = 0
fact = 1
# assert I : (0 ≤ count ≤ n) ∧ (fact = count!)
while (count != n):
    # assert I ∧ (count ≠ n)
        count = count + 1
        fact = fact * count
        # assert I
# assert I ∧ (count = n)
#assert B : fact = n! */
```

It is important to note that every tail-recursive functional algorithm can be represented using a *while-do* loop as in the above example. The auxiliary variables of the tail-recursive function (e.g.. $fact$ and $count$) become local variables of the imperative algorithm, and both the tail-recursive function and the *while do* loop are described in terms of the same invariant. The condition of the *while-do* loop becomes the complement of the termination condition of the tail-recursive function.

### 4.1.5 Functions and procedures in the imperative model

While designing an algorithm to perform certain computation it may often be necessary for different parts of the computation to share some components. For example, consider an algorithm that needs to compute $factorial(m)$ and $factorial(n)$ for different values of $m$ and $n$. Rather than duplicating the imperative algorithm for factorial computation at two different places, it is more convenient to have a separate function for the sub-algorithm for computing factorial of a number, which may then be invoked with different inputs $m$ and $n$. In what follows we give two imperative functions for factorial computation.

**Example 4.7** *Imperative style functions for computing* $factorial(n)$.

An imperative style function for the factorial computation using the recursive algorithm of Example 3.8 can be written using `Python` syntax as

⟨*Factorial (recursive)*⟩≡

```
def factorial(n):
# assert: (n >= 0)
        if (n == 0):
                return 1
        else:
                return (n * factorial(n-1))
```

`Python` does not require explicit type specification and infers type dunamically. So, it is the responsibility of the programmer to ensure type discipline. In this case the function declaration `def factorial(n)` assumes that the function `factorial` can be invoked with an integer input and that it returns an integer value.

Note, however, that the imperative style function defined above is fundamentally different from its corresponding functional definition given in the last chapter, though they use the same algorithm. Here `n` is an imperative style variable which stores a value. An invocation of the function from the calling environment with

```
a = factorial(5)
```

assigns the value 5 to `n`. The function returns an integer value through its name `factorial`.

Alternatively we can write an imperative style function using the iterative algorithm of Example 3.17 as

⟨*Factorial (iterative)*⟩≡

```
def factorial(n):
# assert: (n >= 0)
        f,c = 1,0 #assign 1 to f and 0 to c
        #INV: (0 <= c <= n) and (f = c!) and (n! = f * (c+1)*(c+2)...*n)
        while (c != n):
                c = c + 1
                f = f * c
        # assert: (f = n!)
        return f
```

This function can also be invoked in the same way. This function uses its own local variables `f` and `c` which are not visible from the calling environment.

In some cases it may not be necessary to return a value through the function name. Rather we may wish to define a generic algorithm which effects some state changes. In such cases we call it a *procedure*.

**Example 4.8** *Swapping the values of two variables.*

A procedure for swapping the values of two global variables `a` and `b` can be defined as follows

⟨*Swap*⟩≡

```
def swap():
        # assert: (a = a0) and (b = b0)
        global a,b
        t = a
        a = b
        b = t
        # assert: (a = b0) and (b = a0)
```

Note that the procedure swap defined above does not have any formal input passed from the calling program as an argument. Neither does it return any output. It merely affects some state changes of the globally defined variables `a` and `b`

```
a,b = 3,4
swap();
/* assert: (a = 4) and (b = 3) */
```

The input parameters to a procedure or a function are specified within brackets in the declaration. These parameters are called the *formal parameters* of a procedure or a function. For example, if a function or procedure is invoked from the calling environment with an instruction like `a = factorial(b)`, the formal parameter `n` is initialized by copying the value of the variable `b`. The effect is similar to that of an assignment like `n = b`. Thus, in this case, `n` and `b` are two different variables, and even if `n` is modified in the function the variable `b` in the calling environment remains unaffected. The variable `n` is local to the function and is not accessible from the point of invocation in the calling environment.

In what follows we describe a complete `Python` program for computing $factorial(n)$ using the iterative method.

**Example 4.9** *A complete* `Python` *program for computing factorial of a number.*

A complete `Python` program for computing factorial using the iterative method is given as follows:

⟨*Complete program*⟩≡

```
def factorial(n):
# assert: (n >= 0)
        f,c = 1,0
        #INV: (0 <= c <= n) and (f = c!) and (n! = f * (c+1)*(c+2)...*n)
        while (c != n):
                c = c + 1
                f = f * c
        # assert: (f = n!)
        return f

a = int(input("Input an integer: "))
print(factorial(a))
```

## Problems

1. Develop `Python` programs/functions for each of the problems in the last Chapter.
2. Write a `Python` function to find the minimum of three input integers.
3. Develop `Python` programs for the following
   (a) Given three points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ to determine whether they are collinear, and if so which point lies between which two.
   (b) Given the coordinates of the centers of two circles and their radii, find whether they intersect, and if so find the points of their intersection.
   (c) Given the coordinates of four points on a plane, determine whether they form a quadrilateral, and if so classify it if possible as a square, rectangle, rhombus, parallelogram etc.
4. Develop a `Python` function which reads $n$ integers from the input and returns the maximum.
5. Given the coordinates $(x_0, y_0)$ of the center of a circle and a point $(x_1, y_1)$ lying on its circumference, develop a `Python` program that outputs the coordinates $(x_1, y_1), \ldots, (x_6, y_6)$ of the vertices of a regular hexagon inscribed in the circle.
6. Develop a `Python` program to read a sequence of digits, terminated by a non-digit character, and to find the value of the integer the sequence represents.
7. An automatic cash register is given as input
   (a) the cost of the customer's purchases (in Rupees)
   (b) the amount of money the customer has given (in Rupees)

   It outputs the balance the customer should be paid indicating how many notes of each denomination (Rs. 100, Rs. 50, Rs. 20, Rs. 10, Rs. 5, Rs. 2, Rs. 1) should be paid so that the minimum number of notes are given to the customer. Develop an iterative `Python` program to solve this problem. Assume that all costs and notes are integral multiple of the Rupee.
8. Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial. Assuming that the input is received in the order
   $$x, n, a_0, a_1, \ldots, a_n$$
   develop a `Python` program to do the following tasks.
   (a) evaluate the polynomial in $O(n)$ time
   (b) evaluate both the polynomial and its derivative in $O(n)$ time

# Chapter 5

# Step-wise refinement and Procedural Abstraction

Now that we have studied the essentials of the functional and the imperative models of computation, we are ready to develop more complex programs. In this chapter we will study the methodology of developing complex programs using step-wise refinement, procedural abstractions and higher order functions.

## 5.1 Step-wise refinement

The primary objective in computational problem-solving is to develop a correct and efficient computer program for solving the problem at hand. In this chapter we will illustrate how, given a problem definition, a precise algorithm and program implementation can be developed, in a step-wise manner, from an *outline* of the solution using a powerful method called *step-wise refinement* or *top-down design*. The objective of the design methodology is to first establish the overall structure and the relationships between the various parts of the problem, and then address the specific and complex issues of the implementations of the various sub-parts.

The main phases of computer based problem solving are:

**Outline of solution:** which identifies the basic principle by which the input can be transformed to the output and gives an outline of the solution.

**Algorithm design and analysis:** which makes precise the outline indicated, identifies the various components required and gives a precise method of computing the solution. A crucial aspect in algorithm design is the analysis of correctness and efficiency.

**Program design:** which involves an implementation of the algorithm using the syntax of a programming language.

We have already seen an example of step-wise refinement in Example 3.13. The problem of deciding whether a given positive integer is a perfect number or not was decomposed

into many sub-problems; and separate functional algorithms were then written for each of those. The main idea behind the decomposition strategy is not merely to divide a program into parts, but to ensure that each sub-problem accomplishes a clearly identifiable task and is written as a separate function or procedure which can then be used as a module to define other functions. For example, the function $perfect$ was defined in terms of the function $addfactors$, and while describing the function $perfect$ we were able to regard $addfactors$ as a black-box. Thus the actual design of the function $addfactors$ could be postponed till later and we were only concerned with what it computes and not how it computes. Thus, as far as the function $perfect$ is concerned, $addfactors$ is not really a procedure but a *procedural abstraction*. Procedural abstraction is an integral part of step-wise refinement and it allows program development in a modular and systematic fashion.

### 5.1.1 Executable specifications and rapid-prototyping

In our method of step-wise refinement we will first develop our algorithms in the functional model of computation, primarily because it is closer to mathematics and it is easier to analyze correctness and efficiency in the functional model. It is also easier, in general, to identify the various sub-parts required in the functional model resulting in greater modularity in the algorithm design. The functional description can then serve as the specification for the imperative program, and we can translate the functional description into an imperative `Python` program. If the functional description is also coded in `ML` then the `ML` program can serve as an *executable specification* for the imperative program. In such a case, since a `ML` program is easier to implement, the executable specification can be thoroughly tested before undertaking the design of the corresponding `Python` program. The `ML` interactive environment can then serve as an environment for rapid-prototyping.

A rapid-prototyping environment allows for the following:

1. A direct translation of a mathematical specification of a computation into an executable form. Executable specifications are written in a language which is very similar to normal mathematical notations. Hence proofs of correctness and analysis of efficiency can be carried out easily.

2. An interactive mode of program development, testing and verification. An interactive mode also allows for testing, debugging and frequent changes to proceed concurrently.

3. A finer analysis of both correctness and efficiency of the possible algorithms that may be candidate solutions to a problem.

Functional programs are small in size even though they may represent complex algorithms and may take a long time to execute with realistic input. The small size of functional code and the lack of fine details which are not immediately relevant allows for shorter development times. Hence a functional programming language like `ML` with its interactive mode of computation provides an excellent environment for rapid-prototyping of programs and allows specifications to be executed, tested and even improved before coding it up in a more efficient imperative programming language.

### 5.1.2 Examples of step-wise refinement

In what follows we illustrate the methodology of step-wise refinement through a few examples.

**Example 5.1** *Determining whether a given positive integer $n$ is a prime (Method 1).*
**Outline of the solution:** A positive integer $p$ is a prime if its only positive divisor other than 1 is $p$ itself. Hence 1 is not a prime. Since all even numbers are divisible by 2, the only even prime is 2 itself. Hence a prime other than 2 is necessarily an odd number. If $n$ is an odd number, we can determine whether $n$ is a prime by testing the divisibility of $n$ by test divisors chosen successively from the sequence $3, 5, 7, 9 \cdots$ till one of the following conditions is realized:

1. A test divisor divides $n$ exactly; in such a case $n$ is not a prime.
2. The square of the test divisor exceeds $n$. In such a case $n$ does not have a positive divisor other than 1 and $n$, and, consequently, $n$ is a prime.

**Algorithm design and analysis:** We will first develop a functional algorithm based on the above computational strategy. The function we are seeking is of the type $prime : \mathbb{P} \rightarrow \{true, false\}$. We can give a top level description of the function as

$$prime(n) = (n = 2) \vee ((odd(n) \wedge (smallest\_divisor(n) = n)))$$

Note that we have given a top-level description of the algorithm in terms of the procedural abstractions *odd* and *smallest_divisor*. We may now design algorithms for these two procedures after deriving their exact specifications from the above description.

The function $odd : \mathbb{P} \rightarrow \{true, false\}$ can be defined as

$$odd(n) = ((n \ mod \ 2) = 1)$$

The function $smallest\_divisor(n)$ should return the smallest divisor of $n$ in the interval between $2 \ldots n$. We can compute the smallest divisor of $n$ using an iterative algorithm whose state space is described in terms of two variables - $n$ and *test_divisor*. An invariant condition for the iterative algorithm can be described as

$$\begin{aligned} INV \quad = \quad & (3 \leq test\_divisor \leq \lfloor \sqrt{n} \rfloor + 2) \\ & \wedge \\ & (\text{no odd integer in the interval } [3 \ldots (test\_divisor - 2)] \text{ divides } n) \end{aligned}$$

We can then define the function $smallest\_divisor : \mathbb{P} \rightarrow \mathbb{P}$ as

$$smallest\_divisor(n) = find\_iter(n, 3)$$

where the auxiliary function $find\_iter : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ can be defined according to the above invariant condition as

$$find\_iter(n, test\_divisor) = \begin{cases} n & \text{if } (test\_divisor^2 > n) \\ test\_divisor & \text{if } ((n \ mod \ test\_divisor) = 0) \\ find\_iter(n, test\_divisor + 2) & \text{otherwise} \end{cases}$$

The correctness of the above algorithm can be established from the invariant condition. Alternatively, the correctness of the iterative algorithm can also be established using **PMI**.

**Exercise 5.1** For the above functional algorithm

1. Verify that the iterative procedure satisfies the invariant condition given above. In particular verify that the invariant condition holds initially, and the desired final result can also be obtained from the invariant condition.

2. Establish the correctness of the above algorithm using **PMI**.

Since the algorithm tests for divisors between 1 and $\sqrt{n}$ the time complexity of the algorithm is clearly $O(\sqrt{n})$. The space complexity of the iterative process in $O(1)$.

**Program development:** The algorithm developed above can now be treated as a specification for the imperative `Python` program. We can first translate the above functional description into a `ML` program which can then be tested before designing the corresponding `Python` program.

⟨*Prime-test 1 (ML)*⟩≡
```
fun prime(n) =
        let
                ⟨Code for smallest_divisor⟩
                fun odd(n) = (n mod 2 = 1)
        in
                (n=2) orelse (odd(n) andalso (n = smallest_divisor(n)))
        end;
```

⟨*Code for* `smallest_divisor`⟩≡
```
fun smallest_divisor(n) =
        let
                ⟨Code for find_divisor⟩
        in
                find_divisor(n,3)
        end;
```

⟨*Code for* `find_divisor`⟩≡
```
fun find_divisor(n,test_divisor) =
        let
                fun square(x:int) = x*x
        in
                if square(test_divisor) > n then
                        n
                else if (n mod test_divisor) = 0 then
                        test_divisor
                else
                        find_divisor(n, test_divisor + 2)
        end
```

The ML program given above can be thoroughly tested. In fact, if the ML program is written by defining every component function at the top-level (i.e., by not defining a function within the scope of another function as suggested above), then each component function can also be tested individually. Thus any mistake, due to oversight at the algorithm design phase, can be removed by testing the functional code.

Experiments can also be conducted to verify the expected time complexity of the algorithm. One can use the special `timer` function available in the ML basis library to obtain the evaluation time of a function along with the overheads for garbage collection [1]. The overall execution time minus the the garbage collection time and the system time (this is the time spent in input/output) gives a reasonable estimate of the run-time of the program. Note that the run-time thus obtained is not a very accurate measure because of other system overheads but suffices for rough profiling of functional programs.

**Exercise 5.2** Since the algorithm for primality testing described above has a time complexity of $O(\sqrt{n})$, the time taken to test the primality of a number $\approx 10m$ should be about three times the time taken to test the primality of $m$. Experiment to find out whether this is indeed true. Explain any discrepancy that you may observe. The experiment may be conducted on the numbers 21893 and 218947, both of which are primes.

A Python program for the above can be written as
⟨*Prime-test (Python)*⟩≡

⟨*Code for* `prime`⟩

```
n = int(input("Input an integer: "))
if (prime(n)):
        print str(n)+" is a prime"
else:
        print str(n)+" is not a prime"
```

[1] Garbage collection is a part of the memory management of the ML run-time system, which identifies and recycles memory space which is not required any more. Hence different runs of the same program with the same inputs could give you different run-times, depending upon how often the garbage collector is called in the interactive mode.

⟨*Code for* `prime`⟩≡

```
def smallest_divisor(n):

        test_divisor = 3
        # assert: INV
        while ((test_divisor*test_divisor <= n) and ((n % test_divisor) != 0)):
                test_divisor = test_divisor + 2;
        # assert: INV and
        #       ((sqr(test_divisor) > n) OR ((n mod test_divisor) = 0))

        if ((n % test_divisor) == 0):
                return test_divisor
        else:
                return  n

def prime(n):
        return ((n ==2) or ((n%2 ==1) and (smallest_divisor(n) == n)))
```

Note that in the `Python` implementation we have replaced the tail-recursive algorithm for computing the smallest divisor with a while loop and we have used the same invariant condition for the tail-recursive function and the while loop. The condition of the while loop is the same as the *otherwise* clause of the tail-recursive procedure. The extra variable `test_divisor` required to define the state space is now a local variable of the function `smallest_divisor` and it has been initialized before the while loop. As we have mentioned before, a tail-recursive function can always be directly translated into an imperative procedure described by a while loop.

To further illustrate the methodology of step-wise refinement we will develop an alternative method for primality testing using a different computational theory.

**Example 5.2** *Determining whether a given positive integer n is a prime (Method 2).*

**Outline of the solution:** Our new method of primality testing is based on a result in number theory known as the Fermat's little theorem.

**Fermat's little theorem:** If $n$ is a prime number and $a$ is any positive integer less than $n$, then $a$ raised to the $n^{th}$ power is congruent to $a$ modulo $n$.

Two numbers are said to be *congruent modulo* $n$ if they both have the same remainder when divided by $n$. The remainder of a number $a$ when divided by $n$ is also referred to as $a$ *modulo* $n$.

If $n$ is not a prime, then, in general, most of the numbers $a < n$ will not satisfy the above relation. Thus, given a number $n$, we can pick a random number $a < n$ and the compute the remainder $a^n$ *modulo* $n$. If this is not equal to $a$, $n$ is certainly not a prime. Otherwise, chances are good that $n$ is a prime. We can assume that the probability that $n$ is a prime is 0.5. We can keep repeating the above experiment and stop if either

1. at any stage we find that $n$ is not a prime, or
2. we find that the probability that $n$ is not a prime has decreased to an acceptable level. Note that with the successive experiments, the probability that $n$ is not a prime decreases as $0.5, 0.25, 0.125 \ldots$.

Unfortunately, there are numbers which fool Fermat's test. These numbers are called Carmichael numbers. Little is known about these numbers except that they are extremely rare. There are 16 Carmichael numbers below 100,000. The smallest few are 561, 1105, 1729, 2821 and 6601. The algorithm fails for Carmichael numbers but we will ignore this fact in this example[2]. Note that whenver the algorithm outputs composite it does so with certainty, but when it declares a number as prime it does so with a possibility of error. The probability of the error however goes down with the number of iterations. In fact, for a given upper bound $\epsilon \approx 2^{-k}$, we require only $O(k)$ iterations. Such algorithms – whose definitions are based on random numbers – are called randomized algorithms.[3].

**Algorithm design and analysis:** We can now develop a functional algorithm for primality testing based on the above computational theory. We start by defining an iterative function $prime(n, q)$ of the type $prime : \mathbb{P} \times \mathbb{N} \rightarrow \{true, false\}$, where $n$ is the number whose primality is to be tested and $q$ is the maximum number of times the Fermat's test is to be applied.

$$prime(n, q) = \begin{cases} true & \text{if } n = 2 \\ prime\_test(n, q, false) & \text{otherwise} \end{cases}$$

[2]The interested student may study the Miller-Rabin primality test which is not fooled by Carmichael numbers

[3]People who are not satisfied with the validity of Fermat's test because of the possibility of incorrect results may consider the following fact. In testing the primality of very large numbers chosen at random, the chance of stumbling upon a number that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm.

where the iterative function $prime\_test : \mathbb{P} \times \mathbb{N} \times \{true, false\} \rightarrow \{true, false\}$ is defined using the invariant

$$INV = (\neg failed = (n \text{ has passed Fermat's test } (q_0 - q) \text{ times}))$$

(where $q_0$ is the initial value of $q$) as

$$prime\_test(n, q, failed) = \begin{cases} true & \text{if } q = 0 \\ false & \text{if } failed \\ prime\_test(n, q-1, \neg Fermat\_test(n)) & \text{otherwise} \end{cases}$$

where, the function $Fermat\_test(n)$ applies the Fermat's test on the number $n$ once. Note that $prime\_test$ returns $false$ if the Fermat's test fails even once. We can define the function $Fermat\_test : \mathbb{P} \rightarrow \{true, false\}$ as

$$Fermat\_test(n) = (expmod(a, n, n) = a)$$

where $a$ is a random number between 2 and $n-1$ and $expmod(b, e, m)$ computes $b^e$ *modulo* $m$.

We can, to start with, define the function $expmod : \mathbb{P} \times \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ in terms of the fast exponentiation function $power(b, e)$ of Example 3.9 as

$$expmod(b, e, m) = power(b, e)\ mod\ m$$

Since the time complexity of the function $power(b, e)$ is $O(\lg e)$, we can expect to compute $expmod$ also with $O(\lg e)$ multiplications. Note, however, that the unit cost here is a multiplication. However, multiplying, say, 423223 with 378127 is far more complex than multiplying, say, 23 with 35. Thus, the cost of multiplication itself depends on the number of digits of the multiplicands. In order to compute $b^e$ (or, $a^n$) with the above algorithm, for large $n$, we have to multiply large numbers, thereby increasing the cost of the multiplications. Hence, even though the time complexity of the algorithm is $O(\lg n)$ multiplications, the cost of each multiplication cannot be treated as $O(1)$.

We can speed up the process by observing that

$$x * y\ mod\ m = ((x\ mod\ m) * (y\ mod\ m))\ mod\ m$$

Thus, if we take the *mod* operation inside the scope of *square* in the fast exponentiation algorithm of Example 3.9, then we can get by without ever having to multiply numbers larger than $m$. This gives us the following algorithm for $expmod$:

$$expmod(b, e, m) = \begin{cases} 1 & \text{if } e = 1 \\ (expmod(b, (e\ div\ 2), m))^2\ mod\ m & \text{if } even(e) \\ (b * expmod(b, (e-1), m))\ mod\ m & \text{otherwise} \end{cases}$$

**Exercise 5.3** Establish the correctness of the $expmod$ function defined above by the following steps:

1. Show that for $x, y, m \in \mathbb{P}$, $(xy \ mod \ m) = (x \ mod \ m)(y \ mod \ m)$.

2. Prove the correctness of the *expmod* function using **PMI**.

It is important to note that though we can establish the correctness of $Fermat\_test$, the correctness of the overall algorithm cannot be established because the algorithm is *not correct* for primality testing owing to the existence of the Carmichael numbers. Theoretically this remains an inexact method for computing primality.

**Exercise 5.4** Show that the time complexity of the *expmod* function defined as above is $O(\lg e)$.

Consequently, the Fermat's test can be conducted once in $O(\lg n)$ time. Thus the time complexity of the overall algorithm is $O(q \lg n)$ where $q$ is maximum number of times Fermat's test has to be applied.

**Exercise 5.5** Given that $\epsilon$ is the acceptable probability of error in declaring a number $n$ as a prime, find out $q$, the number of times Fermat's test must be executed, in terms of $\epsilon$. Assume that if the Fermat's test succeeds once, the chances are even that $n$ is a prime. Ignore the existence of numbers which may fool Fermat's test.

Actually, the probability that a number $n$ is prime if it passes the Fermat's test once is more than even, and, consequently, only a few tests will suffice. Even if the probability is 0.5, the expected number of times the test has to be conducted to determine whether $n$ is composite is only 2. Thus $q$ can be taken as a constant and the average time complexity of the overall algorithm is only $O(\lg n)$.

**Program development:** We will first develop a `ML` program for the above algorithm. We will use an `ML` library `Random` to generate the random numbers necessary for the Fermat's test.

⟨*Prime-test 2 (ML)*⟩≡

```
fun prime(n,q) =
        let
                ⟨Code for prime_test⟩
        in
                if (n = 2) then
                        true
                else
                         prime_test(n,q,false)
        end;
```

⟨*Code for* `prime_test`⟩≡
```
fun prime_test(n,q,failed) =
        let
                ⟨Code for Fermat_test⟩
        in
                if q = 0 then
                        true
                else if failed then
                        false
                else
                        prime_test(n,q-1,not(Fermat_test(n)))
        end
```

⟨*Code for* `Fermat_test`⟩≡
```
fun Fermat_test(n) =
        let
                ⟨Code for NextInt⟩
                ⟨Code for expmod⟩
        in
                let
                        val a = NextInt r;
                in
                        (a = expmod(a,n,n))
                end
        end
```

⟨*Code for* `expmod`⟩≡
```
fun expmod(b,e,m) =
        let
                fun sqr(x) = x*x : int;
        in
                if e = 0 then
                        1
                else if (e mod 2) = 0 then
                        sqr(expmod(b,e div 2,m)) mod m
                else
                        ((b mod m) * expmod(b,e-1,m)) mod m
        end
```

⟨*Code for* `NextInt`⟩≡
```
val NextInt = Random.randRange (2,n-1);
val r = Random.rand (1,1);
```

**Exercise 5.6** Test each of the functions defined above separately.

**Exercise 5.7** Implement the primality test, separately, using each of the above definitions of the function *expmod*. Experimentally determine in which case you get a behaviour closer to that suggested by the $O(\lg n)$ time complexity. You can measure the times required to test the primality of 218947 and 21893, compute the ratios of the the two times and check if they are close to $(\lg 218947/\lg 21893)$. Explain any discrepancy that you may observe.

The `Python` code for the function `prime(n,q)` can be now written as follows.

⟨*Prime-test 2 (Python)*⟩≡
```
import random

⟨Code for prime⟩

n = int(input("Input n: "))
q = int(input("Input q: "))

if (prime(n,q)):
        print str(n)+" is a prime"
else:
        print str(n)+" is not a prime"
```

⟨*Code for* `prime`⟩≡
```
⟨Code for expmod (Python)⟩

⟨Code for fermat_test (Python)⟩

def prime(n, q):
        failed = False
        while (q != 0 and (not failed)):
                failed = not fermat_test(n)
                q = q-1
        return (q == 0)
```

⟨*Code for* `fermat_test (Python)`⟩≡
```
def fermat_test(n):
        a = random.randint(2,n-1) #generates a random a such that 2 <= a <= n-1
        return (expmod(a,n,n) == a)
```

⟨*Code for* `expmod (Python)`⟩≡

```
def expmod(b, e, m):
        def square(x):
                return x*x

        if (e == 0):
                return 1
        elif (e%2 == 0):
                return square(expmod(b,e/2,m)) % m
        else:
                return ((b%m) * expmod(b,e-1,m)) % m
```

The imperative function `randint` is used to generate a random number in the range $2 \ldots (n-1)$ from the standard `Python` library `random()`.

**Exercise 5.8** Try to test the primality of a large prime number (say 218947, or even larger) with your `ML` and `Python` programs. What is the largest prime for which you obtain a correct result?

The reason that the above `ML` and `Python` programs will not work correctly for large numbers has nothing to do with the correctness of the algorithm. The largest integer that any `ML` or `Python` implementations can represent is limited. In order to compute $a^n$ *modulo* $n$ using the function `expmod`, we need to compute products of numbers which may be as large as `n`. For large `n`, this product may exceed largest integers that the programming languages `ML` and `Python` can represent causing integer overflow. The interpretercan detect the overflow and give a suitable error message. We will see later how such errors can be detected and handled in both `ML` and `Python` .

**Example 5.3** *Computation of square root by Newton's method.*

**Outline of the solution:** The Newton's method for iterative computation of $\sqrt{x}$ can be described as follows.

1. Start with a suitable guess $g_0$ for $\sqrt{x}$ and iterate as given below[4].

$$g_{n+1} = \frac{1}{2}\left(g_n + \frac{x}{g_n}\right)$$

2. Stop if at any stage the relative error $\mid g_n - \sqrt{x} \mid / \sqrt{x} < \epsilon$, where $\epsilon$ is a desired level of accuracy.

**Algorithm design and analysis:** We will design an algorithm for computing the function $sqrt(x, \epsilon)$ which returns the square root of a real number $x > 0$ within a relative error of $\epsilon$.

The function we are seeking is of the type $sqrt : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

We can determine the initialization and the termination conditions from the following analysis. We have, for $n > 0$

$$g_n - \sqrt{x} = \frac{1}{2g_{n-1}}(g_{n-1} - \sqrt{x})^2$$

and

$$g_n + \sqrt{x} = \frac{1}{2g_{n-1}}(g_{n-1} + \sqrt{x})^2$$

Thus, we have

$$\frac{g_n - \sqrt{x}}{g_n + \sqrt{x}} = \left(\frac{g_{n-1} - \sqrt{x}}{g_{n-1} + \sqrt{x}}\right)^2 = \left(\frac{g_0 - \sqrt{x}}{g_0 + \sqrt{x}}\right)^{2^n} \geq 0$$

Hence $g_n \geq \sqrt{x}$ for all $n > 0$ even if $g_0 < \sqrt{x}$. We also have that

$$g_n - \sqrt{x} = 2\sqrt{x}\frac{q^{2^n}}{1 - q^{2^n}}$$

where

$$q = \frac{g_0 - \sqrt{x}}{g_0 + \sqrt{x}}$$

Thus, to obtain

$$\lim_{n \to \infty} g_n = \sqrt{x}$$

we must have $\mid q \mid < 1$ or $g_0 > 0$.

Thus, with a choice of $g_0 = 1$ we have

$$\lim_{n \to \infty} g_n = \sqrt{x} \quad \text{and} \quad g_n \geq \sqrt{x} \ \ \forall n > 0$$

[4]This is also called *Hero's algorithm.*

Also note that

$$g_{n-1} - g_n = g_{n-1} - \frac{1}{2}\left(g_{n-1} + \frac{x}{g_{n-1}}\right) = \frac{g_{n-1}^2 - x}{2g_{n-1}} > 0$$

Hence the approximations $g_n$ for $n \geq 1$ form a monotonically decreasing sequence

$$g_1 \geq g_2 \cdots \geq g_{n-1} \geq g_n \geq \cdots \geq \sqrt{x}$$

We also have from the above that

$$g_{n-1} - \sqrt{x} \leq g_{n-1} - \frac{x}{g_{n-1}} = \frac{g_{n-1}^2 - x}{g_{n-1}} = 2(g_{n-1} - g_n)$$

Hence, if $0 \leq 2(g_n - g_{n+1})/\sqrt{x} < \epsilon$ for $n \geq 2$, it is guaranteed that $0 \leq (g_n - \sqrt{x})/\sqrt{x} < \epsilon$, and we may use this condition as the stopping criterion.

Now we may design an algorithm for computing square root as follows. We may write an invariant for the iterative computation as

$$INV = [n = 0] \vee [(n > 0) \wedge (g_n - \sqrt{x} = 2\sqrt{x}\frac{q^{2^n}}{1 - q^{2^n}})]$$

where $q$ is as above. We start with $g_0 = 1$. At any stage stop if the error $2(g_n - g_{n+1})$ falls below a desired level of accuracy $\epsilon$.

The complete algorithm can then be given as

$$sqrt(x, \epsilon) = sqrt\_iter(x, \epsilon, 1, 0)$$

where the auxiliary function $sqrt\_iter : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is defined as

$$sqrt\_iter(x, \epsilon, g_n, n) = \begin{cases} g_n & \text{if } acceptable(g_n, gnew, x, \epsilon) \\ sqrt\_iter(x, \epsilon, gnew, n+1) & \text{otherwise} \end{cases}$$

where

$$gnew = update(g_n, x)$$

Note that we have given a top-level description of the algorithm in terms of the procedural abstractions *acceptable* and *update*. We may now design algorithms for these two procedures after deriving their exact specifications from the above description.

The function $acceptable : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{B}$ accepts $g_n$, $gnew$, $x$ and $\epsilon$ as input and determines whether the termination condition is satisfied. The function $update : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ updates the guess $g_n$ according to Newton's iteration formula. We may define these functions as

$$acceptable(g_n, gnew, x, \epsilon) = (4(g_n - gnew)^2 < x * \epsilon^2)$$

and

$$update(g, x) = (x/g + g)/2$$

**Correctness**
It can easily be verified that the initialization satisfies the invariant. Also the choice $g_0 = 1$ ensures that $\mid q \mid < 1$. Thus, according to the invariant the successive guesses generate a monotonically decreasing sequence, i.e.,

$$g_1 \geq g_2 \cdots \geq g_{n-1} \geq g_n \geq \cdots \geq \sqrt{x}$$

and the convergence of the process is guaranteed. The termination condition ensures that we have the solution to the desired level of accuracy.
**Efficiency**
Suppose we wish to have an accuracy up to the $k^{th}$ decimal digit, i.e.,

$$\frac{g_n - \sqrt{x}}{\sqrt{x}} \leq \epsilon = 10^{-k}$$

Then, the number of iterations, $n$, required can be estimated from the invariant condition as follows. We require that

$$\frac{g_n - \sqrt{x}}{\sqrt{x}} = 2\frac{q^{2^n}}{1 - q^{2^n}} \leq 10^{-k}$$

where $\mid q \mid < 1$ is a constant. In the asymptotic analysis the denominator term, $1 - q^{2^n}$, can be bounded by a constant, i.e., for some $n > n_0$ we must have $1 - q^{2^n} > c$ for some constant $c$ such that $0 < c < 1$. Hence, for $n > n_0$, we require that

$$2\frac{q^{2^n}}{1 - q^{2^n}} < 2\frac{q^{2^n}}{c} \leq 10^{-k}$$

Hence, the number of iterations required is

$$n = O(\lg k)$$

**Program development:** Now we are ready to translate the above algorithm description into programming syntax. The complete `ML` program can be given as follows.

⟨*Sqrt*⟩≡
```
fun sqrt(x,epsilon) =
   let
      ⟨Code for sqrt_iter⟩
   in
      sqrt_iter(x,epsilon,1.0,0)
   end;
```

⟨*Code for* `sqrt_iter`⟩≡
```
fun sqrt_iter(x,epsilon,gn,n) =
   let
      ⟨Code for acceptable⟩
      ⟨Code for update⟩
   in
      let val gnew = update(gn,x)
      in
         if acceptable(gn,gnew,x,epsilon) then
            gn
         else
            sqrt_iter(x,epsilon,gnew,n+1)
      end
   end
```

⟨*Code for* `acceptable`⟩≡
```
fun acceptable(gn,gnew,x,epsilon) =
   let
      fun sqr(x:real) = x*x
   in
      ((4.0*sqr(gn-gnew)) <= (x * sqr(epsilon)))
   end
```

⟨*Code for* `update`⟩≡
```
fun update(gn,x) = (gn + x/gn)/2.0;
```

**Exercise 5.9** Test each of the `ML` functions developed above by actual execution.

Note that in the above algorithm we have not used the variable $n$ explicitly. Its only role is to establish the correctness.

**Exercise 5.10** Modify the above code to return the pair $(gn, n) \in \mathbb{R} \times \mathbb{N}$ and verify that the number of steps required (given by $n$) to compute the square root of a number agrees with the theoretically derived time complexity.

Once we understand abstract role of the variable $n$ in the above program (and complete the actual testing of the program), we may drop it from the actual program.

**Exercise 5.11** Rewrite the functional algorithm and the `ML` program without using the iteration counter $n$ explicitly.

**Exercise 5.12** Develop a `Java` function for square root computation by translating the above `ML` program.

## 5.2 Procedural abstraction using higher-order functions

Till now we have only considered functions which accept numbers as input values and return numbers as output values. But even in numerical computations we will be severely restricted if we only consider functions of this type. There may be several computational problems whose algorithmic solutions are based on a common central idea and are similar in structure. In such a case we should be able to abstract out these common features in a high level algorithm which can then be tailored to define solutions to particular problems with these features. Such abstract functions can be written as *higher-order functions*. Higher order functions are those which can accept functions as arguments and return functions as values. In this section we will illustrate the use of such higher-order functions.

### 5.2.1 Functions as input parameters

Consider the following problems for computing three different sums.

1. Computing the sum of all integers from $a$ to $b$ (Example 3.12). A functional algorithm for the problem can be given in terms of a function $sum : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ as
$$sum(a, b) = \begin{cases} 0 & \text{if } (a > b) \\ a + sum(a + 1, b) & \text{otherwise} \end{cases}$$

2. Computing the sum of squares of all integers from $a$ to $b$. A functional algorithm for the problem can be given in terms of the function $sum\_squares : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ as
$$sum\_squares(a, b) = \begin{cases} 0 & \text{if } (a > b) \\ square(a) + sum\_squares(a + 1, b) & \text{otherwise} \end{cases}$$

3. Computing the sum of a sequence of terms in the following series which converges to $\pi/8$:[5]
$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \ldots$$
A functional algorithm for the problem can be given in terms of the function $pi\_sum : \mathbb{N} \rightarrow \mathbb{R}$ as
$$pi\_sum(n) = \begin{cases} 0 & \text{if } (a > b) \\ (1/(a * (a + 2))) + pi\_sum(a + 4, b) & \text{otherwise} \end{cases}$$

Clearly, the three algorithms share a lot in common; so much so that they warrant the design of a common function which combines the common characteristics of the three different different functions. This can be achieved by defining a generic summation whose functionality is given by
$$\sum_{x=a, succ(x)}^{b} f(x)$$

[5]This series, usually written as $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$, is due to Leibniz

i.e., the summation of an arbitrary function $f(x)$ in the range $[a, b]$ in steps defined by the successor function $succ(x)$. In order to write a functional algorithm for such a generic summation one needs to be able to accept two functions $f : \mathbb{N} \to \alpha$ and $succ : \mathbb{N} \to \mathbb{N}$ as input in addition to the parameters $a$ and $b$ which belong to the set $\mathbb{N}$. Here $\alpha$ may be any generic type on which the operation $+$ is defined (e.g. $\mathbb{N}$ or $\mathbb{R}$). Hence, the generic summation must be a *higher-order* function whose *type* is

$$summation : \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \to \alpha) \times (\mathbb{N} \to \mathbb{N}) \to \alpha$$

A functional algorithm for the generic summation can then be given as

$$summation(a, b, f, succ) = \begin{cases} 0 & \text{if } (a > b) \\ f(a) + summation(succ(a), b, f, succ) & \text{otherwise} \end{cases}$$

The advantage of defining such a higher-order function independent of any particular problem is that the analysis of correctness and efficiency of the algorithm can be carried out in a general setting.

**Exercise 5.13** For the function *summation*

1. Establish the correctness assuming the correctness of the functions $f$ and *succ*. Note that $f$ and *succ* are just procedural abstractions of two unknown functions.
2. Determine the time and the space complexity in terms of number of calls to the functions $f$ and *succ*.

In `ML` the higher-order summation function can be written as follows:

⟨*summation*⟩≡
```
fun summation(a:int,b,f,succ) =
   if (a > b) then
      0
   else
      f(a) + summation(succ(a),b,f,succ);
> val summation = fn : int * int * (int -> int) * (int -> int) -> int
```

Then the summation functions *sum* and *sum_squares* of the type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ can be defined in terms of the higher-order summation function as follows:

⟨*sum*⟩≡
```
fun sum(a,b) =
   let
      fun term(x) = x : int;
      fun next(x) = x + 1
   in
      summation(a,b,term,next)
   end;
> val sum = fn : int * int -> int
```

⟨*sum_squares*⟩≡
```
fun sum_squares(a,b) =
   let
      fun term(x) = x*x : int;
      fun next(x) = x + 1
   in
      summation(a,b,term,next)
   end;
> val sum_squares = fn : int * int -> int
```

The functions *sum* and *sum_squares* are of the type $\mathbb{N}\times\mathbb{N} \rightarrow \mathbb{N}$. In contrast, the function *pi_sum* is of the type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$. In order to satisfy the strict type checking in ML , we have to re-define the function *summation* so that it returns a value of the type $\mathbb{R}$.

⟨*real_summation*⟩≡
```
fun real_summation(a : int,b,f,succ) =
   if (a > b) then
      0.0
   else
      f(a) + summation(succ(a),b,f,succ);
> val real_summation = fn : int * int * (int -> real) * (int -> int) -> real
```

Note that the only change in code that is required is the replacement of the identity element of summation. We can now define *pi_sum* as

⟨*pi_sum*⟩≡
```
fun pi_sum(n) =
   let
      fun term(x) = 1.0/real(x*(x+2));
      fun next(x) = x + 4
   in
      summation(1,n,term,next)*8.0
   end;
> val pi_sum = fn : int * int -> real
```

Thus we see that the same form of the abstract function *summation* can be used to compute sums of various different kinds. It may appear that the same effect may be achieved by defining the summation function directly (not as a higher-order function) as

⟨*summation (incorrect)*⟩≡
```
fun summation(a,b) =
   if (a > b) then
      0
   else
      f(a)+summation(succ(a),b);
> Error: unbound variable or constructor: succ, f
```

and by defining a particular summing function, say `sum` as

⟨*sum (incorrect)*⟩≡
```
fun sum(a:int,b:int) =
   let
      fun f(x : int) = x;
      fun succ(x) = x + 1
   in
      summation(a,b)
   end;
```

However, this will not work because the functions `f` and `succ` are local to the function `sum` and are not defined in the global scope of the function `summation`. Both `f` and `succ` are free parameters in the scope of the function `summation` and in order to evaluate the function `summation` these two functions must be defined in its global scope with the same names. It is not advisable to define the function `summation` with `f` and `succ` as free parameters because then, in order to use the function `summation`, one has to look *inside* its definition to find these names and it cannot be used as a black-box. In principle this would be similar to having to open the back panel of a TV with a screw-driver in order to determine whether it has 110V or 220V power input. Clearly, then the TV ceases to be a black-box (at least figuratively speaking).

### 5.2.2 Polymorphic functions

There is still something unsatisfactory about our higher order *summation*. We have had to define two versions of the same function to account for the type checking in ML . Since the two versions are essentially the same, it should be possible to write a single type independent version. One option would be to pass the identity element of summation as a formal parameter to the higher order function. But, in such a case, we will need to pass the operator ('+' in this case) also as a formal parameter because the identity element depends on the operator. If we change both the identity element and the operator, it will be inappropriate to call the resulting higher order function *summation*. Then, with the same function, we will be able to compute $\prod_{i=a}^{b} f(i)$ as well by setting the operator to '*' and the identity element to 1. In view of this we will call the modified function *accumulator*. The type of the higher order function will be $accumulator : \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \to \alpha) \times (\mathbb{N} \to \mathbb{N}) \times (\alpha \times \beta \to \beta) \times \beta \to \beta$. Here $\alpha$ and $\beta$ represent sets of generic type which can be substituted with any specific instance. Such functions of generic types are called *polymorphic*. Such a function can be written in ML as follows:

⟨*accumulator*⟩≡
```
fun accumulator(a:int,b,f,succ,oper,iden) =
   if (a > b) then
      iden
   else
      oper(f(a),accumulator(succ(a),b,f,succ,oper,iden));
>val accumulator = fn
  : int * int * (int -> 'a) * (int -> int) * ('a * 'b -> 'b) * 'b -> 'b
```

We can now use the polymorphic higher order function to compute *sum* defined above in the following way:

⟨*sum (modified)*⟩≡
```
fun sum(a,b) =
   let
      fun term(x) = x : int;
      fun next(x) = x + 1
   in
      accumulator(a,b,term,next,op+,0)
   end;
> val sum = fn : int * int -> int
```

Here `op+` is the `ML` syntax for converting the infix operator `+` to the corresponding binary function. The function *pi_sum* can be defined in terms of *accumulator* as

⟨*pi_sum (modified)*⟩≡
```
fun pi_sum(n) =
   let
      fun term(x) = 1.0/real(x*(x+2));
      fun next(x) = x + 4
   in
      accumulator(1,n,term,next,op+,0.0)
   end;
> val pi_sum = fn : int -> real
```

The higher order *accumulator* can even be used to compute $factorial(n)$ as follows:

⟨*factorial (modified)*⟩≡
```
fun factorial(n) =
   let
      fun term(x : int) = x;
      fun next(x) = x + 1
   in
      accumulator(1,n,term,next,op*,1)
   end;
> val factorial = fn : int -> int
```

In fact, we can go one step further and re-define *accumulator* so that the input parameters $a$ and $b$ are also polymorphic. In such a case we will also have to pass a comparison function in order to make the $>$ operator of a generic type. We give the definition as follows:

⟨*accumulator (modified)*⟩≡
```
fun accumulator(a:'a,b,comp,f,succ,oper,iden) =
   if comp(a,b) then
      iden
   else
      oper(f(a),accumulator(succ(a),b,comp,f,succ,oper,iden));
> val accumulator = fn
  : 'a * 'b * ('a * 'b -> bool) * ('a -> 'c) * ('a -> 'a) * ('c * 'd -> 'd)
    * 'd
    -> 'd
```

We can then use the higher order *accumulator* to compute summations on the real line as well.

**Example 5.4** *Computing definite integrals of smooth functions.* A definite integral of a function $f$ between the limits $a$ and $b$ can be approximated numerically using the formula

$$\int_a^b f(x)dx \approx \left[f\left(a+\frac{\Delta x}{2}\right)+f\left(a+\Delta x+\frac{\Delta x}{2}\right)+f\left(a+2\Delta x+\frac{\Delta x}{2}\right)+\ldots\right]\Delta x$$

We can develop a higher-order `ML` function for computing definite integral as suggested above using the modified *accumulator* defined above. The function integral takes a function $f$ and the parameters $a$,$b$ and $dx$ as input and returns the value of the definite integral as the output. Thus, it has the following type:

$$integral : (\mathbb{R}\to\mathbb{R})\times\mathbb{R}\times\mathbb{R}\times\mathbb{R}\to\mathbb{R}$$

We can define the function integral as follows:

$$integral(f,a,b,dx) = accumulator((a+\frac{dx}{2}),b,op>,f,add\_dx,op+,0.0)*dx$$

where the function $add\_dx(x)$ can be defined as

$$add\_dx(x) = x+dx$$

The corresponding `ML` function can then be given as

⟨*integral*⟩≡

```
fun integral(f,a,b,dx) =
   let
      fun add_dx(x) = x + dx
   in
      accumulator((a+dx/2.0),b,op>,f,add_dx,op+,0.0)*dx
   end;
> val integral = fn : (real -> real) * real * real * real -> real
```

**Exercise 5.14** Use the function *integral* defined above to compute $\int_0^{\pi/2}\sin(x)dx$

The advantage of writing such generic higher order functions is that the correctness and efficiency analysis of the algorithmic process needs to be carried out only once. The higher order function can then be applied to solve a whole class of similar problems instead of a single one.

### 5.2.3 Constructing functions using `lambda` ($\lambda$)

$\lambda$, or `fn` in `ML` , is a special notation used in functional programming to denote a nameless function. The value of a $\lambda$-expression is a function. For example, the expression

$$\lambda x[x + 4]$$

is used to denote "a function which accepts $x$ as input and returns $x + 4$ as the output". The corresponding definition in `ML` is given as:

⟨*fn example*⟩≡
```
fn x => x + 4;
> val it = fn : int -> int
```

The general form of a $\lambda$-expression in mathematical notation is

$$\lambda\langle\text{formal parameters}\rangle[\langle\text{body}\rangle]$$

and in `ML` it is

⟨*fn*⟩≡
```
fn (⟨formal parameters⟩) => (⟨body⟩);
```

The scope of the formal parameters is limited to the body of the $\lambda$-expression. Thus `fn` is used to create functions in the same way as `fun` except that a function defined by `fn` has no name. For example we can use $\lambda$ to define the function *pi_sum* described in the previous section without using any auxiliary function as follows:

$$pi\_sum(n) = accumulator(1, n, op >, \lambda x[1/(x * (x + 2))], \lambda x[x + 4], op+, 0)$$

The equivalent definition in `ML` would be

⟨*pi_sum*⟩≡
```
fun pi_sum(n) =
   accumulator(1,n,op>,fn x:int => 1.0/real(x*(x+2)),fn x => x+4,op+,0.0);
> val pi_sum = fn : int -> real
```

The `ML` function `let` defined in Section 3.2.1 is a special form of `fn`. A `let` expression

⟨`let`⟩≡
```
let
   val ⟨identifier 1⟩ = ⟨exp 1⟩);
   val ⟨identifier 2⟩ = ⟨exp 2⟩);
         .
         .
         .
   val ⟨identifier n⟩ = ⟨exp n⟩)
in
   ⟨body⟩
end;
```

is equivalent to the $\lambda$-expression

⟨*equivalent*⟩≡
```
(fn (⟨var 1⟩,⟨var 2⟩,...,⟨var n⟩) => ⟨body⟩)
   (⟨exp 1⟩,⟨exp 2⟩,..., ⟨exp n⟩);
```

In what follows we will see many other applications of $\lambda$-expressions.

### 5.2.4 Functions as returned values

We can use $\lambda$-expressions to describe higher-order functions which return functions as values. Consider the following example.

**Example 5.5** *Curried multiplication.*
Consider the following definition of the function *mult*:

$$mult = \lambda x[\lambda y[x * y]]$$

here the type of the function *mult* is as follows:

$$mult : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

The function *mult* accepts a number $x \in \mathbb{N}$ as its input and returns a function of the type $(\mathbb{N} \to \mathbb{N})$ as its output. The corresponding definition in `ML` can be given as

⟨*mult*⟩≡
```
val mult = fn x:int => fn y => x*y;
> val mult = fn : int -> int -> int
```

A completely equivqlent way of writing this in `ML` is as follows:

⟨*mult (equivalent)*⟩≡
```
fun mult x y = x*y : int;
> val mult = fn : int -> int -> int
```

Thus typing, say, `(mult 2)` at the prompt of the `ML` interpretor returns a function $\lambda y(2 * y)$ of the type $\mathbb{N} \to \mathbb{N}$.

⟨*interactive*⟩≡
```
(mult 2);
>val it = fn : int -> int
```

which indicates that it is a function of `y`. One can now apply the function returned by `(mult 2)` to, say, 3 and obtain 6 as the answer.

⟨*interactive*⟩+≡
```
(mult 2) 3;
> val it = 6 : int
```

By defining `mult` as above, the binary function $*$ has been represented in terms of two unary functions. The advantage of defining `mult` as above is that it becomes a higher-order function and one can define other functions based on multiplication in terms of `mult`. For example, consider the `ML` definitions

⟨*double*⟩≡
```
val double = (mult 2);
> val double = fn : int -> int
```
and

⟨*hundred_times*⟩≡
```
val hundred_times = (mult 100);
> val hundred_times = fn : int -> int
```

The resulting functions $double : \mathbb{N} \to \mathbb{N}$ and $hundred\_times : \mathbb{N} \to \mathbb{N}$ are both derived from a higher-order function `mult`. These functions can now be applied to multiply a number by 2 or 100.

⟨*interactive*⟩+≡
```
double(5);
> val it = 10 : int
hundred_times(5);
> val it = 500 : int
```

We can define Curied multiplication in similar way in `Python` as well

⟨*Curried mult in Python*⟩≡
```
mult = lambda x: lambda y: x*y;
```
and invoke it as

⟨*interactive*⟩+≡
```
>>> mult(2)(3)
6
```

The above strategy of representing an $n$-ary function in terms of $n$ unary functions by using $\lambda$-expressions to return functions as parameters is called *Currying*[6]. We will see several more examples of Currying in the following examples.

**Example 5.6** Composing two functions.

[6] After the logician Haskell B. Curry. The technique is actually credited to Moses Schönfinkel who conceived it in 1924 but somehow "Schönfinkeling" has not caught on.

The composition $f \circ g : \gamma \rightarrow \beta$ of two functions $f : \alpha \rightarrow \beta$ and $g : \gamma \rightarrow \alpha$ can be defined as a higher-order function as follows:

$$f \circ g = compose = \lambda f[\lambda g[\lambda x[f(g(x)]]]$$

The type of the function *compose* is then given by

$$compose : (\alpha \rightarrow \beta) \rightarrow ((\gamma \rightarrow \alpha) \rightarrow (\gamma \rightarrow \beta))$$

The corresponding definition in `ML` can be given as

⟨*compose*⟩≡
```
val compose = fn f => fn g => fn x => f(g(x));
> val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

or, equivalently, as

⟨*compose (equivalent)*⟩≡
```
fun compose f g x = f(g(x));
> val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

The function `compose` can then be used to compose two functions, say $sin(x)$ and $cos(x)$ as follows:

⟨*sin of cos*⟩≡
```
val sin_of_cos = compose(Math.sin) Math.cos;
> val sin_of_cos = fn : real -> real
```

The function `sin_of_cos` can then be invoked on any input, say $\pi$, to compute $sin(cos(\pi))$ as

⟨*invocation*⟩≡
```
sin_of_cos(Math.pi);
> val it = ~0.841470984808 : real
```

or directly as

⟨*invocation*⟩+≡
```
(compose(Math.sin) Math.cos) Math.pi;
> val it = ~0.841470984808 : real
```

**Exercise 5.15** Indicate the type of the ML functions

1. `(compose Math.sin)`
2. `((compose Math.sin) Math.cos))`

**Exercise 5.16** Give a ML function for composing $f \circ g$ where $g(x, n) = x^n$ and and $f(x) = x + 1$ where the input $x$ and $n$ are integers. Indicate the type of composed function.

**Exercise 5.17** Consider the following alternate description of *compose* where it is of the type $compose : (\alpha \to \beta) \times (\gamma \to \alpha) \to (\gamma \to \beta)$ and is defined as

$$compose(f, g) = \lambda x[f(g(x))]$$

In fact there is an ML operator $\circ$ which follows this definition of compose:

⟨*invocation*⟩+≡

```
op o;
> val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
Math.sin o Math.cos;
> val it = fn : real -> real
(Math.sin o Math.cos) Math.pi;
> val it = ~0.841470984808 : real
```

1. Give the equivalent definition in ML and show how two functions, say *sin* and *cos*, can be composed.
2. Why is our original definition a more preferred option? Give examples to show that the original definition is more general.

**Example 5.7** $n^{th}$ repeated application of a function $f$.

If $f : \alpha \to \alpha$ is any function then we can form the $n^{th}$ repeated application of $f$ whose value at $x \in \alpha$ is $f(f(\ldots(f(x))\ldots))$. For example, if $f(x) = x + 1$ then the $n^{th}$ repeated application of $f$ is the function $g$ where $g(x) = x + n$. If $f$ is the operation of squaring a number, then the $n^{th}$ repeated application of $f$ is the function that raises its input to the $(2^n)^{th}$ power. We can define a higher-order function $repeat : (\alpha \to \alpha) \to (\mathbb{N} \to (\alpha \to \alpha)$ which accepts $f$ and $n$ as input and returns a function of the type $(\alpha \to \alpha)$ as its output.

$$repeat(f) = \lambda n \left[ \begin{cases} \lambda x = x & \text{if } (n = 0) \\ compose(f, repeat(f)(n-1)) & \text{otherwise} \end{cases} \right]$$

In ML the function *repeat* can be written as

⟨*repeat*⟩≡

```
fun repeat f n  =
        if n=0 then (fn x => x)
        else f o (repeat f (n-1));
> val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a
```

**Exercise 5.18** Use `repeat` in the `ML` interactive environment

1. To compute $x + n$ by repeating $n$ times a function $f(x) = x + 1$.
2. To compute $cos(cos \ldots (cos(1)) \ldots)$ where the function $cos$ (pre-defined in `ML` as Math.cos) is applied 100 times. This should give you the solution of the equation $cos(x) = x$ which is approximately 0.7391.[7]

The higher order `repeat` function can be written in `Python` as

⟨*repeat (Python)*⟩≡
```
def repeat(f, n):
    if n == 1:
        return f
    else:
        return lambda x: f(repeat(f,n-1)(x))
```

and it can be invoked as

⟨*interactive*⟩≡
```
>>> a = repeat(lambda x: x+1,10)(0)
>>> print(a)
10
```

**Example 5.8** Computing the derivative of a function with respect to $x$.

Consider the statement "the derivative of $sin(x)$ is $cos(x)$". What this really means is that the derivative of a function whose value at $x$ is $sin(x)$ is another function whose value at $x$ is $cos(x)$. Thus derivative may be regarded as a function which given a function $f$ as input, returns another function $Df$ as the output. Thus, if $f$ is a function and $dx$ is some small number, then the derivative $Df$ of $f$ is that function whose value at any number $x$ is given by (in the limit of small $dx$)

$$Df(x) = \frac{f(x + dx) - f(x)}{dx}$$

Thus derivative may be thought of as a higher-order function of the type

$$deriv : \mathbb{R} \to (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

which takes $dx$ as input and returns a function which takes $f$ as its input and returns the function $Df$ as the output. It can be defined using a $\lambda$-expression as

$$deriv(dx) = \lambda(f)\lambda(x)[(f(x + dx) - f(x))/dx]$$

In `ML` the equivalent definition would be

⟨*deriv*⟩≡
```
fun deriv dx f x = (f(x+dx) - f(x))/dx : real;
> val deriv = fn : real -> (real -> real) -> real -> real
```

[7] The solution of $f(x) = x$ is called the fixed-point of the function $f$.

We can use the function `deriv` to approximate the derivative of $sin(x)$ at $\pi$ (which is $\cos \pi = -1$)as follows:

⟨*invocation*⟩≡
```
(deriv(0.000001) Math.sin) Math.pi;
> val it = ~1.00000000014 : real
```

**Exercise 5.19** Use the higher-order function `deriv` to approximate the value of the derivative of $x^3$ at 5.

We can even use the higher-order derivative function to compute the partial derivative of a multi-variate function. For example, if $f(x, y) = x^3y$ then the partial derivative of $f$ with respect to $x$ is $3x^2y$. The partial derivative evaluated at say $x = 3, y = 4$ is 108.

In order to compute the partial derivative of $f$ with respect to $x$ we need to write the function $f$ in the curried form as

$$f = \lambda y[\lambda x[x^3 * y]]$$

The function $f$ then has the type $f : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$. It can be defined in `ML` as

⟨*curried f(x,y)*⟩≡
```
fun f y x = x*x*x*y : real;
> val f = fn : real -> real -> real
```

Then in order to approximate the partial derivative of $f(x, y) = x^3y$ at $x = 3, y = 4$ we can use

⟨*invocation*⟩+≡
```
((deriv (f 4) 0.001) 3) [return]
   108.036003999985
(deriv(0.000001) (f 4.0)) 3.0;
> val it = 108.000036022 : real
```

**Exercise 5.20** Use the function `deriv` to compute the partial derivative of $x^3y^2$ with respect to $y$ at $x = 3, y = 4$. Indicate the type of any function that you may need to define.

We can also combine the higher order functions *repeat* and *deriv* to define a function to compute the $n^{th}$ derivative of a given function.

**Example 5.9** *Computing the $n^{th}$ derivative of a function.*
The higher order function $nderiv : \mathbb{R} \times \mathbb{N} \to ((\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}))$ can be written as

$$nderiv(dx, n) = (repeat(derivdx)n)$$

It takes the parameters $dx$ and $n$ as input, and returns a function which takes $f$ as input and returns a function to compute the $n^{th}$ derivative of $f$ as the output. It can be written in `ML` as

⟨*nth derivative*⟩≡
```
fun nderiv(dx,n) =  (repeat (deriv dx) n);
> val nderiv = fn : real * int -> (real -> real) -> real -> real
```

We can now define a sequence of functions $D0$, $D1$, $D2$, ... to compute the zeroth, first, second, ... derivatives of a given function.

⟨*higher derivatives*⟩≡

```
val D0 = nderiv(0.001,0);
> val D0 = fn : (real -> real) -> real -> real
val D1 = nderiv(0.001,1);
> val D1 = fn : (real -> real) -> real -> real
val D2 = nderiv(0.001,2);
> val D2 = fn : (real -> real) -> real -> real
```

**Exercise 5.21** Use the functions $D0$, $D1$, $D2$, ... to compute the higher derivatives of $sin(x)$ and $cos(x)$ at a given value of $x$.

**Example 5.10** Computing the root of a function using Newton's method.

A root of a function $f(x)$ is a value $r$ such that $f(r) = 0$. Newton's method for computing the roots of a differentiable function can be described as follows. If $f$ is a function and $y$ is an approximation to the root of $f$, then a better approximation to the root can be obtained by

$$y - \frac{f(y)}{Df(y)}$$

This generalizes the formula we used for computing the square root of a number in Example 5.3.

**Exercise 5.22** Consider $f(y) = y^2 - x$. Clearly, the root of $f(y)$ gives the square root of $x$. Show that the formula for iterative improvement used in Example 5.3 is a special case of Newton's method.

Newton's method does not always converge to an answer, but it can be shown that in cases where it does converge every iteration of Newton's method doubles the number of digits of accuracy of the approximation to the root. In the special case of computing square roots we have already established that the Newton's method is guaranteed to converge.

Newton's method for computing a root of an arbitrary function $f$ can be written as a higher-order function. It should accept a function $f$, a parameter *guess* and an accuracy factor $\epsilon$ as input and return the root as the output. Hence it has the following type.

$$newton : (\mathbb{R} \to \mathbb{R}) \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

It can be defined as

$$newton(f, guess, \epsilon) = \begin{cases} guess & \text{if } acceptable?(guess, f, \epsilon) \\ newton(f, update(guess, f), \epsilon) & \text{otherwise} \end{cases}$$

where the functions $acceptable? : \mathbb{R} \times (\mathbb{R} \to \mathbb{R}) \times \mathbb{R} \to \mathbb{B}$ and $update : \mathbb{R} \times (\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$ are defined as

$$acceptable(guess, f, \epsilon) = (\mid f(guess) \mid \leq \epsilon)$$

and

$$update(guess, f) = guess - f(guess)/Df(guess)$$

where

$$Df = (deriv(dx)f)$$

We can set $dx$ to some small value like $10^{-6}$. The equivalent definition in `ML` can then be given as

⟨*newton*⟩≡

```
fun newton(f,guess,epsilon) =
   let
      fun acceptable(guess,f,epsilon) = (abs(f(guess)) <= epsilon);
      fun update(guess,f) = guess - f(guess)/((deriv 0.000001) f guess);
   in
```

```
        if acceptable(guess,f,epsilon) then guess
        else newton(f,update(guess,f),epsilon)
    end;
> val newton = fn : (real -> real) * real * real -> real
```

**Exercise 5.23** Use the `ML` code for Newton's method to

1. Compute the square root of a number by Newton's method. Compare the execution time with the program developed in Example 5.3.
2. Compute the fixed point of the equation $x = cos(x)$. Start with an initial value of $guess = 1$.

The idea of using functions as input parameters and returned values may take some getting used to, or it may appear to be little more than a mathematical trick. However the increased flexibility in expressing programming ideas is enormous and using higher-order functions it becomes possible to abstract out the essence of a general idea or algorithm without having to bother about the specific details. Thus, programming in a language that supports higher-order functions is really convenient.

## Problems

1. Use the following timer function to profile the run-times of both our methods of primality testing.

The following function `timer` measures the execution speed of any arbitrary `ML` expression. The arguments `f` is the function to be timed and `x` is its input. `n` is the number of times the function should be invoked. The function doesn't take in to account the time spent in the control loop of the `timer` function itself, and, consequently, the timing is somewhat inaccurate. Further, different runs of the same program may however give slightly different times because of system overheads.

⟨*timer*⟩≡

```
fun timer f x n =
        let fun ntimes(f,x,n) =
                let fun ntimes_iter(f,x,n,i) =
                        if (i=n) then
                                ()
                        else
                                (f(x);ntimes_iter(f,x,n,i+1))
                in
                        ntimes_iter(f,x,n,0)
                end
        in
                let val dummy =
                        Timer.startCPUTimer ()
                in ntimes(f,x,n); Timer.checkCPUTimer(dummy)
                end
        end;
> val timer = fn
  : ('a -> 'b) -> 'a -> int -> {gc:Time.time, sys:Time.time, usr:Time.time}
```

The timer function can be invoked from the `ML` interpreter to evaluate, say, `prime(79)` hundred times as follows:

⟨*invocation*⟩≡

```
(timer prime 79 1000);
> val it =
  {gc=TIME {sec=0,usec=0},sys=TIME {sec=0,usec=0},usr=TIME {sec=0,usec=10000}}
  : {gc:Time.time, sys:Time.time, usr:Time.time}
```

Note that the timer function outputs the garbage collection time, the system time and the user time separately.

2. Two prime numbers $p$ and $q$ are said to be *twin primes* if $q = p+2$. Develop a `Python` program, using step-wise refinement, to output the first twin primes after $n$ where $n$ is an input parameter.

3. Assuming that a function $f(x)$ is given, develop `Python` functions, using step-wise refinement, for

   (a) finding a root of the function between two limits by bisection method (see Section 0.7 of Kreyszig).

   (b) finding the definite integral of the function between two limits $a$ and $b$, by

      i. the trapezoidal rule (see Section 0.8 of Kreyszig)

      ii. Simpson's rule (see Section 0.8 of Kreyszig)

4. Obtain an algorithm for finding the $k^{th}$ root of a number $x$ and perform an analysis similar to that in Example 5.3.

5. The reciprocal $1/x$ of a number $x > 0$ can be computed as follows. Start with an initial guess $y_0$ and iteratively update the guess as

$$y_{n+1} = y_n * (2 - x * y_n)$$

   and terminate the process when $\mid 1 - x * y_n \mid < \epsilon$ for a given $\epsilon$. Use step-wise refinement to

   (a) determine a suitable value of $y_0$.

   (b) develop a complete `Python` program for the problem.

   (c) carry out an analysis of the number of steps required.

6. Define a higher-order double summation function to compute

$$\sum_{i=a}^{b} \sum_{j=c}^{d} f(i,j)$$

   in terms of the higher-order function *summation* defined in Section 5.2.1.

7. The higher-order summation function developed in Section 5.2.1 is recursive. Write an iterative version of the higher-order function.

8. The idea of smoothing a function is an important concept in signal processing. If $f$ is a function and $dx$ some small value, then the smoothed version of $f$ is the function whose value at a point $x$ is the average of $f(x - dx)$, $f(x)$ and $f(x + dx)$. Write a function called *smooth* which takes as input a function that computes $f$ and returns a function that computes the smoothed $f$.

9. It is sometimes valuable to repeatedly smooth a function (i.e., smooth the smoothed function, and so on) to obtain a *n-fold smoothed* function. Show how to generate the $n$-fold smoothed function of any given function using *smooth* and *repeat*.

10. Newton's method is an example of a still more general computational strategy known as *iterative improvement*. An iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a higher-order function called *iterative_improve* that takes two procedures as arguments: a method for telling whether the guess is good enough and a method for improving the guess. *iterative_improve* should return as its value a function that takes a guess as argument and keeps improving the guess until it is good enough. Express the following using *iterative_improve*:

    (a) The algorithm for computing the square root of Example 5.3

    (b) The algorithm developed in Problem 5 for computing the reciprocal of a number.

    (c) A fixed-point iteration for computing the solution of the equation $x = cos(x)$.

    (d) Newton's method.

11. Can you use the higher-order function *iterative_improve* to compute the factorial of a given number $n$ iteratively? *Iteration* itself can be regarded as a higher-order function. What would be required to write a general purpose higher-order function that can represent any iteration?

12. In a language like `ML` which can manipulate functions, we can get by without numbers (at least insofar as non-negative integers are concerned) by implementing 0 and the operation of adding 1 as

```
val zero = fn f => fn x => x;

val succ = fn n => fn f => fn x => f ((n f) x);
```

    This representation is known as Church numerals, after the logician Alonzo Church who invented $\lambda$-calculus.

    (a) Figure out how the above definitions work.

    (b) Define `one` and `two` directly (not in terms of `zero` and `succ`). [**Hint:** Use substitution to evaluate `(succ zero)`]

    (c) define addition directly.

# Part II

# Data-directed programming

# Chapter 6

# Abstract Data Types

So far we have been computing only with *integer*, *boolean* and *real* data types. These basic data-types, along with the *character* data type, are supported by most programming languages. However, for more complex computations we need to deal with more structured forms of data. Examples of such structured data-types are rational and complex numbers, lists and sequences, polynomials, vectors and matrices, trees, stacks, queues, files etc. In this chapter we will see how structured data-types like rational and complex numbers, lists and sequences can be constructed out of the basic data-types. Our ultimate objective is to create a hierarchy of data-types, as shown in Figure 6.1, with more complex ones defined on top of simpler ones, so that the programs which use a particular data-type become independent of how the data-type is implemented. Complex computations which need to manipulate objects of these types can then be designed at a higher conceptual level in terms of these data types.

Any *data-type* which is not supported natively by a programming language is called an *abstract data type*. It includes a representation for items of the data type and associated operations on the items. An implementation of an *abstract data type* is called a *data structure*. In what follows we will illustrate the design of some *abstract data types* in both `ML` and `Python` .

## 6.1 Building the *Rational* data-type (pairs)

Let us consider the task of designing an arithmetic system to carry out computations with rational numbers. The abstract data-type rational is defined as

$$RAT = \{(p, q) \mid p, q \in \mathbb{Z}, q \neq 0\}$$

where $\mathbb{Z}$ is the integer data-type. Rational numbers are interpreted as the ratio $p/q$. Our design objective is to construct functions like $plus : RAT \times RAT \rightarrow RAT$, $minus : RAT \times RAT \rightarrow RAT$, $mult : RAT \times RAT \rightarrow RAT$, $div : RAT \times RAT \rightarrow RAT$, $equalto : RAT \times RAT \rightarrow \mathbb{B}$ which adds, subtracts, multiplies, divides and checks the equality of rational numbers. Any user program that uses rational arithmetic may then

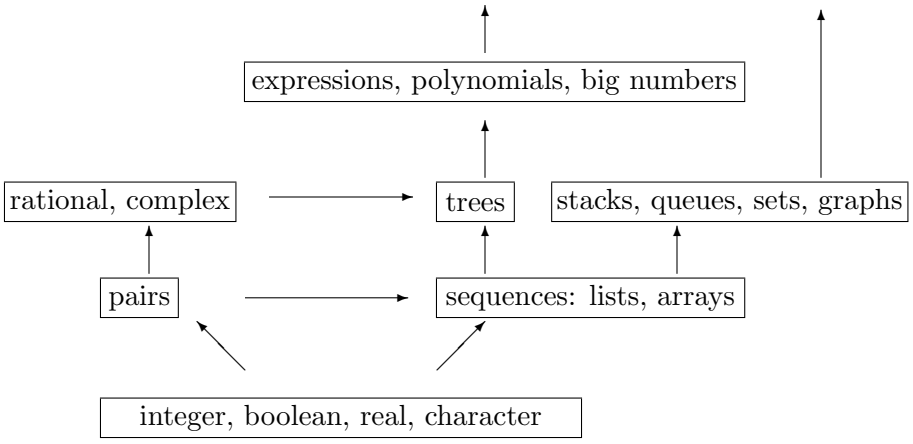


**Figure 6.1**: Hierarchy of types.

directly use these functions and treat $RAT$ as a data-type in its own right. The major advantage of such a scheme is increased modularity. Even if we decide, at a later point of time, to change our implementation of the data-type $RAT$, there should be no need to modify any higher level computation that uses rational numbers.

Let us assume, for the time being, that we can construct a rational number given a numerator and a denominator using the function

$$make\_rat : \mathbb{Z} \times \{\mathbb{Z} - \{0\}\} \rightarrow RAT$$

i.e., $make\_rat(n, d)$ returns the rational number $n/d$. Let us also assume the we have the functions

$$\begin{aligned} numer &: RAT \rightarrow \mathbb{Z} \\ denom &: RAT \rightarrow \{\mathbb{Z} - \{0\}\} \end{aligned}$$

which given a rational number as the input, return the numerator and the denominator respectively.

We can then add, subtract, multiply, divide and check equality of rational numbers using

the following relations:

$$\begin{array}{rcl}
\frac{n_1}{d_1} + \frac{n_2}{d_2} & = & \frac{n_1 * d_2 + n_2 * d_1}{d_1 * d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} & = & \frac{n_1 * d_2 - n_2 * d_1}{d_1 * d_2} \\
\frac{n_1}{d_1} * \frac{n_2}{d_2} & = & \frac{n_1 * n_2}{d_1 * d_2} \\
\frac{n_1}{d_1} / \frac{n_2}{d_2} & = & \frac{n_1 * d_2}{d_1 * n_2} \\
\frac{n_1}{d_1} = \frac{n_2}{d_2} & \Leftrightarrow & (n_1 * d_2) = (d_1 * n_2)
\end{array}$$

We can write our functions for manipulating objects of the data-type $RAT$ in terms of the above rules as

$$\begin{array}{rcl}
plus(a,b) & = & make_rat\,(numer(a) * denom(b) + numer(b) * denom(a), denom(a) * denom(b)) \\
minus(a,b) & = & make_rat(numer(a) * denom(b) - numer(b) * denom(a), denom(a) * denom(b)) \\
mult(a,b) & = & make_rat(numer(a) * numer(b), denom(a) * denom(b)) \\
div(a,b) & = & make_rat(numer(a) * denom(b), denom(a) * numer(b)) \\
equalto(a,b) & = & (numer(a) * denom(b) = denom(a) * numer(b))
\end{array}$$

Thus, as long as two integers can be *glued* together using the function *make_rat* and *un-glued* using the functions *numer* and *denom*, we have an effective way of implementing the data-type $RAT$ and its associated functions. We have to, of course, decide how to implement the functions *make_rat*, *numer* and *denom*.

## 6.2 *Rational* data-type in ML

### 6.2.1 signature, datatype and module

The programming language ML allows us to specify an *abstract data type* in terms of its *signature*. The *signature* tells us what are the essential components of the data-type available to any program that uses the data-type. It is merely a specification of the *type* and a list of operations associated with an abstract data type. It specifies *what* the operations are, but not *how* they are implemented. Consider, for example, the following ML declaration of the signature of a rational number.

⟨*Signature of rational*⟩≡
```
signature Rational =
sig
   type number;
   val makerat : int *int -> number;
   val plus : number * number -> number;
   val minus : number * number -> number;
   val mult : number * number -> number;
   val div : number * number -> number;
   val equalto : number * number -> bool;
end;
```

The *signature* tells us that any implementation of the *abstract data type* Rational must define a data-type called number; a function called makerat for constructing a rational number given two integers, and the functions plus, minus, mult, div and equalto which allow us to manipulate rational numbers. Further, only these functions will be available as the interface of the abstract data type Rational for other programs to use.

In order to implement the data-type Rational we need a facility to *glue* two integers in to a compound value. We may do this in ML using the primitive declaration datatype and a *constructor* in the following way:

⟨*datatype declaration*⟩≡
```
datatype number = ratify of int*int;
```

The above declaration defines the data-type number to be a compound value created with the constructor ratify. Note that the choice of the name of the constructor is arbitrary and we could choose any name for the constructor. The constructor ratify accepts two input arguments and returns a compound value which is a *pair* with the two input arguments as parts. Thus, if the two input integers are a and b then the compound value is ratify(a,b). The compound value can be *un-glued* by a powerful ML concept called *pattern matching*. The ML declarations

⟨*pattern matching*⟩≡
```
fun numer(ratify(a,_)) = a;
fun denom(ratify(_,b)) = b;
```

define `numer` and `denom` to be functions that accept compound items created with the constructor `ratify` as input and return the components `a` and `b` respectively as output. The mechanism for extracting the components `a` and `b` from the compound item is called *pattern matching.* The notation `ratify(a,_)` means that we do not care about the second argument in the pattern matching.

Armed with the above, we can now define an `ML` *module* called `RAT1` which is our first implementation of the abstract data type `Rational`.

⟨*Module RAT1*⟩≡

```
structure RAT1 : Rational =
struct
   datatype number = ratify of int*int;

   exception DenomIsZero;

   fun gcd(a,b) =
      if b = 0 then
         a
      else gcd(b,a mod b);

   fun makerat(n:int,d:int) : number =
   if d = 0 then
      raise DenomIsZero
   else
      let
         val pn =  abs(n);
         val pd =  abs(d);
         val sign =
            if n = 0 then
               0
            else (n div pn) * (d div pd);
         val g = gcd(pn,pd)
      in
         ratify(sign * pn div g,pd div g)
      end;

   fun numer(ratify(x,_)) = x;
   fun denom(ratify(_,y)) = y;

   fun plus(a,b) =
      let
         val x =  numer(a)*denom(b) + numer(b)*denom(a);
         val y = denom(a)*denom(b);
```

```
        in
           makerat(x,y)
        end;

    fun minus(a,b) = ...
    fun mult(a,b) = ...
    fun div(a,b) = ...
    fun equalto(a,b) = ...
end;
```

**Exercise 6.1** Complete the functions `minus`, `mult`, `div` and `equalto` in the above module definition.

The above `ML` module `RAT1` is a particular implementation (a *data structure*) of the abstract data type `Rational`. The declaration

⟨*structure declaration*⟩≡
```
structure RAT1 : Rational =
struct
   ...
end;
```

specifies that the module `RAT1` adheres to the signature `Rational` defined earlier. We can use the module `RAT1` from the `ML` prompt in the following way.

⟨*ML usage examples*⟩≡
```
- val x = RAT1.makerat(2,4);
val x = ratify (1,2) : RAT1.number
- val y = RAT1.makerat(3,~6);
val y = ratify (~1,2) : RAT1.number
- RAT1.plus(x,y);
val it = ratify (0,1) : RAT1.number
```

In the above example `DenomIsZero` is declared to be an `exception` which is a standard data type in `ML` . The `DenomIsZero` exception is *raised* when the denominator is zero in `makerat`. A typical `ML` usage example is

⟨*ML exception*⟩≡
```
- RAT1.makerat(2,0);
uncaught exception DenomIsZero
  raised at: rat.ml:23.13-23.24
```

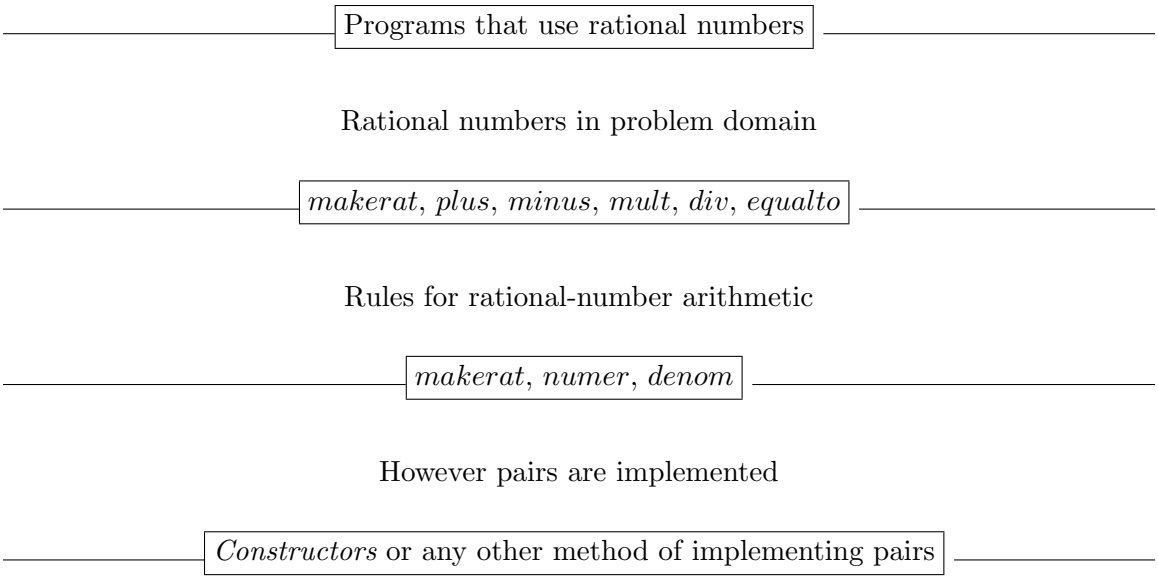


**Figure 6.2**: Abstraction barriers in the implementation of rational numbers

We will illustrate later how to write code to *catch* and exception to facilitate a graceful exit.

Note that the `ML` module `RAT1` uses the internal function `gcd` to reduce a rational number to its canonical form. It also uses functions `numer` and `denom` for the internal representation of the data type. However, since these functions are not a part of the signature `Rational`, they are not visible outside the module `RAT1`, and, consequently, an usage like `RAT1.gcd(4,6)` is illegal. Only those components of a module which are explicitly specified in the *signature* of the abstract data type are available for use from outside.

We have thus implemented the data-type for rational numbers using several *abstraction barriers*. At the lowest level is the pairing of integers using the primitive `datatype` and the constructor facility in `ML` . At the next higher level are our functions `makerat`, `numer` and `denom`. Using these functions we have defined, at a still higher level, our functions for manipulating rational numbers. Any user program at a higher level may now simply define variables of the type rational (`RAT1.number`), construct rational numbers using the function `makerat` and use the functions for carrying out arithmetic on rationals directly. The overall hierarchy is illustrated in Figure 6.2. Thus, at the interface to the highest level of any program using rational numbers, only the functions `RAT1.makerat`, `RAT1.plus`, `RAT1.minus`, `RAT1.mult`, `RAT1.div` and `RAT1.equalto` need to be available. The rest of the detail of the implementation may be completely hidden from an user program. This is achieved through the *signature* declaration.

The above idea of hiding the details of the implementation of an abstract data type is key to modular programming. After all, in order to drive a car we need to know only the

methods available - the use of the steering, the brake, the indicators [1], the clutch and gear changing. We definitely do not want to know about the details of the transmission system, the viscocity of the brake fluid etc. These issues are relevant when we are designing a car, exactly in the same way the internal representation of the data type is relevant to the designer of the data-type module. This clear cut separations of concerns is key to succesful development of large programs.

We may even change the underlying representation of rational numbers without affecting any computation at the higher levels. For example, we may consider a totally different, though somewhat inefficient, implementation of our functions `makerat`, `numer` and `denom`. If we consider only positive rational numbers, we may define the function `makerat` as

$$makerat(n,d) = 2^{n/gcd(n,d)} * 3^{d/gcd(n,d)}$$

The function *makerat* then returns an integer as defined above. Using the *unique prime factorization theorem* which states that any positive integer can be uniquely expressed as a product of primes, we can define the functions *numer* and *denom* by simply counting the numbers or 2's and 3's, respectively, in the prime factorization of any integer returned by *makerat*. We may thus define these functions as

$$numer(r) = \begin{cases} 0 & \text{if } r \ mod\ 2 \neq 0 \\ numer(r\ div\ 2) + 1 & \text{otherwise} \end{cases}$$

and

$$denom(r) = \begin{cases} 0 & \text{if } r \ mod\ 3 \neq 0 \\ denom(r\ div\ 3) + 1 & \text{otherwise} \end{cases}$$

We can then implement an entirely different module (a different data structure) for the same abstract data-type `Rational` in the following way.

⟨*Module RAT2*⟩≡

```
structure RAT2 : Rational =
struct
   type number = int;

   exception DenomIsZero;

   fun gcd(a,b) =
      if b = 0 then
         a
      else gcd(b,a mod b);

   fun power(x,n) = ...
```

[1] Not in Delhi!

```
fun makerat(n:int,d:int) : number =
   if d = 0 then
      raise DenomIsZero
   else
      let
         val pn =  abs(n);
         val pd =  abs(d);
         val sign =
            if n = 0 then
               0
            else (n div pn) * (d div pd);
         val g = gcd(pn,pd)
      in
         if sign = 0 then
            power(2,pn div g)*power(3,pd div g)
         else
            sign * power(2,pn div g)*power(3,pd div g)
      end;

fun numer(r : number) =
   let
      val sign = r div abs(r);
      fun niter(r,i) =
         if (r mod 2) <> 0 then
            i
         else
            niter(r div 2,i+1)
   in
      sign*niter(abs(r),0)
   end;

fun denom(r : number) =
   let
      fun niter(r,i) =
         if (r mod 3) <> 0 then
            i
         else
            niter(r div 3,i+1)
   in
      niter(abs(r),0)
end;
```

```
    fun plus(a,b) =
       let
          val x = numer(a)*denom(b) + numer(b)*denom(a);
          val y = denom(a)*denom(b);
       in
          makerat(x,y)
       end;

    fun minus(a,b) = ...
    fun mult(a,b) = ...
    fun div(a,b) = ...
    fun equalto(a,b) = ...
end;
```

**Exercise 6.2** Verify that if we use the above definitions of the functions `makerat`, `numer` and `denom`, we do not need to modify any of the functions `plus`, `minus`, `mult`, `div` and `equalto`. Also verify that any top level program that was defined using the module `RAT1` can be used unchanged with the module `RAT2`.

## 6.3 *Rational* data-type in Python

### 6.3.1 Interfaces, Classes and Objects: Basics of Object Oriented Programming

Unlike in ML where we only have functions which return values of specified types, in an imperative language an instance of an abstract data type has a physical presence in the sense that it occupies a definite position in memory (like imperative variables discussed before) and is described through the *state* of the memory. Such instances of abstract data types are called *objects*. In addition to their state in the memory *objects* also have associated *methods* which represent the operations associated with the abstract data type.

In an *object oriented* programming language, abstract data types are implemented using *interfaces*, *classes* and *objects* which we describe below.

An *interface* in Python is a specification of an abstract data type. It is analogous to a *signature* in ML and is merely a list of the operations associated with the abstract data type. Like *signatures* in ML , *interfaces* in Python specify what the operations associated with an abstract data type are, but not how they are implemented. An interface for the abstract data type Rational can be specified in Python as follows.

⟨*Rational interface*⟩≡

```
class RationalInterface:
        def show(self):
                pass
        def add(self,a):
                pass
        def sub(self,a):
                pass
        def mult(self,a):
                pass
        def div(self,a):
                pass
        def equal(self,a) -> bool:
                pass
        def lessthan(self,a) -> bool:
                pass
```

The interface `RationalInterface` specifies that any realization of this data-type must support the methods `show`, which prints the *object*; `add`, `sub`, `mult` and `div` which return the results of arithmetic operations with other objects; and `lessthan`, and `equal` which return the results of comparison with other objects. `pass` indicates that the actual methods will be defined later.

A *class* in `Python` is a particular realization of an abstract data type, pretty much similar to a *module* in `ML` . A *class* implements an *interface* if it implements all the methods of an *interface*. A class typically is a specification of the *data fields* or *instance variables* that an *object* of this *class* contains as well as the *methods* or *operations* that an *object* can execute. An *object* is a specific instance of a class which is created with a *constructor*. When an *object* is created, memory is allocated for its data fields which are initialized to specific beginning values.

In what follows, we give a *class* corresponding to the abstract data type `Rational`. Class methods in `Python` must have an extra first parameter called `self` in the method definition, which refers to the object itself.. We do not give a value for this parameter when we call the method, `Python` provides it itself. If we have a method that takes no arguments, then we still have to have one argument.

⟨*Rational class*⟩≡

```
class Rational(RationalInterface):
    __numer,__denom = 0,0
    def __gcd(self,n,d):  # we don't want gcd to be visible outside the class
        pn,pd = abs(n),abs(d)
        rem = pn % pd;
        while (rem != 0):
            pn,pd =  pd,rem
            rem = pn % pd
        return pd

    def __init__(self,n,d):   # the constructor
        pn = abs(n)
        pd = abs(d)
        if (pd == 0): # the denominator is 0. we can't proceed */
            raise ZeroDivisionError("denom is 0. psychonalysis suggested!")
            exit(0)
        if (pn == 0):
            sign = 0
        else:
            sign = (n//pn)*(d//pd)
        g  = self.__gcd(pn, pd)
        self.__numer = sign * (pn//g)
        self.__denom = pd//g;
```

```
def show(self):
    print(self.__numer,"/",self.__denom)

def add(self,a):
    n = self.__numer * a.__denom + self.__denom * a.__numer
    d = self.__denom * a.__denom
    return Rational(n,d)

def sub(self,a):
    n = self.__numer * a.__denom - self.__denom * a.__numer;
    d = self.__denom * a.__denom;
    return Rational(n,d)

def mult(self,a):
    n = self.__numer * a.__numer
    d = self.__denom * a.__denom
    return Rational(n,d)

def div(self,a):
    if (a.__numer == 0):
        raise ZeroDivisionError("psychonalysis suggested!")
    else:
        n = self.__numer * a.__denom
        d = self.__denom * a.__numer
        return Rational(n,d)

def equal(self,a):
    return (self.__numer == a.__numer) & (self.__denom == a.__denom)

def lessthan (self,a):
    return (self.__numer *  a.__denom) < (a.__numer * self.__denom)
```

The above *class* description associates the instance variables `__numer` and `__denom` with every *object*. The prefix double underscore (`__`) indicates that these instance variables are private to the class and cannot be accessed from outside the class. A new *object* of the type `Rational` can be created with the *constructor* called `Rational`. The name of the constructor for a *class* and the name of the *class* must be the same. The instruction `Rational(n, d)`, used at several places in the above code, creates a new `Rational` *object*. New memory is allocated for the *instance variables* `__numer` and `__denom` and these are initialized to the values `n` and `d` respectively, as specified by the constructor. Note that it is not necessary to define and initialize the instance variables to 0 outside the constructor, as we have done. Instead, they may be defined and initialized directly within the constructor itself.

Apart from the *instance variables* the *object* also has the associated *methods* - `add`, `sub` etc.

Note that the use of double underscore (`__`) in the above code specifies that the corresponding *instance variables* or *methods* are private to the class and these items are hidden from other classes. The *instance variables* `__numer` and `__denom` and the *method* `__gcd` are details of the internal representation of a rational number in this particular *class* definition and there is no need to make these visible outside the *class* definition. This principle of making only the necessary items visible outside a *class* definition is called *data hiding*.

In what follows we give an example of the usage of the `Rational` data type from a top level program.

⟨*Testing the Rational class*⟩≡

```
print(issubclass(Rational, RationalInterface))
n,d = int(input("n1: ")),int(input("d1: "))
a = Rational(n,d)
a.show()
n,d = int(input("n2: ")),int(input("d2: "))
b = Rational(n,d)
b.show()
c = a.div(b)
c.show()
print(a.equal(c))
```

In the above definition of the *class* `Rational`, the methods `show`, `add`, `sub` etc. are all *object methods*. Note that they are invoked from above test program as `a.show()`, `a.add(b)` etc., where `a` is the name of an object. `issubclass` is a special `Python` function that checks whether the *class* `Rational` indeed implements the *interface* `RationalInterface`.

**Exercise 6.3** Implement a `Python` class for `Rational` using the same interface given above using the second scheme described in the previous section using uniqueness of prime factorization. Show that the test program can be used unchanged.

## Problems

Implement the following in both `ML` and `Python` .

1. Develop an abstract data-type for carrying out arithmetic with complex numbers.
2. Develop an abstract data-type called *interval*. An interval is defined by two real numbers signifying the *lower* and *upper* bounds. Develop the following functions for carrying out operations on intervals:
   (a) *make_interval*, for creating an interval
   (b) *intadd*, for adding two intervals. Here the minimum value of the sum should be the sum of the two lower bounds and the maximum value of the sum should be the sum of the two upper bounds.
   (c) *intsubtract*, for subtracting two intervals.
   (d) *intmult*, for multiplying two intervals.
   (e) *intdiv*, for dividing two intervals.
3. Suppose that you have to compute resistances in electrical circuits where the resistance of each resistor is known only up to some tolerance. For example, a resistor labeled "6.8 ohms with 10% tolerance" has a resistance between 6.12 ohms and 7.48 ohms. Using the abstract data-type called *interval* developed in the previous exercise, develop functions for computing the resistances of
   (a) two resistors connected in series.
   (b) two resistors connected in parallel.

# Chapter 7

# Programming with Lists

List is among the simplest of all data structures yet one of the most important, because a very large variety of application programs are based on them. In this chapter we will study the basics of programming with lists in both functional and imperative settings. We will use `Java` to illustrate imperative style list programming, because `Python` supports lists natively and is not very different from `ML` .

## 7.1 Lists

Let $\alpha$ be an arbirary data-type. The abstract data-type $\alpha$–$LIST$, which a list of elements of $\alpha$, is defined recursively as follows. For any arbitrary data-type $\alpha$

1. The empty list [] is an element of $\alpha$–$LIST$.
2. $\alpha$–$LIST = \alpha \times \alpha$–$LIST$

Thus an instance of $\alpha$–$LIST$ is a finite *sequence* of a basic data-type

$$\alpha\text{–}LIST = \alpha^* = \bigcup_{n=0}^{\infty} \alpha^n$$

In other words a member of $\alpha$–$LIST$ may be empty or may contain an arbitrarily long sequence of the elements of the set $\alpha$. We will denote the data-type of non-empty lists as

$$\alpha\text{–}LIST^+ = \alpha^+ = \bigcup_{n=1}^{\infty} \alpha^n$$

Following the methodology of implementing a new data-type illustrated in the last chapter, we will develop the $\alpha$–$LIST$ data-type using abstraction barriers. In addition to deciding on a representation for lists in the `ML` and `Java` programming languages, we will also provide certain functions available at the interface of the next higher level for manipulating lists. In particular we will provide the following list functions:

1. $attach : \alpha \times \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST$, which given an element from the set $\alpha$ and a list (which may be empty) attaches the element to the front of the list. For example, if $ls = [1, 2, 3]$, then $attach(0, ls)$ should return the list $ls = [0, 1, 2, 3]$, and $attach(0, [])$ should return the list $[0]$.

2. $empty : \alpha\text{–}LIST \rightarrow \{true, false\}$, which given an input list determines whether it is empty or not.

3. $head : \alpha\text{–}LIST^{+} \rightarrow \alpha$, which given a non-empty list as its input returns the first element of the type $\alpha$.

4. $tail : \alpha\text{–}LIST^{+} \rightarrow \alpha\text{–}LIST$ which given a non-empty list as its input returns the sub-list without the first element. It returns the empty list if the input list has only one element.

Assuming, for the time being, that we can implement the abstract data-type $\alpha\text{–}LIST$ with the above associated methods in both `ML` and `Java` programming languages, we can illustrate its use by developing some algorithms for manipulating lists.

**Example 7.1** *Determining whether a given list is a singleton list (contains only one element).*
We can define the function $singleton : \alpha\text{–}LIST \rightarrow \{true, false\}$ as

$$singleton(ls) = \begin{cases} false & \text{if } empty(ls) \\ empty(tail(ls)) & \text{otherwise} \end{cases}$$

**Example 7.2** *Finding the maximum element in a non-empty list of integers.*
Here $\alpha = \mathbb{N}$ and we are seeking a function of the type

$$MAXM : \alpha\text{–}LIST^{+} \rightarrow \alpha$$

An algorithm for the function $MAXM$ can be specified as

$$MAXM(ls) = \begin{cases} head(ls) & \text{if } singleton?(ls) \\ max(head(ls), MAXM(tail(ls))) & \text{otherwise} \end{cases}$$

where the binary function $max : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ may be defined as

$$max(a, b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{otherwise} \end{cases}$$

**Correctness:** We can establish the correctness of the above functional description by demonstrating that

1. the number returned by $MAXM$ is and an element of the input list, and

2. it is the largest element of the list (note that there may be more than one occurrence of the largest value).

*Proof:* By induction on $n$, the length of the list.

**Base case.** $(n = 1)$ or $singleton?(ls)$. If the list has only one element then $MAXM(ls) = head(ls)$ which is an element of the list and is trivially the largest.

**Induction hypothesis.** $MAXM(ls)$ returns the largest value in the list if the size of the list is $(n - 1)$.

**Induction step.** Consider a list $ls$ such that the size is $(n > 1)$. Note that $tail(ls)$ is a list of size $(n - 1)$. Now,

$$MAXM(ls) = max(head(ls), MAXM(tail(ls))) = max(a, b)$$

where $a = head(ls)$ is an element of the list and $b = MAXM(tail(ls))$ is the largest element in the sub-list $tail(ls)$ by the induction hypothesis. By the definition of the binary function $max$, whose correctness is trivially established, $MAXM$ thus returns the largest element in the list $ls$.

□

The time complexity of the above algorithm is obviously $O(n)$.

**Exercise 7.1** Write an iterative version of the function $MAXM$.

**Example 7.3** *Computing the length of a list.*
We can define the function $length : \alpha\text{–}LIST \to \mathbb{N}$ recursively as

$$length(ls) = \begin{cases} 0 & \text{if } empty(ls) \\ 1 + length(tail(ls)) & \text{otherwise} \end{cases}$$

Altenatively, we can construct an iterative version of the above function using the invariant condition

$$INV = \text{after } i \text{ iterations: } length(ls) = n - i \wedge len = i \wedge 0 \leq i \leq n$$

where $ls0$ is the length of the original list as follows:

$$length(ls) = length\_iter(ls, 0)$$

where

$$length\_iter(ls, len) = \begin{cases} len & \text{if } empty(ls) \\ length\_iter(tail(ls), len + 1) & \text{otherwise} \end{cases}$$

**Exercise 7.2** Establish the correctness of the above algorithms for computing the length of a given list and estimate the time complexity.

**Example 7.4** *Appending two lists.*
The functiona $append : \alpha\text{–}LIST \times \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST$ can be written as follows.

$$append(l1, l2) = \begin{cases} l2 & \text{if } empty(l1) \\ attach(head(l1), append(tail(l1), l2)) & \text{otherwise} \end{cases}$$

The correctness of the function *append* can be established by **PMI**.
**Correctness:** To show that if $l1 = [l1_1\ l1_2 \ldots l1_n]$ and $l2 = [l2_1\ l2_2 \ldots l2_m]$, then $append(l1, l2)$ returns the list $[l1_1\ l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m]$.

*Proof:* By induction on $n$ (the length of $l1$).

**Basis.** $n = 0$ or $l1 = []$). $append(l1, l2) = l2 = [l2_1\ l2_2 \ldots l2_m]$ by function definition.

**Induction hypothesis.** For all $0 \leq k \leq n$ such that $k = n - i + 1$ is the length of $l1 = [l1_i\ l1_2 \ldots l1_n]$, $append(l1, l2)$ returns the list $[l1_i\ l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m]$.

**Induction step.** Consider $l1 = [l1_1\ l1_2 \ldots l1_n]$. We have that

$$\begin{array}{rcll} append(l1, l2) & = & attach(head(l1), append(tail(l1), l2)) & \text{by function definition} \\ & = & attach(l1_1, append(tail(l1), l2)) & \text{by definition of } head \\ & = & attach(l1_1, [l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m]) & \text{by induction hypothesis} \\ & = & [l1_1, l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m] & \text{by definition of attach} \end{array}$$

□

**Exercise 7.3** Show that the time complexity of *append* is $O(n)$ where $n$ is the size of $l1$. What is the space complexity?

**Example 7.5** *Higher order list functions:* map *and* filter.
The higher order function *map* is of the type $map : (\alpha \rightarrow \alpha) \times \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST$. Given a function and a list as its input *map* returns the list formed by applying the input function on every element of the input list. For example, if the input list is $ls = [1, 2, 3, 4, 5]$, then $map(square, ls)$ should return the list $[1, 4, 9, 16, 25]$ and $map(cube, ls)$ should return the list $[1, 8, 27, 64, 125]$.

The higher order function $filter$ is of the type $filter : (\alpha \rightarrow \mathbb{B}) \times \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST$. It accepts a predicate (boolean function) of the input type $\alpha$ and a list as its input and returns a sub-list of those elements for which the predicate is *true*. For example, if the input list is $ls = [1, 2, 3, 4, 5]$, then $filter(odd, ls)$ should return the list $[1, 2, 5]$ and $filter(prime, ls)$ should return the list $[2, 3, 5]$.

We can write functional algorithms for *map* and $filter$ as follows:

$$map(f, ls) = \begin{cases} [] & \text{if } empty(ls) \\ attach(f(head(ls)), map(f, tail(ls))) & \text{otherwise} \end{cases}$$

$$filter(pred?, ls) = \begin{cases} [] & \text{if } empty(ls) \\ attach(head(ls), filter(pred?, tail(ls))) & \text{if } pred?(head(ls)) \\ filter(pred?, tail(ls)) & \text{otherwise} \end{cases}$$

**Exercise 7.4** Establish the correctness of *map* and *filter* using **PMI** and estimate the time complexities.

Hence we see that the abstract data-type $\alpha$–*LIST* along with its associated functions *attach*, *empty*, *head* and *tail* is quite powerful and we can indeed develop complex functions for list manipulation using these. We have to, of course, address the issues of implementing the abstract data-type in both ML and Java programming languages. In what follows, we will first develop the list data-type in ML and then follow it up with a Java implementation.

## 7.2 The $\alpha$–*LIST* data-type in ML

$\alpha$–*LIST* happens to be a built-in standard data-type in ML which provides the following *constructors*:

1. The constructor `[]` creates an empty instance of $\alpha$–*LIST*.
2. If `x` $\in \alpha$ and `ls` $\in \alpha$–*LIST*, then `x::ls` is an instance of $\alpha$–*LIST* with `x` attached to the front of `ls`. Hence, the constructor `::` is the function *attach* is ML .

We may define the basic methods *attach*, *empty*, *head* and *tail* of the data-type $\alpha$–*LIST* in ML as follows

⟨ *List data-type in* ML ⟩≡

```
structure LIST =
struct
   exception emptylist;

   fun empty([]) = true
     | empty(x::ls) = false;

   fun attach(x,ls) = x::ls;

   fun head([]) = raise emptylist
     | head(x::ls) = x;

   fun tail([]) = raise emptylist
     | tail(x::ls) = ls;
end;
```

Thus, the abstract data-type $\alpha$–*LIST* can easily be realized in `ML` using the basic constructors `[]` and `::` and using *pattern matching*.

However, since `ML` already provides the basic *constructors* for list programming, we will continue to use the `ML` primitives directly in our subsequent programs instead of using `struct LIST`. We will do so only for convenience with the understanding that both are equivalent.

We can now translate the functional descriptions of Examples 7.1, 7.2, 7.3, 7.4 into `ML` programs.

⟨*singleton?*⟩≡
```
fun singleton([]) = false
  | singleton(x::[]) = true
  | singleton(ls) = false;
```

⟨*MAXM*⟩≡
```
fun max([]) = raise empty
  | max(x::[]) = x
  | max(x::ls) =
        if x > max(ls) then
                x
        else
                max(ls);
```

⟨*length*⟩≡
```
fun length([]) = 0
  | length(x::ls) = length(ls) + 1;
```

⟨*length (iterative)*⟩≡
```
fun length(ls) =
  let
     fun length_iter([],len) = len
       | length_iter(x::ls,len) = length_iter(ls,len+1)
  in
     length_iter(ls,0)
  end;
```

⟨*append*⟩≡
```
fun append([],l2) = l2
  | append(x::ls,l2) = x::append(ls,l2);
```

⟨*map*⟩≡
```
fun map f [] = []
  | map f (x::ls) = f(x)::(map f ls);
```

⟨*filter*⟩≡
```
fun filter pred [] = []
  | filter pred (x::xs) =
      if pred(x) then
         x::(filter pred xs)
      else
         (filter pred xs);
```

**Exercise 7.5** Use the ML interpreter to execute each of the above functions.

Now that we have implemented the data-type $\alpha$–$LIST$ in ML , we can consider a few more algorithms for lists.

**Example 7.6** *Finding the $n^{th}$ element of a list.*
The function we are seeking is of the type $nth : \mathbb{N} \times \alpha\text{–}LIST^{n+} \to \alpha$, where $\alpha\text{–}LIST^{n+}$ is the set of lists of size at least $n$. If $n = 0$, then the function should return the first element of the list. Otherwise, it should return the $n^{th}$ element of the list taking the first element as the $0^{th}$. We may define the function in ML as

⟨*nth*⟩≡
```
fun nth(n,[]) = raise emptylist
  | nth(n,x::ls) =
      if n = 0 then x else nth(n-1,ls);
```

**Exercise 7.6** Establish the correctness of the above function using **PMI**.

**Example 7.7** *Reversing a given list.*
We can define the function $reverse : \alpha\text{–}LIST \to \alpha\text{–}LIST$ inductively as follows. Inducting on the length of the list, the base case is clearly given as `reverse([]) = []`. Given that we can solve the problem `reverse(ls)` (induction hypothesis), the problem `reverse(x::ls)` can clearly be solved as `append(reverse(ls),x::[])`. Hence we have

⟨*reverse*⟩≡
```
fun reverse([]) = []
  | reverse(x::ls) = append(reverse(ls),x::[]);
```

**Exercise 7.7** Establish the correctness of the above function using **PMI**.

Note, however, that the above function `reverse` has an unacceptably high time complexity. To see this we may write a recurrence relation describing the time complexity of the above function. Let $T(n)$ be the number of operations required to reverse a list of size $n$ using the above algorithm. Note that in order to reverse a list of size $n$, it is required to reverse a list of size $n-1$ and append it to a single element list. Also note that `(append x y)` requires $O(m)$ steps when the list `x` is of size $m$ and a `::` operation is required to form a single element list. A recurrence for $T(n)$ may be given as

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + n \end{aligned}$$

We can solve the above recurrence by telescoping, i.e.,

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &\vdots \\ &= T(0) + 1 + 2 + \ldots + n \\ &= n(n+1)/2 \end{aligned}$$

Hence the overall time complexity of the function `reverse` given above is $O(n^2)$.

**Exercise 7.8** Estimate the space complexity of the above algorithm.

Alternatively, we may define an iterative version of the above function using the invariant

$INV =$ after $i$ iterations: $length(ls) = n - i \wedge length(rev) = i \wedge append(reverse(rev), ls) = ls0$

⟨*reverse (iterative)*⟩≡
```
fun rev(ls) =
  let
    ⟨Code for rev_iter⟩
  in
    reviter(ls,[])
  end;
```

⟨*Code for* `rev_iter`⟩≡
```
fun reviter([],rev) = rev
  | reviter(x::ls,rev) = reviter(ls,x::rev)
```

**Exercise 7.9** Establish the correctness of the above iterative algorithm for reversing a given list and show that the time complexity is $O(n)$. Compare the space complexities of the two algorithms.

**Exercise 7.10** Execute the ML programs for both the algorithms for reversing random lists of size 2,4,8,32,64,128,256,512 and 1024 and compare the execution times of the two algorithms. Verify that each time the problem size is doubled, the execution time of the first algorithm increases by roughly four times whereas the execution time of the second algorithm doubles.

**Example 7.8** *Inserting an element into a sorted list.*
We will develop the function of the type $insert : \alpha \times \alpha\text{–}LIST^{sorted} \rightarrow \alpha\text{–}LIST^{sorted}$, where $\alpha\text{–}LIST^{sorted}$ is the data-type denoting all lists sorted in the ascending order. We can develop the function *insert* by inducting on the length of the list. The base case is clearly given as `insert(a,[]) = a::[]`. Given the inductive hypothesis that we can solve the problem `insert(a,ls)`, we can program the induction step as follows

⟨*insert*⟩≡
```
fun insert(a,[]) = [a]
  | insert(a,x::ls) =
     if a < x then a::x::ls else x::insert(a,ls);
```

**Exercise 7.11** Establish the correctness of the above algorithm using **PMI** and show that the worst case time complexity (measured in terms of number of `::` operations) of the above algorithm for inserting an element into a list of size $n$ is $n+1$. Under what condition of the input list does the worst case situation occur?

**Example 7.9** *Merging two sorted lists.*
We can again develop an algorithm for the function $merge : \alpha\text{–}LIST^{sorted} \times \alpha\text{–}LIST^{sorted} \rightarrow \alpha\text{–}LIST^{sorted}$ inductively. Inducting on the length of the lists, we have the base cases `merge([],l2) = l2` and `merge(l1,[]) = l1`. Given that we can solve `merge(l1,y::l2)` when `l1` is of size $n \geq 0$ and `merge(x::l1,l2)` when `l2` is of size $m \geq 0$, we can write the program as

⟨*Code for merge*⟩≡
```
fun merge([],l2) = l2
  | merge(l1,[]) = l1
  | merge(x::l1,y::l2) =
       if x <= y then x::merge(l1,y::l2) else y::merge(x::l1,l2);
```

**Exercise 7.12** Establish the correctness of the above algorithm.

**Exercise 7.13** Develop an iterative version of the above algorithm.

**Example 7.10** *Insertion sort.*
We will define a function $insort : \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST^{sorted}$, in terms of the function *insert* defined in Example 7.8 as follows. Inducting on the length of the list, the base case is clearly `insort([]) = []`. Given that we can solve `insort(ls)`, the problem `insort(x::ls)` is merely the problem of inserting `x` in to the sorted list `insort(ls)`. Hence, we have

⟨*insort*⟩≡
```
fun insort([]) = []
  | insort(x::ls) = insert(x,insort(ls));
```

**Exercise 7.14** Establish the correctness of the *insort* function using **PMI**.

We can analyze the time complexity of the above algorithm as follows. Let $T(n)$ be the number of operations required to sort a list of size $n$ using the above algorithm. Note that in order to solve a sorting problem of size $n$, it is required to solve a sorting problem of size $n-1$ in addition to inserting an element, `x` , into a list of size $n-1$. A recurrence for $T(n)$ may be given as

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + n \end{aligned}$$

which yields $T(n) = n(n+1)/2 = O(n^2)$.

**Exercise 7.15** Show that though the worst case time complexity of the insertion sort algorithm is $O(n^2)$, the best case complexity, when the input list is sorted, is $O(n)$. Hence the algorithm is very efficient when the input list is nearly sorted. What is the average case behaviour of the algorithm when roughly half the elements are in the reverse order?

Alternatively, we can write an iterative version of the function for insertion sort using the invariant

$$\begin{aligned} INV = \text{after } i \text{ iterations: } & length(ls) = n - i \wedge length(result) = i \\ & \wedge result \text{ contains the first } i \text{ elements of } ls0 \text{ in sorted order} \\ & \wedge ls \text{ contains the last } n - i \text{ elements of } ls0 \end{aligned}$$

⟨*insort (iterative)*⟩≡
```
fun insort(ls) =
   let
     ⟨Code for insort_iter⟩
   in
     insort_iter(ls,[])
)
```

⟨*Code for* `insort_iter`⟩≡
```
fun insort_iter([],result) = result
  | insort_iter(x::ls,result) = insort_iter(ls,insert(x,result));
```

**Exercise 7.16** Establish the correctness of the iterative algorithm and compare the space complexities of the recursive and the iterative algorithms for insertion sort.

**Example 7.11** *Merge sort.*
We have discussed a sorting algorithm based in insertion. We can develop a more efficient algorithm for sorting using *divide-and-conquer*. Given that we can merge two sorted lists of sizes $n$ and $m$ in $O(n+m)$ steps and split a list of size $n$ in two roughly equal sub-lists of size $n \ div \ 2$ in $O(n)$ steps, we can apply *divide-and-conquer* in the following way. Using **PMI** version 3 on the length of the list we can give the base case as `msort(x::[]) = x::[]`. If we can split the original list into two half size lists `l1` and `l2`, then, by induction hypothesis, we can sort `l1` and `l2` with `msort(l1)` and `msort(l2)`. Clearly, the induction step is `merge(msort(l1),msort(l2))`. Hence we have the following program.

⟨*msort*⟩≡
```
fun msort([]) = []
  | msort(x::[]) = x::[]
  | msort(ls) =
      let
         ⟨Code for split⟩
         ⟨Code for merge⟩
         val (l1,l2) = split(ls)
      in
         merge(msort(l1),msort(l2))
      end;
```

We can develop the function $split : \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST \times \alpha\text{–}LIST$ using the following invariant. Let the original list be $ls0 = [a_1, a_2, \ldots, a_n]$.

$$\begin{aligned} INV \quad = \quad & (1 \leq i \leq n+1) \wedge (ls = [a_i, \ldots, a_n]) \\ & \wedge \\ & ((\text{either } i \text{ is odd } \wedge l2 = [a_1, a_3, .., a_{i-2}] \wedge l1 = [a_2, a_4, ..., a_{i-1}]) \\ & (\text{or } i \text{ is even } \wedge l2 = [a_1, a_3, .., a_{i-1}] \wedge l1 = [a_2, a_4, ..., a_{i-2}])) \end{aligned}$$

⟨*Code for split*⟩≡

```
fun split(ls) =
   let
     fun splititer([],i,l1,l2) = (l1,l2)
       | splititer(x::ls,i,l1,l2) =
           if (i mod 2 = 0) then
                 splititer(ls,i+1,x::l1,l2)
           else
                 splititer(ls,i+1,l1,x::l2)
   in
     splititer(ls,1,[],[])
   end;
```

**Exercise 7.17** Establish the correctness of the functions `split` and `msort`.

We can calculate the number of steps required for `msort` as follows. Since the number of steps required for both split and merge is $n$, we can write a recurrence for the overall procedure as

$$\begin{array}{lcll} T(1) & = & 1 & \text{(we do not require any step for the singleton input)} \\ T(n) & = & 2T(n/2) + 2n & \end{array}$$

Assuming, for the time being, that $n = 2^m$ (a perfect power of two), we can solve the above recurrence by telescoping, i.e.,

$$\begin{array}{rcll} T(n) & = & 2T(2^{m-1}) + 2.2^m & \\ & = & 2(2T(2^{m-2}) + 2.2^{m-1}) + 2.2^m & = & 2.2T(2^{m-2}) + 4.2^m \\ & = & 2.2(2T(2^{m-3}) + 2.2^{m-2}) + 4.2^m & = & 2.2.2T(2^{m-3}) + 6.2^m \\ & \vdots & & & \\ & = & 2^m T(2^0) + 2m.2^m & & \\ & = & 2n\log_2 n = O(n\log_2 n) & & \end{array}$$

Hence, the worst case complexity of `msort` is better than that of `insort`. We will see later that this is the best we can do for sorting.

**Exercise 7.18** Show that the above result holds even if $n$ is not a perfect power of two.

**Example 7.12** Quick sort.
In the insertion sort algorithm described above we first split a list of size $n$ into a single item and a list of size $n-1$ and then insert the item into the recursively sorted sublist of size $n-1$. A popular alternative is *quick sort* where we first partition the given list in the form

$$[\ldots \text{all elements} \leq x \ldots, x, \ldots \text{all elements} > x \ldots]$$

where $x$ is the first element of the list. One can then recursively apply the sorting algorithm to the two sublists to the left and right of $x$.

We can use the higher order function `filter` defined above to collect the elements $\leq x$ into one list and $> x$ into another. We can then recursively apply *quick sort* to the two sub-lists and append them, with $x$ in between, to obtain the final result.

We can first define a higher order comparison function as

⟨*Code for comp*⟩≡
```
fun comp opr x y = opr(y,x):bool
```

and then define a function for *quick sort*, in terms of `comp` and `filter` as

⟨*qsort*⟩≡

```
fun qsort([]) = []
  | qsort(x::xs) =
      let
         ⟨Code for comp⟩
      in
         append(qsort(filter (comp op<= x) xs),x::qsort(filter (comp op> x) xs))
      end;
```

**Exercise 7.19** Establish the correctness of `qsort` using **PMI**.

**Exercise 7.20** Generalize `qsort` by converting it to a higher order function which takes a suitable comparison function as it's input and show how can one use the same higher order function to sort an arbitrary list in either ascending or descending order.

**Exercise 7.21** We have developed the `ML` function for insertion sort for sorting lists only in the ascending order. Modify the functions `insort` and `insert` into higher order functions like in the previous exercise.

The run time analysis of `qsort` is instructive. The `split` operation always takes $n$ steps. Since the value of the first element, `x`, is arbitrary, the two partitions we obtain are of sizes $i$ and $n-1-i$, where $i$ is any value between 0 and $n-1$ with equal probability. That is, if the partitioning element happens to be the smallest element in the list then $i$ is 0, and if it happens to be the largest element in the list then $i$ is $n-1$. For any other choice $i$ has a value in between. Now, in the extreme cases, if at every recursive stage the partitioning element is either the smallest or the largest, then, to solve a problem of size $n$ we have to solve two sub-problems of sizes 0 and $n-1$. This would obviously happen if the input list is either in increasing or in decreasing order. Note that the problem of size 0 has zero cost. In addition, we have to append the two sublists after attaching the partitioning element to the front of the second list. The cost of the append operation is proportional to the size of the first list. Thus, if the list is arranged in increasing order, the first partition is of size 0 and the second partition is of size $n-1$ and append has no cost. The only cost involved is attaching the partitioning element to the front of the second list and the cost of `split`. We can write a recurrence for the number of steps in quick-sort as

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(0) + T(n-1) + n + 1 = T(n-1) + n + 1 \end{aligned}$$

and we obtain $T(n) = O(n^2)$.

On the other hand, if the list is sorted in descending order, then for every recursive step the size of the first list is $n-1$ and that of the second is 0. In this case, append is costly and requires $n-1$ steps. In addition, we have the unit cost of attaching the partitioning element to the front of the second list and a cost $n$ for splitting the lists. In this case the recurrence becomes

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + T(0) + (n-1) + 1 + n = T(n-1) + 2n \end{aligned}$$

and we obtain $T(n) = O(n^2)$.

Thus we see that in case the input list is already in ascending or descending order, the number of steps required for `qsort` is identical to that for the worst case for `insort`. This is the worst case behaviour for `qsort`.

We are yet to analyze the in between cases. It may so happen, if we are lucky, that every time the partitioning element falls in the middle of the list and the two sublists generated out of partitioning are of roughly equal sizes. In such a case we have to solve two sub-problems of sizes roughly equal to $n/2$ and the append operation also requires $n/2$ steps. The recurrence is given as

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 2T(n/2) + n + n/2 \end{aligned}$$

We have obtained a *divide-and conquer* recurrence and it is easy to verify that this recurrence solves to $T(n) = O(n \log_2 n)$ and the behavior is similar to `msort`.

In most cases, however, god is neither so cruel nor so benevolent, and, we have an in between situation. We obtain splits of sizes $i$ and $n - i - 1$ with $i$ ranging from 0 to $n - 1$ with equal probability. Thus, the average cost of the two sub-problems that we have to solve can be written as

$$\frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)) = \frac{2}{n}\sum_{i=0}^{n-1}T(i)$$

Writing the cost of `split` and `append` as $cn$ for some constant $c$, we have the average case recurrence for `qsort` as

$$\begin{aligned} T(0) &= 0 \\ T(n) &= \frac{2}{n}\sum_{i=0}^{n-1}T(i) + cn \end{aligned}$$

We can multiply the second equation above with $n$ to obtain

$$nT(n) = 2\sum_{i=0}^{n-1}T(i) + cn^2$$

Telescoping, we can also write this as

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2}T(i) + c(n-1)^2$$

Subtracting the second equation from the first, to be rid of the summation, we obtain

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

The constant $-c$ is insignificant for the analysis and can be dropped. We obtain, after rearranging

$$nT(n) = (n+1)T(n-1) + 2cn$$

Dividing the above by $n(n+1)$ we obtain

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

Telescoping, we obtain,

$$\begin{array}{rcl} \frac{T(n-1)}{n} & = & \frac{T(n-2)}{n-1} + \frac{2c}{n} \\ \frac{T(n-2)}{n-1} & = & \frac{T(n-3)}{n-2} + \frac{2c}{n-1} \\ & \vdots & \\ \frac{T(2)}{3} & = & \frac{T(2)}{1} + \frac{2c}{3} \end{array}$$

Adding all the above yields

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i}$$

Now $\frac{T(1)}{2}$ is a constant and the summation to the right is bounded above by $\int_0^{n+1}\frac{1}{x}dx = O(\ln n)$ . Hence, clearly, $T(n) = O(n \ln n)$. Thus, the average case behaviour of `qsort` is closer to it's best case behaviour. In fact, on random data, `qsort` is one of the fastest sorting algorithms.

## 7.3 The $\alpha$–$LIST$ data-type in Java

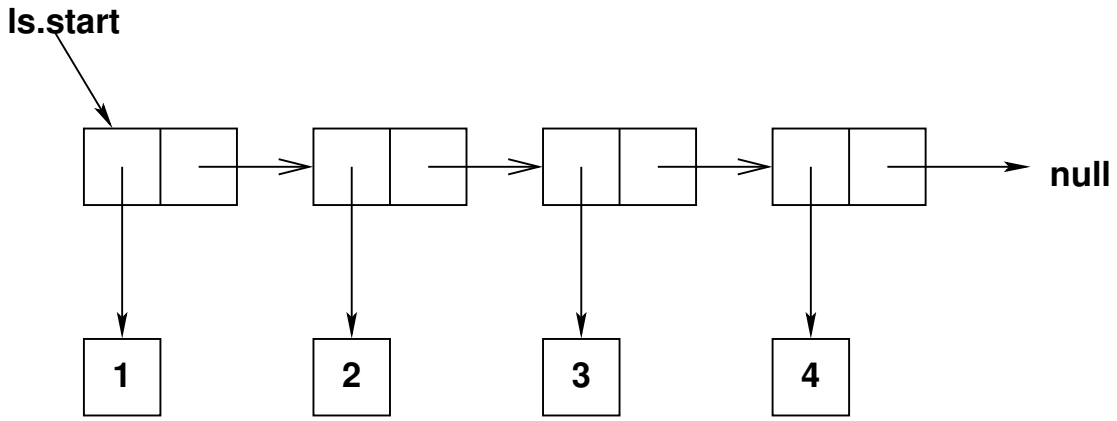


**Figure 7.1**: Linked-list representation of lists.

Building the list data-type in an imperative language like `Java` , which does not support it natively, is more complicated than in `ML` because the storage management has to be explicitly handled. We will represent an object of the type $\alpha{-}LIST$ using the popular *linked-lists* scheme common in imperative languages. Conceptually the representation can be viewed as in Figure 7.1. A *linked-lists* is a linear ordering of *nodes*. Each node is a compound object which stores one reference to an element (an integer object, for example) and another reference to the next node in the representation. The last node may be terminated by assigning the `Java` constant `null` to the *next-node* reference. Note that every variable in an imperative language like `Java` can be thought of as a reference to an *object* or a memory location. Thus the references, both to integer objects and to *next nodes* can be represented by simple `Java` variables. The *class* for a *node* can be defined in `Java` as

⟨*Code for Node class*⟩≡

```
class Node {
   Object data;
   Node link;
   Node(Object d,Node n) {
      data = d; link = n;
   }
}
```

The class defines two *instance variables*, `data` and `link`, of types `Object` and `Node`. These are references to a data element and the next node, respectively. `Object` is a key-word in `Java` used to indicate a polymorphic data-type (analogous to $\alpha$). Note that the definition is circular, because the type of the instance variable `link` is `Node` itself. However, conceptually there is nothing wrong in this circular definition and it is allowed in `Java` . The constructor `Node` takes two references, one to a data element and another to the next node, as input and creates and initializes the instance variables.

We now want to define a `Java` *class* for the abstract data type $\alpha$–*LIST* described at the beginning of this chapter. Apart from the internal representation, the class definition must also support a *constructor*, and the *methods* `empty`, `attach`, `head` and `tail`. Each *object* of the class `alphaList` needs to have only one *instance variable*, `start`, which is a reference to the first node in the list. Thus, we may define the `Java` class `alphaList` as follows.

⟨*class alphaList*⟩≡

```
public class alphaList {

   private Node start;

   ⟨Code for Node class⟩

   public alphaList() {
      start = null;
   }

   private static alphaList makeList(Node start) {
      alphaList T = new alphaList();
      T.start = start;
      return T;
   }

   public static alphaList newList(alphaList ls) {
      alphaList T = new alphaList();
      T.start = ls.start;
      return T;
   }

   ⟨Code for empty⟩
   ⟨Code for attach⟩
   ⟨Code for head⟩
   ⟨Code for tail⟩
}
```

The *instance variable* `start` is declared `private` because we want to hide our internal representation from external classes which may use `alphaList`. It is for the same reason that we have defined the *class* `Node` within the *class* `alphaList`. `Node` is an *inner class* of `alphaList` and is consequently hidden from all external classes.

From our experience with list programming in `ML` we know that we often require to create empty lists, or assign the value of one list to another. Accordingly, we have designed the basic *constructor* for the class `alphaList` so that it returns an empty list indicated by `start == null`. To be able to assign the value of one list to another we have defined the function `public static alphaList newList(alphaList ls)`. Note, from the definition, that `newList` creates a new list object, with it's own *instance variable*, which is then set to the *instance variable* of the old list. The two list objects then have different references (names), but the share the same set of nodes. We will discuss the consequences of sharing of nodes by two lists a little later Section 7.3.1. We also provide a function `private static alphaList makeList(Node start)` for creating new list starting from an arbitrary `start` node reference. The new list created by `makeList` will also share it's nodes with some other lists. The function is declared `private` because it directly uses the internal representation. Note that both `newList` and `makeList` are *class* methods because of the use of the key-word `static`. They cannot be *object* methods because their sole purpose is to create new list objects.

We can now develop the code for list methods as follows.

For the function `empty`, we have to merely check whether `start == null`.

⟨*Code for* `empty`⟩≡

```
public boolean empty() {
   return (start ==null);
}
```

For attaching an `Object x` to an list object we have to first create a new instance of `Node` (an object of type `Node`) and set it's data element to `x`. Then we have to set the `link` field of the new node so that it points to the original list and set the `start` field of the list object to the new node.

⟨*Code for* `attach`⟩≡

```
public void attach(Object x) {
   if (empty())
      start = new Node(x,null);
   else {
      Node T = new Node(x,start);
      start = T;
   }
}
```

`head` returns the reference of the data object of the node pointed at by the `start` of the list

⟨*Code for* `head`⟩≡

```
public Object head() {
   return start.data;
}
```

To return the `tail` of a list we create a new list whose `start` is `start.link` of the input list

⟨*Code for* `tail`⟩≡

```
public alphaList tail() {
   return makeList(start.link);
}
```

Thus, the complete code of the *class* `alphaList` can be given as

⟨*class alphaList*⟩+≡

```
public class alphaList {

   private Node start;

   class Node {
      Object data;
      Node link;
      Node(Object d,Node n) {
         data = d; link = n;
      }
   }

   public alphaList() {
      start = null;
   }

   private static alphaList makeList(Node start) {
      alphaList T = new alphaList();
      T.start = start;
      return T;
   }

   public static alphaList newList(alphaList ls) {
      alphaList T = new alphaList();
      T.start = ls.start;
      return T;
   }

   public boolean empty() {
      return (start ==null);
   }

   public void attach(Object x) {
      if (empty())
         start = new Node(x,null);
      else {
         Node T = new Node(x,start);
         start = T;
      }
   }
```

```
    public Object head() {
       return start.data;
    }

    public alphaList tail() {
       return makeList(start.link);
    }
 }
```

### 7.3.1 Sharing of lists and garbage collection

In the above design of `class alphaList` we have seen that often two different list objects may share some of their nodes. While this saves memory space one has to consider the possibility of side effects - changing data items of one list may mutate the other. This, however, is not a possibility with the class `alphaList`. No external methods can access any of the *instance variable* `start` of a list object, the `Node` objects and the method `makeList` because they are declared `private`. All other `public` methods or constructors create *new* lists and return, thereby ensuring that the input lists are preserved in their original form. Consider, for example, the instruction `l2 = alphaList.newList(l1)`. Though the two lists `l1` and `l2` share all the nodes they are two different list objects with their own `start` variables. A subsequent instruction, like `l2.attach(x)`, will create a new node whose `data` points to `x` and `link` points to the node referenced by the `start` variable of `l2`, and subsequently modify the `start` variable of `l2` to point to the new node. The `start` variable of `l1` continues to point to the same node. Thus, the integrity of both the lists are maintained enabling us to do `ML` style list programming.

Now, in imperative programming, we may wish to delete a list, `ls`, we have no use for. This may be achieved by setting `ls = null`. The `Java` run-time system has a powerful background process called *garbage collection* which automatically marks name-less objects (with no reference) as garbage and frees the memory occupied by it. Setting `ls` to `null` destroys the `start` variable of `ls` and the node pointed at by `start` (and, hence, all subsequent nodes) becomes *garbage*. If however, any of these nodes is referenced by the `start` variable of another list also, it cannot be marked as *garbage* because it is not name-less. Hence, all nodes prior to this node will be freed, but this and all subsequent nodes will remain intact. Because of the *garbage collection* facility is `Java` we do not have to bother about destroying objects we have created, except setting them to `null`. However, in other imperative programming languages we may have to design our own list destructors.

### 7.3.2 List programming in `Java`

Now that we have defined `class alphaList`, we are ready to translate the `ML` list programs that we have developed into `Java` . For implementing the sort functions our data items must support the comparison functions and hence must implement the following interface `Sortable`.

⟨*Interface Sortable*⟩≡

```
package myutils;
public interface Sortable {
   boolean lessthan(Sortable a);
   boolean equal(Sortable a);
}
```

We can now define a *class* called `ListFunctions` in which we can translate most of our ML programs.

⟨*class ListFunctions*⟩≡

```
package myutils;

public class ListFunctions {

   public static alphaList reverse(alphaList ls) {
      alphaList revlist = new alphaList();
      while (!ls.empty()) {
         Object x = ls.head();
         revlist.attach(x);
         ls = ls.tail();
      }
      return revlist;
   }

   public static alphaList append(alphaList l1, alphaList l2) {
      if (l1.empty())
         return alphaList.newList(l2);
      else {
         Object x = l1.head();
         l1 = l1.tail();
         alphaList ls = append(l1,l2);
         ls.attach(x);
         return(alphaList.newList(ls));
      }
   }

   public static alphaList insert(Sortable x, alphaList ls) {
      if (ls.empty()) {
         ls.attach(x); return ls;
      }
      else {
         Sortable hd = (Sortable) ls.head();
         if (x.lessthan(hd)) {
            ls.attach(x); return ls;
         }
         else {
            alphaList temp = insert(x,ls.tail());
            temp.attach(hd);
            return temp;
```

```
        }
      }
   }

   public static alphaList insort(alphaList ls) {
      if (ls.empty())
         return new alphaList ();
      else
         return insert((Sortable) ls.head(),insort(ls.tail()));
   }
}
```

Note that the functions in the `class ListFunctions` are almost literal translations of the `ML` programs developed earlier.

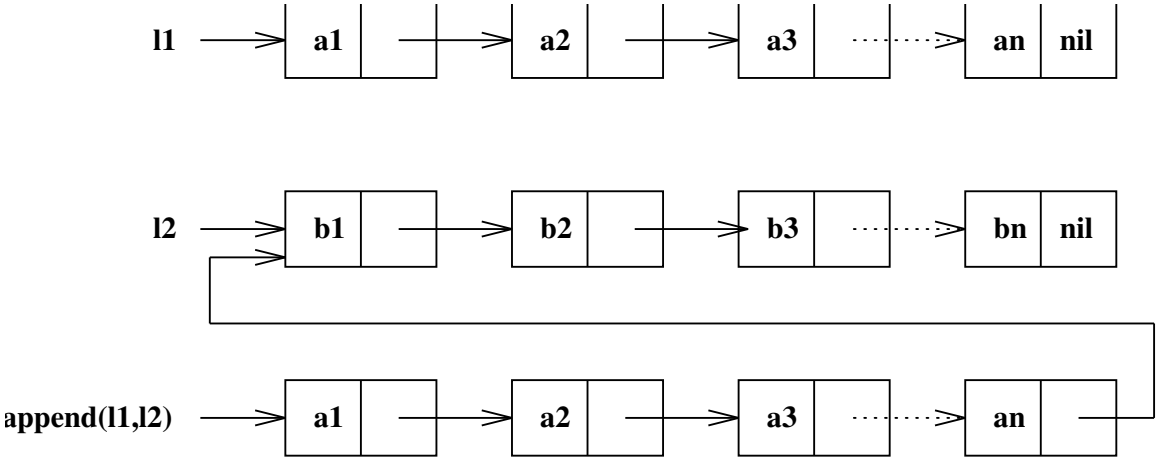


**Figure 7.2**: Appending two lists in `Java` using the function `ListFunctions.append`

It is instructive to consider the actual process of appending two lists with the function `ListFunctions.append`. The process is illustrated in Figure 7.2. The input lists `l1` and `l2` are not modified by the function `append` described above. A new copy of the list `l1` is made using successive calls to the function `attach` and the new list is attached to the front of `l2`. The combined list is returned by the function `append(l1,l2)`. Note that the lists `l1` and `l2` are left intact.

Similarly, the effect of inserting the element 4 in the list `[1,2,3,5]` using `ListFunctions.insert` is illustrated in Figure 7.3. Note that, like the function `ListFunctions.append`, the function `ListFunctions.insert` also generates a new list.

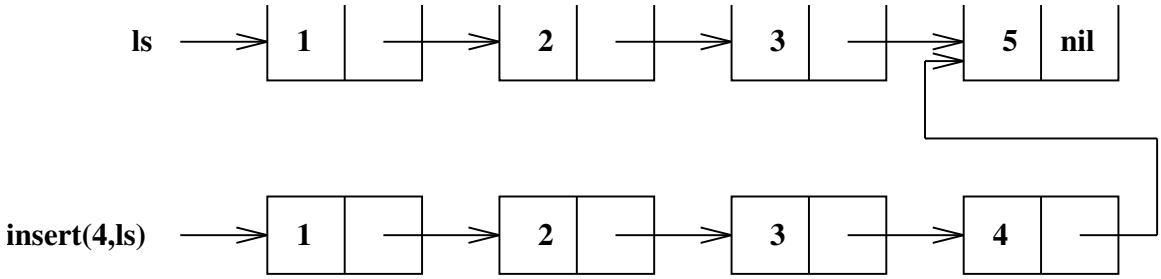


**Figure 7.3**: Inserting an element into a sorted lists in `Java` using the function `ListFunctions.insert`

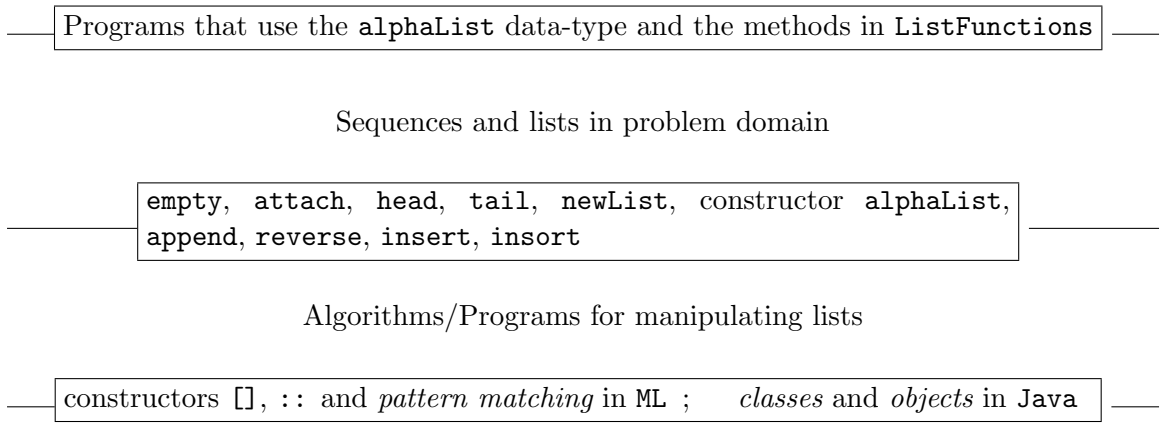


**Figure 7.4**: Abstraction barriers in the implementation of $\alpha$–*LIST*

We have designed the `Java` *classes* `alphaList` and `ListFunctions` using several abstraction barriers. The abstraction barriers in the implementation are illustrated in Figure 7.4. We can now define programs at a higher level, using the above functions, without any explicit reference to the details of the implementation.

In what follows we will give an example of top-level list programming in `Java` using the *classes* `alphaList` and `ListFunctions`. `Java` does not allow us to use the basic data-type `int` interchangeably with the polymorphic type `Object`. In view of this, we will need to define a new `Java` class, `Int`, to test our list programs. In what follows, we only describe the methods of `Int`. It implements `interface Sortable` which we have already described The details of the implementation are given in the appendix.

⟨*class Int*⟩≡

```
package myutils;
public class Int implements Sortable {
   public Int(int a); /* the constructor */
   public int toint(); /* object mehod to convert to int */
```

```
    public boolean lessthan(Sortable x); /* comparison method */
    public boolean equal(Sortable x); /* comparison method */
    public static alphaList readlist() throws IOException;
    public static void printlist(alphaList ls);
 }
```

We can define a top-level program as

⟨*a program to test list classes*⟩≡

```
import java.io.*;
import java.util.*;
import cs120.*;
import myutils.*;
public class listtest {


   public static void main(String args[])
      throws IOException {

      System.out.println("input a:");
      alphaList la = Int.readlist();

      System.out.println("input b:");
      alphaList lb = Int.readlist();

      System.out.print("a = ");
      Int.printlist(la);
      System.out.print("b = ");
      Int.printlist(lb);

      alphaList lc = ListFunctions.reverse(la);
      System.out.print("a = ");
      Int.printlist(la);
      System.out.print("c = ");
      Int.printlist(lc);

      lc = ListFunctions.append(la,lb);
      System.out.print("c = ");
      Int.printlist(lc);

      alphaList ld = ListFunctions.insort(lc);
      System.out.print("d = ");
      Int.printlist(ld);
   }
}
```

**Exercise 7.22** Execute the above program.

**Exercise 7.23** Add the functions *merge sort* and *quick sort* to the *class* `ListFunctions`

## Problems

Implement the following in both `ML` and `Java` .

1. Develop an abstract data-type called *set*. Represent a set of integers as a list (there should be no duplications) and develop functions for

   (a) Adding a new element to a set.

   (b) Checking whether an element belongs to a set.

   (c) Finding the intersection of two sets.

   (d) Finding the union of two sets.

   Estimate the time complexity of each operation.

2. Represent a set as an ordered list of integers and develop functions for each of the above operations. The time complexity of each of the above functions should be linear.

3. Consider a representation of univariate polynomials as lists of pairs of the type (*coefficient, exponent*). For example, the polynomial $7x^8 + 3x^4 + x + 5$ may be represented as the list $[[7, 8], [3, 4], [1, 1], [5, 0]]$. Assume that the polynomials are in their canonical form, i.e, the exponents are in the decreasing order.

   (a) Develop algorithms/functions for adding and multiplying two polynomials. Estimate the time complexities of your algorithms.

   (b) Develop `Java` function/procedure for reading and printing a polynomial.

## Appendix: The `Int` class

In what follows we give the code for the *class* `Int`.

⟨*class Int*⟩≡

```
package myutils;

import java.io.*;
import java.util.*;
import cs120.*;

public class Int implements Sortable {
   private int val;
   public Int(int a) {
      val = a;
   }
   public int toint() {
      return val;
   }
   public boolean lessthan(Sortable x) {
      Int n = (Int) x;
      return (val < n.val);
   }
   public boolean equal(Sortable x) {
      Int n = (Int) x;
      return (val == n.val);
   }
   public static alphaList readlist() throws IOException {
      StringTokenizer T;
      int i,ntokens,c,n;
      String s;
      Int x;
      alphaList ls = new alphaList();
      BufferedReader in = Text.open(System.in);
      s = in.readLine();
      T = new StringTokenizer(s);
      ntokens = T.countTokens();
      if (ntokens != 0) {
         i = 0;
         while (i < ntokens) {
            i++;
            c = Integer.parseInt(T.nextToken());
            x = new Int(c);
            ls.attach(x);
         }
```

```
      }
      return ListFunctions.reverse(ls);
   }
   public static void printlist(alphaList ls) {
      while (!ls.empty()) {
         Int x = (Int) ls.head();
         System.out.print(x.val+" ");
         ls = ls.tail();
      }
      System.out.println();
   }
   public static void readarray(Int a[],int n) throws IOException {
      StringTokenizer T;
      int i,ntokens,c;
      String s;
      Int x;
      BufferedReader in = Text.open(System.in);
      s = in.readLine();
      T = new StringTokenizer(s);
      ntokens = T.countTokens();
      if (ntokens >= n) {
         i = 0;
         while (i < n) {
            c = Integer.parseInt(T.nextToken());
            x = new Int(c);
            a[i] = x;
            i++;
         }
      }
   }
   public static void printarray(Int a[],int n) {
      for(int i=0; i < n;i++) System.out.print(a[i].val + " ");
      System.out.println();
   }
}
```

# Chapter 8

# Computing with arrays

Array is a static data structure, used commonly in imperative settings, to represent sequences of finite size. 'Static' means that an array cannot dynamically grow or shrink in size during runtime, and its size must be pre-declared in a program.

In Figure 8.1 we give an example of an array that stores five elements, $a[0], a[1], \ldots, a[4]$. The elements $a[i]$ of an array – indexed by $i$ – can be of an arbitrary type, but the type has to be pre-specified. Unlike a list, an array is a *random access* structure, which means that any of its elements can be accessed in $O(1)$ time using the array indices. In Figure 8.2, we give an example of an array of integers, where the first element, 3, can be accessed as $a[0]$; the third, 9, as $a[2]$; and the fourth, 7, as $a[3]$.

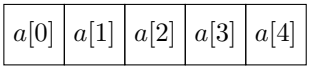


**Figure 8.1**: An example of an array with five elements, $a[0], a[1], \ldots, a[4]$.

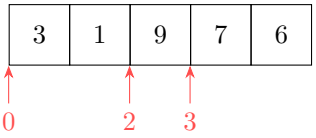


**Figure 8.2**: Accessing array elements using indices.

In essence, an array is a fixed-size collection of imperative style variables, which may be accessed through the array indices. In this chapter we will develop some examples of programming with arrays. We will use `Python` to illustrate our examples. It is to be noted though that `Python` arrays are actually dynamic structures which also support list functions. We will however restrict ourselves to using the `Python` data structure as a

traditional array.

## 8.1 Linear scan of arrays

**Example 8.1** Finding a maximum element in an array of positive integers.

Finding a maximum element in an array `a` in the segment `l..r` may be achieved by scanning the array from left to right as indicated in Figure 8.3.

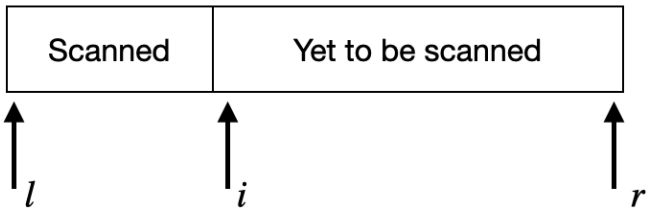


**Figure 8.3**: Scanning an array from left to right

We can write the input and output assertions as follows:

⟨*Maximum in an array*⟩≡
```
def maximum(a,l,r):
    #assert: array a[l..r] is established
    ⟨Code for finding the maximum⟩
    #assert: p such that max = a[p] is the largest element in a[l..r]
    return (max,p)
```

For developing the code for finding the maximum, we can derive the following invariant guided by Figure 8.3 and by relaxing the final assertion above:

⟨*Invariant for finding the maximum*⟩≡
```
#INV: (l <= i <= r+1) and (max is the largest element in a[l..i-1])
#      and (max = a[p] if l <= p <= i-1)
```

Note that the invariant is derived from the final assertion by replacing the constant `r` with `i-1`. Also note that substituting `i` by `r+1` in the invariant gives us our final assertion.

The invariant may be initialised with:

⟨*Initialisation*⟩≡
```
max,i = 0,l # set max to less than all elements in the array
p = -1 # denoting undefined
```

Note, that at initialisation, the range `l..l-1` is empty, and the claim that `max is the largest element in a[l..i-1]` is vaccuously true.

Hence, the function may be completed with:

⟨*Code for finding the maximum*⟩≡

```
⟨Initialisation⟩
⟨Invariant for finding the maximum⟩
while (i <= r):
   if (a[i] > max):
      max,p = a[i],i
   i = i+1
```

The function can be tested with the following code segment in the main program:

⟨*Main program (maximum)*⟩≡

```
import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
(max,p) = maximum(A,0,9)
print(max,p)
```

Thus, we may write the overall code as:

⟨*The complete code*⟩≡

```
def maximum(a,l,r):
   #assert: array a[l..r] is established
   max,i = 0,l # set max to less than all elements in the array
   p = -1 # denoting undefined
   #INV: (l <= i <= r+1) and (max is the largest element in a[l..i-1])
   #      and (max = a[p] if l <= p <= i-1)
   while (i <= r):
      if (a[i] > max):
         max,p = a[i],i
      i = i+1
   #assert: p such that max = a[p] is the largest element in a[l..r]
   return (max,p)

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
(max,p) = maximum(A,0,9)
print(max,p)
```

Finding a maximum in an array using left to right scanning obviously requires $O(n)$ comparisons for $n$ elements in an array.

**Example 8.2** Checking whether an array is a palindrome.

An array is a *palindrome* if it reads the same backwards as forwards. For example, for example the arrays $[1, 2, 3, 2, 1]$ and $[1, 2, 2, 1]$ are both palindromes. Our task is to scan the array to check if a given array is a palindrome or not.

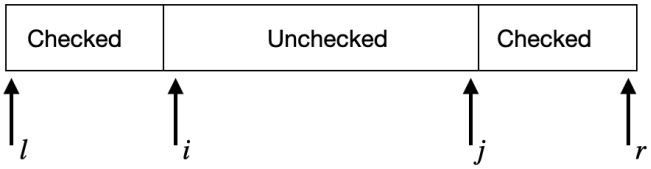


**Figure 8.4**: Checking is an array is a palindrome.

A simple algorithmic idea would be to scan the array starting from both ends – indexed by $l$ and $r$ respectively – as suggested in the invariant depicted in Figure 8.4, and checking if the corresponding elements are identical. The scanning should stop when there are no more elements to check.

We can write the input and output assertions as follows:

⟨*Palindrome*⟩≡
```
def palindrome(a,l,r):
  #assert: a[l..r] established
  ⟨Code for checking palindrome⟩
  #assert: a[l..i-1] is reverse of a[j+1..r], i >= j
  return True
```

Note, according to Figure 8.4, that the condition `i >= j` implies that the unchecked part is empty and there is nothing left to be scanned.

We can develop the loop invariant for scanning the array by relaxing the final assertion as

⟨*Invariant*⟩≡
```
#INV: a[l..i-1] is reverse of a[j+1..r], l <= i <= (l+r)//2 + 1,
#     (l+r)//2 -1 <= j <= r, i <= j+1
```

The invariant may be initialised as:

⟨*Initialisation*⟩≡
```
i,j = l,r
```

Note that the segments `l..i-1` and `j+1..r` are empty at the intialisation, and hence the invariant is vaccuously true.

The loop should terminate immediately if the check for the corresponding elements fails at any stage. Thus, the code for the while loop follows from the above as

⟨*Code for checking palindrome*⟩≡

```
⟨Initialisation⟩
⟨Invariant⟩
while (i < j):
  if (a[i] != a[j]):
    return False
  else:
    i,j = i+1,j-1
```

The `Python` function can be tested with the followind code segement in the main program

⟨*Main*⟩≡

```
import random

⟨Palindrome⟩
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
print(palindrome(A,0,9))
B = [1,2,3,2,1]
print(palindrome(B,0,4))
C = [1,2,2,1]
print(palindrome(C,0,3))
```

The complete code may then be given as:

⟨*Complete code for palindrome*⟩≡

```
import random

def palindrome(a,l,r):
  #assert: a[l..r] established
  i,j = l,r
  #INV: a[l..i-1] is reverse of a[j+1..r], l <= i <= (l+r)//2 + 1,
  #     (l+r)//2 -1 <= j <= r, i <= j+1
  while (i < j):
    if (a[i] != a[j]):
      return False
    else:
      i,j = i+1,j-1
  #assert: a[l..i-1] is reverse of a[j+1..r], i >= j
  return True
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
print(palindrome(A,0,9))
B = [1,2,3,2,1]
print(palindrome(B,0,4))
C = [1,2,2,1]
print(palindrome(C,0,3))
```

## 8.2 Searching for an element in an array

**Example 8.3** Sequential search.

*Sequential search* is the problem of searching for a given key `x` in an unordered array `a[l..r]`. We can use the left to right scanning strategy of Example 8.1 to look for the element `x` in the array segment.

The input and output assertions, and the final check can be written as:

⟨*Sequential search*⟩≡

```
def seqsearch(a,l,r,x):
   # assert: a[l..r] is established, x is established
   ⟨Code for searching⟩
   # assert: found <=> (a[i] = x), l <= i <= r
   if found:
      return i
   else:
      return -1 # indicating not found
```

We can derive the invariant by relaxing the final assertion as before

⟨*Invariant*⟩≡
```
#INV: (l <= i <= r+1) and (not found <=> (x not in a[l..i-1]))
```

and we can initialise the invariant as follows

⟨*Initialisation*⟩≡
```
found,i = False,l
```

The invariant and the initialisation suggests the following while loop

⟨*Code for searching*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while ((not found) and (i <= r)):
   if (a[i] == x):
      found = True
   else:
      i = i+1
```

The overall code may then be written as

⟨*Python code for sequential search*⟩≡
```
def seqsearch(a,l,r,x):
   # assert: a[l..r] is established
   found,i = False,l
   #INV: (l <= i <= r+1) and (not found <=> (x not in a[l..i-1]))
   while ((not found) and (i <= r)):
      if (a[i] == x):
         found = True
      else:
         i = i+1
   # assert: found <=> (a[i] = x), l <= i <= r
   if found:
      return i
   else:
      return -1 # indicating not found

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
x = int(input("x = "))
print(x)
print(seqsearch(A,0,9,x))
```

**Example 8.4** Binary search.

The search problem becomes computationally more efficient if the array segment is sorted. Then, if the middle element is less than the key, then all elements to the left of and including the middle element can be eliminated from the scope of the search. Similarly, we can element the right side if the middle element turns out to be greater than the key. Either way, with a single comparison, we we are left with a sub-problem of half size. This gives us the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{for } n > 1 \end{cases}$$

which evaluates to $O(\lg n)$ (we already know this from Example 3.9).

We can then write the input and output assertions, and the final check as

⟨*Binary search*⟩≡
```
def binsearch(a,l,r,x):
    # assert: a[l..r] is established and is sorted in ascending order
    ⟨Code for searching⟩
    # assert: found <=> (a[mid] = x)), l <= mid <= r
    if found:
        return mid
    else:
        return -1 # indicating not found
```

and we can derive the invariant for the iterative process as

⟨*Invariant*⟩≡
```
#INV: (not found) <=> (x not in a[l..left-1] and a[right+1..r])
#     l <= left <= right+1, left-1 <= right <= r
```

which indicates that at an intermediate stage of the computation the scope of the search is bracketed between `left..right`. We can specify the initialisation as

⟨*Initialisation*⟩≡
```
found = False
left,right = l,r
```

The code for the while loop can then be derived from the invariant in a strightforward manner

⟨*Code for searching*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while ((not found) and (left <= right)):
    mid = (left+right)//2
    if (a[mid] == x):
        found = True
    elif (a[mid] < x):
        left = mid + 1
    else:
        right = mid - 1
```

The overall code may then be written as

⟨*Python code for binary search*⟩≡

```
def binsearch(a,l,r,x):
   # assert: a[l..r] is established and is sorted in ascending order
   found = False
   left,right = l,r
   #INV: (not found) <=> (x not in a[l..left-1] and a[right+1..r])
   #     l <= left <= right+1, left-1 <= right <= r
   while ((not found) and (left <= right)):
      mid = (left+right)//2
      if (a[mid] == x):
         found = True
      elif (a[mid] < x):
         left = mid + 1
      else:
         right = mid - 1
   # assert: found <=> (a[mid] = x)), l <= mid <= r
   if found:
      return mid
   else:
      return -1 # indicating not found

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
A.sort()
print(A)
x = int(input("x = "))
print(x)
print(binsearch(A,0,9,x))
```

## 8.3 Sorting

Without loss of generality, we will consider sorting in ascending order. To begin with, we will consider sorting with compare and exchange as the basic primitives.

**Example 8.5** Selection sort.

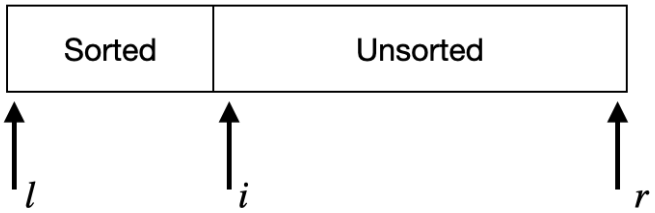


**Figure 8.5**: The main loop for selection sort

Perhaps the simplest method of sorting is to divide the the given array into a sorted part and an unsorted part, as shown in Figure 8.6, and work on the unsorted part. The sorted part can be empty to start with. Then, we can find the minimum element in the unsorted part and swap it with the first element of the unsorted part, $i$, and increment $i$ by one. The resulting method is called *selection sort*.

We can specify the overall structure for sorting as

⟨*Selection sort*⟩≡
```
def SelectionSort(a,l,r):
    #assert: a[l..r] is established
    ⟨Code for selection sort⟩
    #assert: a[l..r] is sorted in ascending order
```

Note, that in the above function, we do not need to explicitly return the final sorted array `a`. This is because unlike a simple variable, only a memoty reference for the array is passed to the function (`SelectionSort` in this case), and hence any changes made to the array from within the function reflects in the calling function as well. Because only a reference is passed, the array is actually shared between the two functions.

We can capture the basic idea in Figure 8.6 with the invariant derived from the final assertion as

⟨*Invariant*⟩≡
```
#INV: a[l..i-1] is sorted in ascending order; (l <= i <= r+1)
#     all a[l..i-1] <= all a[i..r]
```

The invariant can be initialised with

⟨*Initialisation*⟩≡
```
i = l
```

The claim that `a[l..l-1]` is sorted is vaccuously true.

Then, to maintain the invariant, we have to find the minimum element in `a[i..r]` and swap it with `a[i]` before we set `i = i+1`. This suggests the following

⟨*Code for selection sort*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while (i <= r):
   ⟨Code for finding the minimum in the unsorted part⟩
   #assert: p such that a[p] <= a[i..r], i <= p <= r
   a[p],a[i] = a[i],a[p]
   i = i+1
```

We are now left with the task of finding the minimum in the unsorted part, whose precise objective is given in the `assert` in the above code segment. We can derive the invariant for this task by relaxing the assertion

⟨*Invariant for the inner loop*⟩≡
```
#INV: p such that a[p] <= a[i..j-1], i <= p <= j-1,
#     i+1 <= j <= r+1
```

We can initialise this invariant with

⟨*Initialisation for the inner loop*⟩≡
```
p,j = i,i+1
```

Finally, the code for the inner loop can be delveloped as

⟨*Code for finding the minimum in the unsorted part*⟩≡
```
⟨Initialisation for the inner loop⟩
⟨Invariant for the inner loop⟩
while (j <= r):
   if (a[j] < a[p]):
      p = j
   j = j+1
```

We can test the code with

⟨*Main*⟩≡
```
⟨Selection sort⟩

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
SelectionSort(A,0,9)
print(A)
```

The whole code may then be given as

⟨*The complete code*⟩≡

```
def SelectionSort(a,l,r):
   #assert: a[l..r] is established
   i = l
   #INV: a[l..i-1] is sorted in ascending order; (l <= i <= r+1)
   #     all a[l..i-1] <= all a[i..r]
   while (i <= r):
      p,j = i,i+1
      #INV: p such that a[p] <= a[i..j-1], i <= p <= j-1,
      #     i+1 <= j <= r+1
      while (j <= r):
         if (a[j] < a[p]):
            p = j
         j = j+1
      #assert: p such that a[p] <= a[i..r], i <= p <= r
      a[p],a[i] = a[i],a[p]
      i = i+1
   #assert: a[l..r] is sorted in ascending order

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
SelectionSort(A,0,9)
print(A)
```

The correctness of the selection sort algorithm follows from an analysis of the invariants. To estimate the runtime complexity, we note that to solve a sorting problem of size $n$, we need to solve a sorting sub-problem of size $n-1$ (without the minimum), and finding the minimum requires $O(n)$ comparisons. This gives us the following recurrence

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + O(n) & \text{otherwise} \end{cases}$$

which solves to $O(n^2)$.

*Bubble sort* and *insertion sort* are two other sorting methods that differ from *selection sort* only in the inner loop. We describe them next.

**Example 8.6** Bubble sort.

The outer loop assertions, invariant and insertions of *selection sort* remain the same for *bubble sort*.

⟨*BubbleSort*⟩≡
```
def BubbleSort(a,l,r):
   #assert: a[l..r] is established
   i = l
   #INV: a[l..i-1] is sorted in ascending order; (l <= i <= r+1)
   #     all a[l..i-1] <= all a[i..r]
   while (i <= r):
      ⟨Code for the inner loop⟩
      #assert: a[i] <= all a[i+1..r]
      i = i+1
   #assert: a[l..r] is sorted in ascending order
```

Instead of explicitly finding the minimum in the unsorted part, in the inner loop of *bubble sort* the minimumm element is rippled though to the `i`-th position through a process of pairwise comparisons. We can start from `r` and and swap down any element that is smaller than its immediately previous one. This gives us the following invariant:

⟨*Invariant*⟩≡
```
#INV: a[j] <= all a[j+1..r], i <= j <= r
```

The invariant can be initialised with

⟨*Initialisation*⟩≡
```
j = r
```

The inner loop may then be completed as

⟨*Code for the inner loop*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while (j > i):
   if (a[j] < a[j-1]):
      a[j],a[j-1] = a[j-1],a[j]
   j = j - 1
```

The complete code for *bubble sort* may then be given as

⟨*Code for bubble sort*⟩≡

```
def BubbleSort(a,l,r):
   #assert: a[l..r] is established
   i = l
   #INV: a[l..i-1] is sorted in ascending order; (l <= i <= r+1)
   #     all a[l..i-1] <= all a[i..r]
   while (i <= r):
      j = r
      #INV: a[j] <= all a[j+1..r], i <= j <= r
      while (j > i):
         if (a[j] < a[j-1]):
            a[j],a[j-1] = a[j-1],a[j]
         j = j - 1
      #assert: a[i] <= all a[i+1..r]
      i = i+1
   #assert: a[l..r] is sorted in ascending order

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
BubbleSort(A,0,9)
print(A)
```

The time complexity of *bubble sort* is identical to that of *selection sort*. However, the inner loop in the above program may turn out to be wasteful if the array is already sorted. In such a case, there will be no swaps in the inner loop.

**Exercise 8.1** *Modify the above program to enable an early exit if no swap is required in the inner loop.*

**Example 8.7** Insertion sort.

In *isertion sort* we take out the first element of the unsorted part of the array and insert it in its correct position in the sorted part of the array. In a significant departure from the previous two algorithms, we no longer claim that `all a[l..i-1] <= all a[i..r]` in the outer loop invariant. The outer loop, with its invariant condiiton, is given as

⟨*InsertionSort*⟩≡
```
def InsertionSort(a,l,r):
   #assert: a[l..r] is established
   i = l
   #INV: a[l..i-1] is sorted in ascending order; (l <= i <= r+1)
   while (i <= r):
      x = a[i]
      ⟨Code for the inner loop⟩
      #assert: a[l..j-1] sorted, all a[l..j-1] <= x,
      #  a[j+1..i] sorted, all a[j+1..i] > x
      #       l <= j <= i
      a[j] = x
      i = i+1
   #assert: a[l..r] is sorted in ascending order
```

For the inner loop, we make space for insertion by pulling out `a[i]` into `x`, and then find the correct position `j` to insert `x` back. We can derive the invariant for the inner loop as

⟨*Invariant*⟩≡
```
#INV: a[l..j-1] sorted, a[j+1..i] sorted and all a[j+1..i] > x, l <= j <= i
```

We can then complete the initialisation with

⟨*Initialisation*⟩≡
```
j = i
```

the while loop can then be given as

⟨*Code for the inner loop*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while ((j > l) and (a[j-1] > x)):
   a[j] = a[j-1]
   j = j-1
```

The complete code for *insertion sort* is then given as:

⟨*Code for* insertion sort⟩≡

```
def InsertionSort(a,l,r):
   #assert: a[l..r] is established
   i = l
   #INV: a[l..i-1] is sorted in ascending order; (l <= i <= r+1)
   while (i <= r):
      x = a[i]
      j = i
      #INV: a[l..j-1] sorted, a[j+1..i] sorted and all a[j+1..i] > x, l <= j <= i
      while ((j > l) and (a[j-1] > x)):
         a[j] = a[j-1]
         j = j-1
      #assert: a[l..j-1] sorted, all a[l..j-1] <= x,
      #  a[j+1..i] sorted, all a[j+1..i] > x
      #       l <= j <= i
      a[j] = x
      i = i+1
   #assert: a[l..r] is sorted in ascending order

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
InsertionSort(A,0,9)
print(A)
```

Since the inner loop requires $O(n)$ operations, the worst case time complexity remains unchanged from that of *selection sort* and *bubble sort*. The correctness follows from the assertions and invariants.

**Exercise 8.2** *Excecute the three sorting programs* – selection sort, bubble sort *and* insertion sort *for large random input arrays and prepare a table of the run-times. Can you explain the discrepancies in the run-times?*

**Example 8.8** Quick sort.

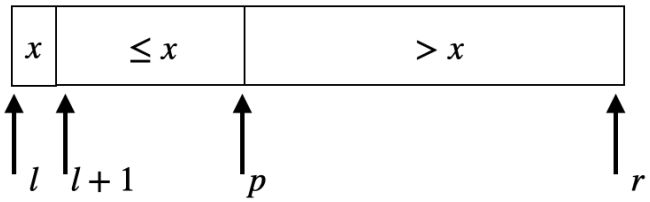


**Figure 8.6**: Partition quick sort.

All the above sorting algorithms have a time complexity of $O(n^2)$, even in the average case. In Example **??** we discussed *quick sort*, and argued that its average case time complexity is $O(n \log n)$. We now implement *quick sort* with arrays.

Without loss of generality, we can pick the first element `a[l]` of the array as the random pivot element `x`, and partition `a[l+1..r]` about `x`. We can then swap `a[l]` with `a[p]`, the last element of the first partition, and quick sort `a[l..p-1]` and `a[p+1..r]` recursively.

This gives us the following overall structure

⟨*Quick sort*⟩≡
```
⟨Code for partition⟩
def QuickSort(a,l,r):
   #assert: a[l..r] established
   if (l < r):
      x = a[l]
      p = partition(a,l+1,r,x)
      #assert: all a[l+1..p] <= x, all a[p+1..r] > x, l+1 <= p <= r
      a[l],a[p] = a[p],a[l]
      QuickSort(a,l,p-1)
      QuickSort(a,p+1,r)
   #assert: a[l..r] sorted in ascending order
```

The code for partition can be developed from the above as

⟨*Code for partition*⟩≡
```
def partition(a,left,right,x):
   #assert: a[left..right] established
   ⟨partition algorithm⟩
   #assert: all a[left..p] <= x, all a[p+1,,right] > x,
   #        left <= p <= right
   return p
```

The invariant for the parition algorithm can be derived by relaxing the final assertion as

⟨*Invariant*⟩≡
```
#INV: all a[left..i-1] <= x, all a[j+1,,right] > x
#     left <= i <= j+1, i-1 <= j <= right
```

This can be initialised with

⟨*Initialisation*⟩≡
```
i,j = left,right
```

The function `partition` can then be completed as

⟨*partition algorithm*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while (i <= j):
   if (a[i] <= x):
      i = i+1
   else:
      a[i],a[j] = a[j],a[i]
      j = j-1
p = j
```

The complete code can then be written as

⟨*Complete code for* quick sort⟩≡

```
def partition(a,left,right,x):
   #assert: a[left..right] established
   i,j = left,right
   #INV: all a[left..i-1] <= x, all a[j+1,,right] > x
   #     left <= i <= j+1, i-1 <= j <= right
   while (i <= j):
      if (a[i] <= x):
         i = i+1
      else:
         a[i],a[j] = a[j],a[i]
         j = j-1
   p = j
   #assert: all a[left..p] <= x, all a[p+1,,right] > x,
   #        left <= p <= right
   return p

def QuickSort(a,l,r):
   #assert: a[l..r] established
   if (l < r):
      x = a[l]
      p = partition(a,l+1,r,x)
      #assert: all a[l+1..p] <= x, all a[p+1..r] > x, l+1 <= p <= r
      a[l],a[p] = a[p],a[l]
      QuickSort(a,l,p-1)
      QuickSort(a,p+1,r)
   #assert: a[l..r] sorted in ascending order

import random
A = [random.randint(1,10) for _ in range(10)] # generate a random array of 10 elements
print(A)
QuickSort(A,0,9)
print(A)
```

## 8.4 Some other array algorithms

**Example 8.9** The sieve of Eratosthenes.

The *sieve of Eratosthenes*[1] is a systematic procedure for finding all prime numbers upto a given integer $n$. It begins by arranging the numbers $2, 3, \ldots, n$ is ascending order, and then strikes out every second number following the number 2, every third number following the number 3, and so on. Of course, it skips the composites that are already struck out in the process. When the procedure ends, what remain are the prime numbers.

The algorithm is often described through the following poem

*Sift the twos and sift the threes*
*the Sieve of Eratosthenes*
*when the composites sublime*
*what remain are prime.*

To develop a procedure for the sieve process, we can write the input and output assertions as

⟨*Sieve of Eratosthenes*⟩≡
```
def sieve(a,n):
    #assert:  for k in [2..n] , a[k] = k
    ⟨Code for sieve⟩
    #assert: for k in [2..n] , a[k] = 0 if and only if p|k
    #        for some  prime number p <= sqrt(n)
```

Note, that in the above assertion, `p|k` denotes `p` divides `k`.

We can derive the invariant for the procedure by relaxing the output assertion as

⟨*Invariant*⟩≡
```
#INV:  for k in [2..n] , a[k] = 0 if and only if
#      p|k and p is a prime <= j
```

which leads to the initialisation

⟨*Initialisation*⟩≡
```
j = 2
```

the code for sieve can then be given as

⟨*Code for sieve*⟩≡
```
⟨Initialisation⟩
⟨Invariant⟩
while (j*j <= n):
    ⟨Code for setting multiples of j to 0⟩
    j = j+1
    ⟨Code for finding the next prime⟩
```

[1]Eratosthenes, in full Eratosthenes of Cyrene, (born c. 276 BCE, Cyrene, Libya—died c. 194 BCE, Alexandria, Egypt), was a Greek scientific writer, astronomer, and poet, who made the first measurement of the size of Earth for which any details are known. One of his major contribution was the sieve algorithm to find prime numbers.

the next prime may be found by the looking for the next non-zero element in the array

⟨*Code for finding the next prime*⟩≡
```
while (a[j] == 0):
   j = j+1
```

The output assertion for setting all multiples of `j` to 0 is

⟨*inner loop assertion*⟩≡
```
#assert: for k in [j*j..n] a[k] = 0 if j|k
```

which can be relaxed to obtain the invariant as

⟨*Invariant for setting multiples of j to 0*⟩≡
```
#INV: for k in [j*j..l] a[k] = 0 if and only if
#     j|k, j*j <= l <= n+j
```

which can be initialised as

⟨*Initialisation for the inner loop*⟩≡
```
l = j*j
```

The inner loop may then be written as

⟨*Code for setting multiples of j to 0*⟩≡
```
⟨Initialisation for the inner loop⟩
⟨Invariant for setting multiples of j to 0⟩
while (l <= n):
   a[l] = 0
   l = l+j
⟨inner loop assertion⟩
```

This gives us the overall procedure with a driver code as

⟨*Complete code for the Sieve of Eratosthense*⟩≡

```
def sieve(a,n):
   #assert:  for k in [2..n] , a[k] = k
   j = 2
   #INV:  for k in [2..n] , a[k] = 0 if and only if
   #      p|k and p is a prime <= j
   while (j*j <= n):
      l = j*j
      #INV: for k in [j*j..l] a[k] = 0 if and only if
      #     j|k, j*j <= l <= n+j
      while (l <= n):
         a[l] = 0
         l = l+j
      #assert: for k in [j*j..n] a[k] = 0 if j|k
      j = j+1
      while (a[j] == 0):
         j = j+1
   #assert: for k in [2..n] , a[k] = 0 if and only if p|k
   #        for some  prime number p <= sqrt(n)

from numpy import *
n = 500
a = zeros([n+1],int)
j = 2
while (j <= n):
   a[j] = j
   j = j+1
sieve(a,n)
print(a)
```

Unfortunately, the time complexity of the algorithm is not very good.

**Exercise 8.3** *Argue that the time complexity of the algorithm is exponential in the number of digits of $n$.*

# Chapter 9

# Computing with trees

# Part III

# Algorithmic frameworks

# Chapter 10

# Divide and conquer algorithms

# Chapter 11

# Depth first search and acktracking algorithms

# Chapter 12

# Numerical algorithms

# Chapter 13

# Randomized algorithms