Chapter 1

Introduction

Understanding a scientific domain consists of a healthy dosis of reasoning along with a vast body of knowledge. Knowing lots of individual facts without having good reasoning capabilities does not bring one very far. A lawyer that has a perfect knowledge of all the laws in his country, but who is not smart enough to apply them to a concrete case will generally not be considered a good lawyer. However, being extremely smart without having the factual knowledge accumulated by previous thinkers will require one to accumulate that knowledge all over again. It is sometimes said that those who do not know history are bound to live it all over again. The same goes for the knowledge accumulated in a scientific field and thus also in the field of computer science.

This course presents an important slice of factual knowledge that any computer scientist ought to have acquired during his academic training. It provides students with an encyclopedic overview of programming knowledge that has been accumulated by computer scientists over the past six decades. That knowledge concerns both knowledge about data structures and knowledge about computations that operate on these structures. These computations are also termed *algorithms*. As we will see, there exist a virtual infinite number of algorithms and a virtually infinite number of ways to construct data structures in a computer memory. On first sight, every time a programmer writes a new program, he has to think about the way his data will be structured in memory. Similarly, on first sight, every time a programmer writes a new program, he has to think about how to organise that program from scratch. However, if this were true, we would not be able to speak about computer science as a science. Doing everything from scratch every time again is known a craftsmanship as opposed to science. On second sight, however, there are quite a number of data structures and many algorithms that seem to be popping up over and over again when writing programs. It is therefore useful to bundle them in a catalogue of algorithms and data structures and to teach that catalogue to newly bred computer scientists. This is what this course is all about. The chapters of the course present a systematic overview of "the" standard textbook algorithms and data structures. But if there exists something like "the" standard textbook

set of algorithms, then why do we need (yet) another textbook on this subject matter? The reason is that a course on algorithms and data structures somehow has to *precisely* specify those algorithms and explicitly show how the data structures are effectively created in a real computer memory. This is done in a *programming language*. Although there exist many textbooks on the subject matter that use C, C++, Java, Pascal and many other programming languages, there is no such book that uses the Scheme programming language. This is why we have written this text.

One might ask the question whether the programming language really matters. Can't we use any programming language to program a certain algorithm? This is indeed the case. The algorithms presented in this text can be programmed in any programming language and this is an important raison d'être for this course. It provides a number of insights about algorithms and data structures that any computer programmer needs, irrespective of the programming language he or she uses. Nevertheless, every programming language has its own technical peculiarities and its own particular style of writing programs. This causes algorithms to be written in slightly different ways when using different programming languages. Hence, the same algorithm written in Java or Pascal will probably look slightly different when using Scheme simply because Scheme offers programmers different ways of writing programs; ways which not always know their equivalent in Java or Pascal. Although an algorithm written in one programming language can always be translated to another programming language, the precise nuances will not be identical. The difference is comparable to that of a poem originally written in Dutch not being exactly the same when translated to English.

We start this chapter by introducing some important terminology that will be used throughout the course. In sections 1.1 and 1.2 we define what we mean by terms such as data, data structure, data type, algorithm and procedure type. As we will see, algorithms and data structures soon get very complex. Sections 1.3 and 1.4 discuss ways to tackle this complexity. An important aspect of this course is that we will be studying data structures and algorithms in a scientific way. E.g., when comparing two different algorithms that solve the same problem, we want to know which algorithm performs best. In section 1.5, we present a scientific technique that can be used to measure the quality of algorithms. We end this chapter by explaining how to apply this technique to measure the quality of Scheme programs.

1.1 Data and Data Constructors

A program can be roughly described as a recipe that prescribes a computer to perform a number of computational steps given some input data. That input data can be as simple as a number in the case of a program that computes factorial numbers. The input data can be as complex as an entire book (consisting of chapters, figures, tables, ...) in the case of a word processor. Therefore, every programming language has ways to describe simple data already existing in the

programming language (such as numbers) and ways to construct new data from the already existing data. The simple data that comes with the programming language itself is usually called *primitive data*. It is built into the programming language and it is characterised by the fact that it cannot be decomposed into smaller units of data that make sense in the programming language. In Scheme, examples of primitive data are built-in numbers, characters and so on. A fairly complete list of Scheme's different kinds of primitive data is shown in figure 1.1. The left hand side of the table shows a name for the kind of data we are considering. The right hand side shows some examples of data of the corresponding kind. We call them *data values* or *data elements*. A complete list of Scheme's different kinds of primitive data values can be found in R6RS (i.e. the official description of standard Scheme).

Name	Example Data Values
number	3, 0+4i, 3.1415, 22/7
boolean	#t, #f
symbol	'apple
character	#\a, #\newline, #\space
procedure	#<primitive:sin>

Figure 1.1: Some of Scheme's primitive data types

Apart from primitive data values, every programming language features ways for constructing new *compound data elements* given a series of 'already existing' data elements. With 'already existing' data elements, we mean both primitive data elements as well as previously constructed compound data elements. Compound data elements are also called *data structures*. The defining characteristic of compound data elements is that they can be decomposed into (simpler) constituent data elements that make sense in the same programming language. As already said, the data elements constituting a compound data elements can be either primitive data values or compound data values that were constructed in their turn. Scheme contains three sorts of compound data values. They are summarized in figure 1.2. Again, the left hand side of the table shows a name for the kind of data elements we are considering. The second column gives us a sample data value of the corresponding kind. The third column is explained below.

Name	Example Data Values	Data Constructor
pair	(1 . 2)	`cons`
string	"Hello World"	`make-string`
vector	#3(1 2 3)	`make-vector`

Figure 1.2: Scheme's compound data types

Constructing a new compound data value is accomplished by applying a Scheme procedure that is known as a *data constructor*. As illustrated in figure 1.2, Scheme features `cons`, `make-string` and `make-vector` as its three main data constructors. Data constructors are procedures that actually create a new data structure: they reserve the necessary amount of computer memory to store the data elements that make up the compound data value and they make sure the compound data value is properly initialized. Initialisation is necessary in order to make sure that the data structure is made of meaningful data elements. For instance, merely reserving the memory needed to store a dotted pair without properly initializing its contents would result in gibberish when applying `car` and `cdr` to the newly constructed pair. This is because the part of the computer memory that is used to store the new pair does not necessarily contain meaningful data. It might contain old "garbage" resulting from pairs that were previously stored in the same part of memory. In order to avoid this, `cons` takes exactly two arguments which are used to properly initialize the dotted pair after having reserved the amount of memory necessary to store it. In brief, data constructors initialize a data structure after having reserved the necessary computer memory. Data constructors can take several forms:

Procedural Data Constructors are constructors such as `cons`, `make-vector` and `make-string`. These Scheme procedures have to be called explicitly by the programmer in order to create a data structure in computer memory. `cons` is a procedure that takes two arguments used to initialize the constituents of the newly constructed pair. In the case of `make-vector` and `make-string`, the constructed data structure still needs to be "filled up" afterwards by calling additional procedures defined for this purpose. For example, by running (`make-vector 10`), we explicitly ask the Scheme interpreter to construct a new vector with 10 entries. However, it is up to the programmer to fill up the entries after the data constructor has been executed. In the example of vectors, this is accomplished by calling `vector-set!`. Strings are the topic of chapter 2.

Literal Data Constructors are notations such as #(...) and "...". These notations create new data structures without explicitly calling a Scheme procedure. Just like procedural data constructors, literal data constructors reserve the necessary amount of memory. Moreover, they initialize that memory by using the data values used in the notation. For example, in Scheme, a string can be constructed by "just" writing its contents between double quotes: "Hello `World`". Hence, the double quotes have to be seen as a data constructor that first reserves the computer memory needed to store the string's contents and subsequently fills up that memory with the characters that come with the notation. The string is said to be written down literally. Hence the name 'literal data constructors'. Apart from the double quotes used to construct strings, Scheme features the #(...) notation to create vectors. For example, writing #(1 2 "Hello `World`") creates a vector with three entries, namely, 1, 2 and "Hello `World`".

Having constructed a compound data value (i.e. a data structure) using a constructor, one will typically encounter three kinds of Scheme procedures that perform meaningful computations on that data value:

Accessors are procedures whose purpose it is to externalize the data elements residing in the data structure. Accessors are used to read data elements from the data structure. Accessors are also known as "getters". Typical examples of accessors are `car` and `cdr`: given a dotted pair (constructed by `cons`), then `car` and `cdr` are used to access the data values residing in the dotted pair. Similarly, `vector-ref` is used to access the data values residing in the entries of a vector. Likewise, `string-ref` can be used to access a string's individual characters.

Mutators can be regarded as the conceptual counterpart of accessors. Mutators are procedures that store a data value in a compound data structure. Mutators are also referred to as "setters". In the case of dotted pairs, `set-car!` and `set-cdr!` are the mutators. These procedures are applied to a dotted pair `p` and a data value `v` (e.g. (`set-car! p v`)). They store the data value `v` in the dotted pair `p`. Likewise, `vector-set!` is a mutator for vectors. Some data structures are said to be *immutable* because they lack mutators. E.g., in Scheme, there are no mutators that can be used to modify the characters of a given string data value.

Operations constitute a third category of procedures defined on compound data values. Operations are procedures that operate on the data structure *without* revealing the internal details of that data structure. An example of an operation is (`reverse a-list`) which reverses a given list (i.e. returns the elements of the list in reverse order). `reverse` operates on the list without revealing its constituting elements. Another example is `member?` which checks whether a given element is a member of a given list. Again, the operation operates on the list without giving its user explicit access to the individual data elements that constitute the list.

Finally, the important notion of a *data type* has to be explained. Every data value in Scheme can be classified in a set. E.g., the data value `3` is said to belong to the set of Scheme numbers. The name of that set (in this case `number`) is said to be the data type of that data value. Other examples of data types are `character`, `pair`, `string` and `vector`. Apart from serving as a name for the kind of the data elements it represents, a data type is also crucial in order to know exactly the procedures that are applicable to its data values. E.g., it is clear that the Scheme procedure + is not applicable to the compound data elements produced by `cons`. Indeed, dotted pairs do not belong to the data type `number` and thus cannot be added. Conversely, the procedure `car` is not applicable to the value `3`. This is because `car` is only applicable to data values whose data type is `pair`. To summarize, a data type is *the* indicator for the set of procedures (i.e. accessors, mutators and operations) that are applicable to the data values of that data type. For Scheme's primitive data types, this set of

operations is simply too big to be listed in this text. We refer to the R6RS for a complete list of operations defined on Scheme's primitive data types. Table 1.3 shows a number of compound Scheme data types along with the most common accessors, mutators and operations that are applicable to the data values of the corresponding data type.

Name of the Type	Operations applicable

`pair`	`car`
	`cdr`
	`set-car!`
	`set-cdr!`
	`equal?`
	`eq?`
	`eqv?`
	...

`string`	`string-length`
	`string-ref`
	`string<?`
	...
	`string-ci<?`
	...
	`substring`
	`string-append`
	`string->list`
	`list->string`
	`string-copy`
	`string-fill!`

`vector`	`vector-length`
	`vector-ref`
	`vector-set!`
	`vector->list`
	`list->vector`
	`vector-fill!`

Figure 1.3: Operations applicable to some of Scheme's compound data types

1.2 Algorithms and Algorithmic Constructors

Now that we have developed a precise vocabulary to talk about data, data values, data structures and data types, let us turn our attention to algorithms. Technically spoken, an algorithm is just a Scheme procedure. It is a textually specified recipe for a certain number of computational steps to be executed on a given input. You are probably already familiar with a number of algorithms

from your Scheme course. Examples include procedures to compute the square root of a number, numerical procedures for computing integrals and a procedure to reverse a Scheme list.

The origin of the word *algorithm* is to be found in the name of the 9th century Persian mathematician Abu Abdullah Muhammad ibn Musa al-Khwarizmi ("the one from Khwarizmi"). The word "algorism" originally referred to the rules needed for doing arithmetic in our decimal system (in which numbers are written using ten symbols having the values 0 through 9 and in which each symbol has ten times the weight of the one to its right). The meaning of the word has evolved, via European Latin translation of al-Khwarizmi's name, into "algorithm" in the 18th century. The word evolved to include all definite procedures for solving problems or performing computational tasks.

The two foremost important properties of algorithms is that they are generally applicable procedures that can be executed by following a well-defined set of elementary computational steps:

Generality. An algorithm has to be generally applicable to all possible inputs of a certain data type. If one was to write a Scheme procedure `fac` that is only applicable to odd numbers smaller than 101, then this would not be considered a valid algorithm for computing factorials. The reason is that we *know* that factorials exist for all other positive numbers as well and that we *know* that there are easily executable procedures that lead to those factorial numbers. Hence, the Scheme procedure would not be considered an algorithm for computing factorials because it is not general enough.

Computability. An algorithm should consist of a number of clearly specified computational steps. For example, a procedure that includes instructions such as "ask an oracle to crack the code of my bank card and then print this code on the screen" would not be considered a valid algorithm. The underlying point is that we demand from an algorithm that it can be mapped onto expressions in a programming language such as Scheme. When specifying algorithms without using a real technical programming language (but instead use plain English for instance), there is a potential danger that the result cannot be programmed at all. In order to avoid confusion, this text therefore specifies all algorithms directly in Scheme.

Anyone who already had an introductory course on Scheme will recognise this as a mere Scheme procedure. So what is the difference between the algorithms presented in an introductory course on Scheme and the algorithms presented here? In order to see this difference, it is necessary to have a look at the inputs for algorithms. In a typical introductory course on programming, the algorithms presented usually operate on fairly simple types of input data. In Scheme this usually implies algorithms that operate on numbers (like e.g. computing the square root of a given number with a given precision) or lists (like e.g. reversing a list or appending two lists). Seen from the perspective of this text, these are extremely simple kinds of input. As a result, the algorithms

that operate on them are fairly simple as well. Things get more interesting if we consider algorithms that operate on richer data structures. This is exactly what this course is about. In general one might say that the more complex the data structures get, the more complex (and thus the more interesting) the algorithms get that operate on those data structures.

Let us now develop some vocabulary to talk about algorithms in the same spirit of the vocabulary for data structures which we presented in the previous section. Similar to terms like data value, data type, primitive data, compound data and data constructor, we introduce the terms algorithm, algorithmic type, primitive algorithms, compound algorithms and algorithmic constructor.

We will call any procedural Scheme element that cannot be divided into more atomic Scheme building blocks a *primitive algorithm*. Examples of primitive algorithms include Scheme's built-in procedures such as +, `sin` and `display`. Even some parenthesis can be considered as primitive Scheme algorithms: those parenthesis that are used to denote a procedure call cannot be decomposed into more primitive computations. Therefore, calling a procedure is one of Scheme's most fundamental building blocks for writing programs. It is a mechanism that is so inherent to Scheme that we consider it as a primitive algorithmic construction. For example, the Scheme expression (`sin` 3.14) consists of two primitive algorithms: one — the parentheses — to call a procedure and another one — the `sin` operator — that corresponds to the built-in sine procedure. However, not all parentheses denote primitive algorithms. To see this, we define a *compound algorithm* as any Scheme algorithm that was created by applying a number of *algorithm constructors* to primitive algorithms or previously built compound algorithms. The algorithm constructors are those Scheme special forms that form more complex algorithms from simple algorithms. Examples of algorithmic constructors are `do`, `let`, `let*`, `letrec`, `lambda` and `if`. These special forms are used to build compound algorithms given some primitive algorithms or a set of previously built compound algorithms. For example, consider the following factorial function implemented in Scheme.

```
(define fac
  (lambda (n)
    (if (= n 0)
      1
      (* n (fac (- n 1))))))
```

Based on the primitive algorithms +, *, −, = and Scheme's procedure calling mechanism, the `if` and `lambda` algorithmic constructors are used to build a compound algorithm that is clever enough to compute factorials.

Notice that we do not consider `define` as an algorithmic constructor because `define` doesn't really create a new compound algorithm. Instead, `define` allows us to give a *name* to (compound) algorithms. In the aforementioned factorial example, it is `lambda` that creates the compound factorial algorithm. `define` is merely used to associate the name `fac` with the algorithm. `define` therefore allows us to abstract away from the details of a compound algorithm by referring to its name instead of referring to its constituting details. This is known as

procedural abstraction.

1.3 Procedural Abstraction and Procedure Types

Given the data constructors and algorithmic constructors discussed above, programmers can create an infinite number of data structures and algorithms. However, it soon appears that the intellectual effort to understand the complexity of large data structures and big algorithms gets enormous. This is very problematic since understanding algorithms and data structures (written by others) is one of the most important activities performed by programmers. Programmers need to understand algorithms and data structures whenever they try to use other people's work for example when they have to maintain a software system that was written by people that are no longer around (because they have moved to a different company e.g.).

In order to reduce the complexity arising from huge amounts of applications of data constructors and algorithmic constructors, computer scientist have developed abstraction techniques. Abstraction is one of the key means to tackle complexity in any science and this is no different in computer science. The major way to realize abstraction in computer science is by giving a name to complex things. In the case of complex data structures, this is called *data abstraction*. In the case of algorithms it is called *procedural abstraction*.

Procedural abstraction consist of giving a name to a Scheme procedure by means of `define`. This has two important benefits. First, by giving a meaningful name to a compound algorithm, someone reading the algorithm will understand much easier what the algorithm is about. Indeed, the classical version of our good old factorial algorithm is much easier understood when the name `fac` is used instead of just any other name (like, say `qfxttq`). In other words, procedural abstraction — when correctly used — can enhance the readability of programs enormously. Second, by giving a name to a compound algorithm, it becomes possible to refer to the compound algorithm simply by mentioning its name in a procedure call. This means that we do not need to copy the algorithm whenever we need it. We simply call it by using its name. In other words, procedural abstraction avoids the duplication of code. This is an important factor in computer science since duplication of code also means duplication of errors and duplication of maintenance efforts.

We conclude that programs that heavily rely on procedural abstraction are easier to read and easier to maintain. Of course, it is crucial to use good names in doing so. Using such names as `temp` or `my-procedure` soon results in programs that are totally unreadable.

Now that we know what primitive algorithms, compound algorithms, algorithm constructors and procedural (i.e. algorithmic) abstraction are, we can turn our attention to the notion of an *algorithmic type*, also called *procedural type* or simply *type of a procedure*. The type of a procedure is a description of the data types of the data values to be used as the input of the procedure, along with the data type of the output values that are produced by the procedure.

For example, for the primitive procedure `sin` we will say that its procedure type is (`number` → `number`). We use this expression to express that the type of `sin` is "from number to number". Similarly, we can say that the type of `reverse`, the compound Scheme procedure for reversing lists, is (`pair` → `pair`). The procedure takes a dotted pair (representing a list) and produces a dotted pair (representing the reversed version of the input list). We say `reverse` is "from dotted pair to dotted pair". Whenever a procedure has more than one argument, we simply enumerate all their data types on the left hand side of the arrow. For example, the type of `append` (which consumes two lists and returns the list representing the second list appended to the first one) is (`pair pair` → `pair`).

Finally, let us have a look at the procedural type of a higher procedure. Consider for example the procedure (`zero f a b epsilon`) which computes a number `v` between `a` and `b` such that (`f v`) equals zero with a precision `epsilon`. Its implementation is as follows:

```
(define (zero f a b epsilon)
  (define c (/ (+ a b) 2))
  (cond ((< (abs (f c)) epsilon) c)
        ((< (* (f a) (f c)) 0) (zero f a c epsilon))
        (else (zero f c b epsilon))))
```

Clearly, this procedure takes four arguments, the three latter of which are of type `number`. But what is the type of `f`? The algorithm assumes that `f` is a procedure that operates on numbers. Hence, the data type of `f` is (`number` → `number`). Therefore, the type for `zero` is:

((`number` → `number`) `number number number` → `number`)

This procedural type expression formally shows us that `zero` takes four arguments: one function of type (`number` → `number`) and three other arguments of type `number`. The result of the procedure is a `number`.

It is important to notice that procedural type expressions like (`number` → `number`) are *not* a part of the Scheme programming language. Instead, they are a convention that we use to specify the types of the arguments and the results of Scheme procedures.

1.4 Data Abstraction: Abstract Data Types (ADTs)

Just like procedural abstraction allows us to give a name to a complex compound algorithm, *data abstraction* allows us to group a number of data elements into one single compound data element by giving it a meaningful name. As an example, suppose we are to write a mathematical software system that helps engineers perform complicated calculations using complex numbers. As Scheme programmers, we might be tempted to think of complex numbers as dotted pairs in which the car is used to store the real part of a complex number and in which the cdr is used to store its imaginary part. In this setting, the algorithms that

operate on complex numbers refer to (`car c`) whenever they need the real part of some number `c`. Similarly, (`cdr c`) would be used to refer to `c`'s imaginary part. Although this solution is simple, it has two important drawbacks that have huge repercussions on the maintainability of software systems.

- First, we observe the same problems w.r.t. readability of programs as the ones described in the section on procedural abstraction. Not using meaningful names to denote data structures and the elements they contain means that all data manipulation has to be done directly in terms of Scheme's accessors like `car`, `cdr` and `vector-ref`. For complex data structures this can result in extremely complex expressions. E.g., just from reading a program, one would probably never guess that an expression like (`vector-ref` (`caddr` (`vector-ref invoices` 10)) 12) might designate the 12th most important company that sells the product that was sold by the company listed on the 10th invoice stored by our system. Instead of thinking in terms of "a vector of lists of vectors", it is much more comprehensible to think in terms of "a collection of companies that sell products whose product name on the invoice contained in the collection of invoices". Hence, it pays off to give data structures a meaningful name.

- Apart from the readability of the procedures that operate on a data structure, adaptability is also an important issue when it comes to designing data structures. E.g. in the mathematical software system described above, one might decide to change the representation of complex numbers from dotted pairs to vectors with two entries. Without abstraction, this would require one to manually replace all occurrences of `car`'s and `cdr`'s that relate to the manipulation of complex numbers by equivalent calls to `vector-ref`. Moreover, one cannot *blindly* replace all occurrences of `car` and `cdr` in the code since there are probably many occurrences of `car` that have nothing to do with complex numbers. Obviously, those occurrences do not have to be replaced. Needless to say, manually looking for those occurrences that do need to be replaced is a very error-prone process since the chances that one misses one such occurrence are quite high.

To alleviate these drawbacks, computer scientists resort to data abstraction by using a device called *Abstract Data Types* or *ADT*s for short. An abstract data type is a data type in the sense of the definition presented in section 1.1. In other words, it is a name for a set of data values. The idea of a data type being abstract means that the name is used to draw an abstraction border between the *users* of the ADT and the *implementor* of the ADT. As a consequence, the implementor of the ADT can choose whichever *implementation of the ADT* that he wants to provide. For example, given the abstract data type `complex`, a programmer might want to use dotted pairs to represent data values of type `complex`. In a later version of the system, he might want to choose vectors (of size two) to represent the data values of type `complex`. However, an ADT is more than just a new name for a compound data type. Remember from section 1.1 that a data type is inherently associated with the name and the procedural

types of the procedures that operate on the data values of that data type. This is also the case for *abstract* data types. The procedural types of these applicable procedures form an integral part of the definition of the ADT. We therefore define:

An **abstract data type** *is a name for a data type along with a set of procedural types that prescribe the kind of procedures (i.e. constructors, accessors, mutators and operations) that can be applied to the data elements of the data type.*

As an example, the following shows the ADT `complex`. Firstly, the ADT specifies a name for a set of data values: `complex`. It is the data type for those data values. Secondly, the ADT also lists the procedural types for the constructor, the accessors (this ADT does not have mutators) and the operations that are to operate on the data values of type `complex`. `new` is the name of the constructor. Its procedural type teaches us that it takes two Scheme numbers and that it returns a new complex number that has the two given numbers as real and imaginary parts. `real` and `imag` are the accessors that can be used to access these parts. `complex?` is an operation that can be applied to any scheme value. It returns #`t` if that value is a complex number and #`f` in all other cases. The other procedure types specify the operations that can be performed on complex numbers. Notice that we use the name `any` to indicate the data type consisting of all possible Scheme values.

```
1  ADT complex
2
3          new        ( number number → complex )
4          complex?   ( any → boolean )
5          +          ( complex complex → complex )
6          −          ( complex complex → complex )
7          /          ( complex complex → complex )
8          *          ( complex complex → complex )
9          modulus    ( complex → number )
10         argument   ( complex → number )
11         real       ( complex → number )
12         imag       ( complex → number )
```

Given a particular complex number data value `c` that was constructed using `new`, the fact that its data type is `complex` really is the only thing that users can assume about `c`. The concrete implementation of the procedures and the concrete representation of `c` is hidden from the user of the ADT. Users of the ADT are only allowed to manipulate complex numbers using these abstract descriptions. This is the essence of data abstraction.

A programmer that wishes to *implement* the ADT must decide on a concrete *representation* for the data values of type `complex`. He could choose to represent those values as dotted pairs containing the real and imaginary parts. However, he could also choose to represent the values as dotted pairs containing their argument and modulus. After all, given the real and the imaginary part, one can

always compute the modulus and the argument of a complex number, and vice versa. Many different representations are possible. E.g., in a system where speed is of utmost importance, he could avoid the computations converting between real and imaginary parts, and, argument and modulus. To achieve this, he could represent complex numbers as vectors of size four in order to explicitly store the real part, the imaginary part, the modulus and the argument. A virtually infinite number of representations for the ADT's data values are imaginable. For each representation, a different implementation will need to be provided for all the procedural types listed in the ADT's definition. According to the principle of data abstraction, users of the ADT cannot explicitly rely on this representation but use the ADT's procedures instead. This makes their code much more readable as that code is clearly about complex numbers instead of dotted pairs or vectors. It also makes the user code easier to adapt since it is independent of the implementation of the ADT.

Before we move our attention to the definition of some other useful ADTs, let us first have a look at how to implement ADTs in Scheme.

1.4.1 Implementing Data Abstraction in Scheme

Scheme has two radically different programming styles that can be used to implement ADTs. They are known as *the procedural style* and *the object-based style*. We present them one after the other.

Procedural Style

The simplest way to implement ADTs in Scheme is to write Scheme procedures for the constructors, accessors, mutators and operations as specified by the ADT definition. Let us have a look at an implementation for the `complex` ADT presented in the previous section. The implementation is said to follow the procedural style because all the operations are implemented using plain Scheme procedures that take the data elements of the ADT as explicit arguments.

The procedural implementation shown below uses R6RS Scheme's library system. A library can be thought of as a collection of Scheme names, some of which are exported towards users of the library. Every library has a name (`complex` in our case), has an `export` clause listing the names it exports and has an `import` clause that specifies which other libraries it relies on. The import clauses allows the library to access the names that are exported by these libraries in its turn. For instance, by stating (`import` (`rnrs io simple`)), a library can access Scheme's standard input/output-facilities such as `display` and `newline`. A simple import clause such as (`rnrs io simple`) imports *all* the names exported by the corresponding libraries. Sometimes, this can be problematic when names imported from two different libraries are identical. In order to resolve such situations, we can chose not to import certain names using the `except` clause. By importing (`except` (`rnrs base` (`6`) `complex?`) we state that we are importing the library (`rnrs base` (`6`)) except for the procedure `complex?`. In our case, this is because the procedure `complex?` that we will program as part of our `complex`

ADT has the same name as the standard procedure `complex?`. We therefore decide not to import the standard procedure. Sometimes we need to resolve a name clash by renaming a number of procedures imported from a library. This is accomplished using the `rename` import clause. E.g., by using the clause (`rename` (`rnrs base` (6)) (+ `number+`) (∗ `number∗`)) we state that we are importing the standard library (`rnrs base` (6)) but that we desire to refer to the standard procedures + and ∗ by using the names `number+` and `number∗`. This allows us to use the names + and ∗ without ending up with name clashes.

The following code shows the procedural implementation of the `complex` ADT. It shows a library that provides an implementation for all the procedures listed in the `ADT` definition. This implementation represents complex data values as lists of length three: a tag used to identify complex numbers, and two numbers that correspond to the real and the imaginary part of a complex number.

```
(library
 (complex)
 (export new complex? real imag + - / * modulus argument)
 (import (rename (except (rnrs base (6))
                         complex?)
                 (+ number+) (* number*) (/ number/) (- number-)))

 (define complex-tag 'complex)
 (define (get-real c)
   (cadr c))
 (define (get-imag c)
   (caddr c))

 (define (new r i)
   (list complex-tag r i))

 (define (complex? any)
   (and (pair? any)
        (eq? (car any) complex-tag)))

 (define (real c)
   (get-real c))

 (define (imag c)
   (get-imag c))

 (define (+ c1 c2)
   (define real (number+ (get-real c1) (get-real c2)))
   (define imag (number+ (get-imag c1) (get-imag c2)))
   (new real imag))

 (define (* c1 c2)
   (define real (number- (number* (get-real c1) (get-real c2))
```

```
                        (number* (get-imag c1) (get-imag c2))))
  (define imag (number+ (number* (get-real c1) (get-imag c2))
                        (number* (get-imag c1) (get-real c2))))
  (new real imag))

(define (- c1 c2)
  (define real (number- (get-real c1) (get-real c2)))
  (define imag (number- (get-imag c1) (get-imag c2)))
  (new real imag))

(define (/ c1 c2)
  (define denom (number+ (number* (get-real c2)
                                  (get-real c2))
                         (number* (get-imag c2)
                                  (get-imag c2))))
  (define real (number+ (number* (get-real c1)
                                 (get-real c2))
                        (number* (get-imag c1)
                                 (get-imag c2))))
  (define imag (number- (number* (get-imag c1)
                                 (get-real c2))
                        (number* (get-real c1)
                                 (get-imag c2))))
  (new (number/ real denom) (number/ imag denom)))

(define (modulus c)
  (sqrt (number+ (number* (get-real c) (get-real c))
                 (number* (get-imag c) (get-imag c)))))

(define (argument c)
  (atan (get-imag c) (get-real c))))
```

Some of the Scheme procedures (`get-real` and `get-imag` to be precise) of the above library are merely needed to implement the operations of the ADT in a more convenient way; they are not part of the definition of the `complex` ADT itself. These procedures are said to be *private to the ADT implementation*. Technically, this is simply achieved by not listing these procedures in the `export` clause of the library. As a consequence, programs (or other libraries) importing the library will not be able to access those names. In our example, `get-real` and `get-imag` are not part of the `export` clause of our `complex` library.

The following Scheme program illustrates how to use an ADT that was implemented in the procedural style. The program imports three libraries: the standard library (`rnrs base` (6), the `complex` library[1] and the standard in-

[1]The number (1) designates the version of in the library in case different versions exist. In our case, this is indeed the case. The first version is the library just discussed. The second version is the object-styled implementation studied in the following section.

put/output library (`rnrs io simple`). Notice that importing our complex library and the standard library result in name clashes. E.g., both provide procedures named +, ∗ and so forth. We therefore decided to refer to the procedures of our `complex` ADT using locally different names: the (`prefix` (`a-d examples complex` (1)) `complex:`) states that we desire to import the procedures of the (`a-d examples complex` (1)) library by first prefixing all its exported names with `complex:`. Like this, we can refer to the standard addition procedure for numbers (using the name +) as well as to our complex addition procedure (using the name `complex:+`).

```
(import (prefix (a-d examples complex (1)) complex:)
        (rnrs base (6))
        (rnrs io simple))

(define cpx1 (complex:new 1 4))
(define cpx2 (complex:new 5 3))
(display (complex:+ cpx1 cpx2)) (newline)
(display (complex:∗ cpx1 cpx2)) (newline)
(display (complex:+ cpx1 cpx2)) (newline)
(display (complex:∗ cpx1 cpx2)) (newline)
(display (complex:real cpx1))   (newline)
(display (complex:imag cpx2))   (newline)
(display (complex:modulus cpx1)) (newline)
(display (complex:argument cpx2))
```

Object-based Style: Encapsulation

One problem with the procedural style of implementing ADTs is that the abstraction barrier offered by the ADT is a mere convention that can easily be circumvented. Consider a complex number `c` that was constructed using the aforementioned ADT implementation. It suffices to evaluate the expression (`cadr c`) to bypass the abstraction barrier of the ADT. In other words, the procedural style of programming ADTs is entirely based on conventions. It does not provide us with a way to *enforce* the abstraction barrier. This is exactly what the object-based style of of implementing ADTs tries to achieve. To accomplish this, it uses a powerful technique to *encapsulate* the representation details of the ADT's data values inside a Scheme procedure, called a *dispatcher*.

As an example, we present an implementation of the `complex` ADT using the object-based style. The precise technical details of this particular implementation are not important right now: the implementation uses a number of advanced Scheme features such as variable size argument lists (notice the . in the dispatcher) which are not always widely known among beginning Scheme programmers. The focus of the discussion is on the way the ADT is implemented: the constructor `make-complex` returns a dispatching procedure that represents a data value of the ADT.

```
(define (make-complex r i)
```

```
(define (complex+ c)
  (make-complex (+ r (c 'real))
                (+ i (c 'imag))))
(define (complex* c)
  (make-complex (- (* r (c 'real))
                   (* i (c 'imag)))
                (+ (* r (c 'imag))
                   (* i (c 'real)))))
(define (complex- c)
  (make-complex (- r (c 'real))
                (- i (c 'imag))))
(define (complex/ c)
  (define denom (+ (* (c 'real)
                      (c 'real))
                   (* (c 'imag)
                      (c 'imag))))
  (define real (+ (* r (c 'real)) (* i (c 'imag))))
  (define imag (- (* i (c 'real)) (* r (c 'imag))))
  (make-complex (/ real denom) (/ imag denom)))
(define (modulus)
  (sqrt (+ (* r r) (* i i))))
(define (argument)
  (atan i r))
(define (real)
  r)
(define (imag)
  i)
(lambda (message . args)
  (cond ((eq? message '+) (apply complex+ args))
        ((eq? message '-) (apply complex- args))
        ((eq? message '*) (apply complex* args))
        ((eq? message '/) (apply complex/ args))
        ((eq? message 'modulus) (modulus))
        ((eq? message 'argument) (argument))
        ((eq? message 'real) (real))
        ((eq? message 'imag) (imag))
        ((eq? message 'complex->list) (list 'complex r i))
        (else (error "Complex Number Message Not Understood")))))
```

The aspects of the ADT's implementation which are important to understand is that there is only one single procedure (namely `make-complex`) installed in the global environment. That procedures "captures" (technically we say *encapsulates*) a number of local procedures (such as `real`) which are stored in the local environment of the anonymous dispatcher that is returned from the constructor. The returned dispatcher is referred to as an *object* in this style of programming since it can be considered as an indivisible thing. It has to be

called with quoted symbols (such as '+). These are said to be *messages* that are *sent* to this object. This is the reason why this style of implementing ADTs is called the object-based style. Notice that the technique of using dispatchers automatically makes private procedures inaccessible to users of the ADT. They are said to be encapsulated along with the data representing the data values of the ADT.

This implementation style has a number of advantages over the procedural implementation:

Encapsulation By encapsulating the representation details of the ADT in a dispatcher, we are *guaranteed* not to externalise these details. As a consequence, user programs will never be able to rely on these details which makes it much easier to adapt the representation of the ADT without affecting the validity of the user program. In other words, encapsulation facilitates the maintenance of both the implementation of the ADT as well as the maintenance of the code that uses the ADT.

Name Space Management Only the constructor for the ADT's data values is installed in the global Scheme environment. All procedures private to the implementation of the ADT are stored inside the dispatcher's local environment of definition. As a result it gets easier to manage one's name space in systems with lots of ADTs. In the procedural implementation style, each and every procedure of each and every ADT is included in the global environment. This makes it much harder to avoid name clashes between different procedures that belong to different ADT.

Code Size The code for the operations in the object-based style is often smaller than the code in the equivalent procedural style implementation. Because of Scheme's lexical scoping rules, one typically needs less accessors. In the `complex` example we use `r` and `i` whereas the procedural style implementation has to call the accessor procedures `get-real` and `get-imag` for obtaining these values.

However, the style also has a number of disadvantages:

Encapsulation Breached The object-based style only lives up half to its expectations. E.g., in the implementation of binary operations (i.e. operations that need two arguments) one needs two objects: the object receiving the message and the argument object. As the implementation for complex number addition shows, it is necessary to bypass the encapsulation of the argument object in order to get access to its implementation details. In the case of the addition, it is necessary to access the real and the imaginary part of the argument in order to be able to calculate the sum of the complex numbers. In our `complex` example this is not a problem since `real` and `imag` are a part of the ADT specification. However, suppose that the designer of the `complex` ADT would decide to remove these two accessors from the ADT. Our implementation for the addition would still

need them. In other words, the object-based style sometimes requires one to add accessors just to be able to implement binary operations, even when those accessors are not part of the ADT definition. Hence, encapsulation is not guaranteed.

Code Complexity Object-based code is often a bit more complex than the equivalent procedural code. As we can see from the implementation, we need more complex scoping. Furthermore, advanced features such as variable length arguments (notice the dot in the dispatcher) are needed. The code is also a bit slower because of the conditional in the dispatcher.

Space Inefficiency The object-based style has a huge problem when it comes to space efficiency. Looking back at the object-based implementation of our `complex` ADT. *Every* time we call the constructor procedure `new`, local procedures such as `complex+` are created that implement the operations of the ADT. In our case, there are seven such local procedures. This means that in a system with one thousand complex numbers, we have seven thousand procedures stored in our computer memory, only seven of which are distinct. Solving this problem requires us to factor out these procedures from the constructor. But then we gradually move back to an implementation with global procedures as is the case with the procedural style.

Although good solutions exist to alleviate these drawbacks, they render the resulting Scheme code more complex. These solutions deserve attention in advanced courses on object-oriented programming. They fall beyond the scope of this text. Therefore, we have opted for the procedural ADT implementation style in this text.

1.4.2 Genericity

From chapter 3 on, we will be *storing* data elements in data structures. A data structure whose main purpose it is to "store" (i.e. "remember") data values for later retrieval is called a *storage data structure* and the data values which it stores are called *records*. Typical examples of storage data structures include a phone index and a library catalogue. These can be considered as data structures whose purpose it is to store personal data and books for later retrieval. This is in sharp contrast with the `complex` ADT, the implementation of which (see section 1.4.1) is technically spoken also a data structure (since it combines two Scheme numbers into a compound data value that represents the complex number). However, we can hardly claim that the raison d'être of complex numbers is to *store* data elements for later retrieval. That is why we do not say that the real and the imaginary values sitting in a complex number are records.

The following example shows the `max-o-mem` ADT. A max-o-mem is a storage data structure that can remember exactly one record. The idea is that user programmers can keep on writing values to the max-o-mem, but that the max-o-mem only remembers the "greatest" value it was ever given. The ADT itself

is very simple. It has a constructor `new`, a procedure `max-o-mem?` that can be used to verify whether or not a given scheme value is a max-o-mem, a procedure `write!` to write a value to the max-o-mem (which will be ignored in case the max-o-mem already contains a value that was greater) and a procedure `read` to read the max-o-mem's "current greatest" value.

```
1  ADT max-o-mem< T >
2
3          new        ( ( T T → boolean ) T → max-o-mem< T > )
4          max–o–mem? ( any → boolean )
5          write!     ( max-o-mem< T > T → max-o-mem< T > )
6          read       ( max-o-mem< T > → T )
```

What can we say about the procedural type of a procedure such as `write!`? Clearly, it requires a max-o-mem and an data element that has to be written to that max-o-mem. It returns the (potentially) modified max-o-mem. On first sight, this results in a procedural type for `write!` that is given by (`max-o-mem any` → `max-o-mem`). But is it really possible to write any scheme value to a max-o-mem? The answer is no. In order to understand this, try to imagine an implementation of `write!`. At some point it will have to compare the procedure's argument with the value that is currently stored in the max-o-mem. In order to do so, it might use the Scheme procedure < ("lesser than"). However, this implies that the max-o-mem can only store Scheme numbers. Suppose that we want to use a max-o-mem to store complex numbers as defined by the ADT presented in the previous section. How can we tell the max-o-mem ADT implementation not to use Scheme's < but to use our own special procedure `complex.<` instead? This is achieved by *parametrizing* the `max-o-mem` ADT by the data type `T` of the data elements it will contain. This parametrisation of the ADT is denoted using angular brackets < and > in the name of the ADT. Hence `max-o-mem<T>` refers to "a max-o-mem that can store data elements of data type `T`". If we use a max-o-mem to store numbers, we say the max-o-mem has the data type `max-o-mem<number>` (i.e. selecting `T = number`). If we use a max-o-mem to store dotted pairs, we refer to that max-o-mem as having data type `max-o-mem<pair>` (i.e. we select `T=pair`). Using this knowledge, we can explain the procedural type of `read` in the ADT. `read` takes any max-o-mem that stores elements of type `T`, i.e. it takes a parameter of type `max-o-mem<T>`. It returns the greatest value currently stored by that max-o-mem, i.e. a value of type `T`. Hence, `read`'s procedural type is (`max-o-mem<T>` → `T`). Similarly, `write!` takes a `max-o-mem<T>` and a value of type `T`. It returns the modified max-o-mem of type `max-o-mem<T>`. Hence its procedural type is (`max-o-mem<T> T` → `max-o-mem<T>`).

How can we provide an implementation in Scheme for such an ADT that is parametrized with a data type `T`? I.e., how can we implement a data structure that does not depend on the data type of the records it stores? In the case of the `max-o-mem` ADT, all we have to do is to make sure that the code of the implementation does not contain a hardwired reference to Scheme's <

procedure. This can be easily done by providing the correct version of "lesser than" when constructing a max-o-mem. Max-o-mems that are required to store Scheme numbers will be provided Scheme's <. Max-o-mems that are required to store complex numbers can be provided our own special procedure `complex<`. Hence, the constructor `new` has to be a higher order procedure that takes a "lesser than" procedure as its first argument. Any such "lesser than" procedure is to decide whichever is the smallest value, given two values of data type `T`, i.e. it has procedural type (`T T` → `boolean`). `new`'s second argument is the initial value stored in the max-o-mem. It is therefore of data type `T`. This explains the procedural type of `new`. It takes a "lesser than" procedure and an initial smallest element. It returns a max-o-mem of type `max-o-mem<T>`. Hence it has procedural type ((`T T` → `boolean`) `T` → `max-o-mem<T>`).

ADTs representing storage data structures that are independent of the data type of the records they are supposed to store are known as *generic ADTs*. At the implementation level, generic ADTs are implemented by *generic data structures*. Generic data structures are data structures whose implementation is independent of the data type of the records they store. In Scheme, generic data structures are realized by turning their constructor into a higher order function that take all the procedures that *do* depend on the data type of the records stored (such as the "lesser than" in our example). Using the procedural style, the implementation for the `max-o-mem` ADT now looks as follows:

```
(library
 (max-o-mem)
 (export new max-o-mem? read write!)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define max-o-mem-tag 'max-o-mem)

 (define (new << init)
   (list max-o-mem-tag << init))

 (define (max-o-mem? any)
   (and (pair? any)
        (eq? (car any) max-o-mem-tag)))

 (define (read mom)
   (caddr mom))

 (define (write! mom new-value)
   (define << (cadr mom))
   (define value (caddr mom))
   (if (<< value new-value)
     (set-car! (cddr mom) new-value))
   mom))
```

Let us have a look at the implementation for the constructor `new`. The idea is to represent a max-o-mem as a tiny list with three elements: the type tag 'max-o-mem, the "lesser than" procedure << and the max-o-mem's initial value `init`. In the implementation of `write!` we observe how the max-o-mem's *own* "lesser than" operator is selected using (`cadr mom`) and subsequently used for the comparison. We can make max-o-mems of numbers by calling the constructor (`max-o-mem.new` < `v`) where `v` is some initial value for the max-o-mem. If we would like to make a max-o-mem storing complex numbers in which we consider $a + b.i < c + d.i$ whenever $\sqrt{a^2 + b^2} < \sqrt{c^2 + d^2}$, then we proceed as follows:

```
(import (rnrs base (6))
        (prefix (a-d examples max-o-mem) mom:)
        (prefix (a-d examples complex (2)) complex:))


(define (complex< c1 c2)
  (define (square x) (* x x))
  (< (sqrt (+ (square (complex:real c1))
              (square (complex:imag c1))))
     (sqrt (+ (square (complex:real c2))
              (square (complex:imag c2))))))

(define complex-mom (mom:new complex< (complex:new 0 0)))
```

The following code excerpt shows how to use these abstractions to write a procedure `greatest` that computes the greatest element of a list of data elements by iterating over the elements of the list and by sequentially storing all those elements in a max-o-mem. At the end of the procedure, the max-o-mem contains the greatest element. It is read and returned as the result of the procedure.

```
(import (rnrs base (6))
        (rnrs mutable-pairs (6))
        (prefix (a-d examples max-o-mem) mom:)
        (prefix (a-d examples complex (2)) complex:))


(define (greatest lst << init)
  (define max (mom:new << init))
  (define (iter lst)
    (mom:write! max (car lst))
    (if (not (null? (cdr lst)))
      (iter (cdr lst))))
  (iter lst)
  (mom:read max))

(define integer-list (list 1 2 3 4 5))
(define complex-list (list (complex:new 1 0) (complex:new 0 1)
                           (complex:new 3 4) (complex:new 4 3)))
```

The point of all this is that we have turned `max-o-mem` into an ADT the constructor of which requires us to pass along an abstract "lesser than" operator. This extra parametrisation turns the `max-o-mem<T>` ADT into a generic ADT since its specification has gotten independent from the data type `T` of the records that it stores. The max-o-mem is therefore much more reusable than would be the case if we would have used some specific built-in "lesser than" operator from Scheme.

1.4.3 The Dictionary ADT

One of the central ADTs studied in this text is the `dictionary` ADT. The `dictionary` ADT is an abstraction that is used in many computer applications. The most trivial example is an actual dictionary program, e.g. a dictionary Dutch-English. The characterising property of dictionaries is that they store so called key-value pairs. In other words, they associate *keys* with *values*. Therefore, dictionaries are also called *associative memories*. In the Dutch-English example, keys are Dutch words. With every Dutch word, a list of translations is associated. This list is the value that is associated with the key. However, the `dictionary` ADT is an abstraction that has many applications that go beyond translating dictionaries. E.g., a phone index associates an address and a phone number with any combination of a first name and a family name. It is said that a phone index is a dictionary that has the names as keys and that has the address and the phone number pairs as values. Similarly, a library catalogue might be considered as a `dictionary` in which names of authors form the keys and in which a book title along with an ISBN number form the value. In all three examples, a `dictionary` is a data structure that associates keys (i.e. Scheme values of a certain data type `K`) with values (i.e. Scheme values of a certain data type `V`). E.g., the Dutch-English dictionary can be thought of as a data structure having data type `dictionary<string list-of-strings>` (i.e. `K` = `string` and `V` = `list-of-strings`).

The `dictionary` abstraction barrier is formally defined by the following ADT specification:

```
1  ADT dictionary < K V >
2
3  new
4      ( ( K K → boolean ) → dictionary < K V > )
5  dictionary?
6      ( any → boolean )
7  insert!
8      ( dictionary < K V > K V → dictionary < K V > )
9  delete!
10     ( dictionary < K V > K → dictionary < K V > )
11 find
12     ( dictionary < K V > K → V ⋃ {#f} )
13 empty?
```

```
14    ( dictionary<K V> → boolean )
15  full?
16    ( dictionary<K V> → boolean )
```

In this ADT definition, `empty?` and `full?` are predicates that can be used by user programs in order to check whether or not a dictionary is empty or full. They both take a dictionary of data type `dictionary<K V>` and return a `boolean`. The most interesting operations are `insert!`, `delete!` and `find`. `insert!` takes a dictionary of type `dictionary<K V>`, a key or type `K` and an associated value of type `V`. It adds the key-value pair to the dictionary and returns the destructively modified dictionary; i.e. a value of type `dictionary<K V>`. `delete!` takes a key and removes the corresponding key-value pair from the dictionary on the condition that the dictionary contains a key-value pair the key of which matches `delete!`'s argument. The destructively modified dictionary is returned from the operation. `find` takes a dictionary and a key. It searches the dictionary for the corresponding key-value pair and returns the value that corresponds to the key. `#f` is returned if the dictionary does not contain a key-value pair that matches the given key. During this searching process, `find` will need a procedure to check whether or not two keys of type `K` are the same. Therefore, the constructor `new` requires an equality operator that is to be used for checking equality on the key data type `K`. Obviously, `new` returns a newly created dictionary of type `dictionary<K V>`.

Remember that the data values that reside in a storage data structure are called records. The individual data values that make up records are called the *fields* of the record. In our library catalogue the title, the author and the ISBN number are all fields which together constitute one record sitting in the catalogue. The fields of the record that identify the record are said to be the *key fields* (or *key* for short) of the record. All the other fields (i.e. the fields that constitute the value) are said to be *value fields*, *peripheral fields* or *satellite fields*. Hence, in a dictionary, every record consists of the key fields along with the value fields. E.g., in a phone index, the name of the person to search for is typically considered the key field while the other data (e.g. the phone number and the address) is peripheral. Using this new terminology, we can say that it is the task of `find` to find the satellite fields that are associated with the requested key fields.

As already explained, dictionaries are used extremely frequently in computer applications. Therefore, large parts of the text are devoted to the study of different implementations of the `dictionary` ADT.

1.5 Measuring Algorithm Performance

One of the reasons for studying algorithms and data structures in a systematic way is to compare their characteristics in order to determine which algorithm or which data structure is "best". But how can we tell whether an algorithm is "good" or when one algorithm is "better" than another? To answer this question

we have to look at the economic factors that are relevant in computing: people find computers "good" when they are fast and when they have lots of memory. This means that — since time and memory are scarce — software is "good" if it doesn't use a lot of memory and if it is fast. We will therefore try to estimate how much time and how much memory an algorithm consumes.

1.5.1 Performance Measure 1: Speed

In order to measure the amount of time needed to execute an algorithm, we might be tempted to consider using a stopwatch. Another possibility — in some Scheme implementations — is to use the expression (`time expression`) which times the evaluation of the `expression` and puts the resulting numbers on the screen. Hence, we might consider using (`time` (`algorithm input`)) in order to find out how long the `algorithm` runs on the given `input`. We might run such experiments for a few (or even for an extensive set of) inputs in order to get a statistically relevant table of values that describes the speed of an algorithm. This is known as *the experimental approach* to measuring algorithm performance and it has a number of important disadvantages:

Non-Generality The experimental approach is not a general approach. Suppose we have an algorithm and suppose we have tested its speed for a number of different inputs. Suppose we have tried to get a general picture of the algorithm's performance based on the test data. How can we be sure that the algorithm really performs in the way prescribed by that general picture? In fact we cannot. Take for example the problem of sorting a list of data elements. In chapter 5 we will see examples of sorting algorithms that appear to perform quite well for *most* input data, but which have horrible performance characteristics for some pathological cases of input data. The famous QuickSort algorithm (see section 5.4.1) is one of the fastest sorting algorithms for most input data. However, if the input data is already sorted, it becomes extremely slow. This shows us that timing an algorithm with a stopwatch is not enough and that a deeper understanding is necessary about why an algorithm performs in a certain way. Just putting some experimentally acquired numbers in a table is not general enough.

Absoluteness A problem with the experimental approach is that it gives us absolute numbers. These numbers depend on a particular implementation of the algorithm (i.e. the test implementation). This implementation and the experimental data it generates are heavily influenced by the particular hardware it is run on, the Scheme evaluator (or compiler) that was used, the operating system on which it runs, and so on. As a result, the test data gets quite useless after a few years have gone by: processors get faster and Scheme implementations get optimised. The test data is said to be absolute data. What is really needed to compare two algorithms is a *relative* measuring technique which allows us to select the best algorithm

by comparing those algorithms. This is impossible if we are provided with absolute test data that was supplied by two unrelated experiments. Our only option would be to implement both algorithms again (on *our* computer, using *our* Scheme on *our* operating system) and conduct the experiments all over again. What we really need is a general technique that allows us to pick two algorithms "from the shelf" along with their performance characteristics, such that they allow us to tell us something about how the algorithms perform w.r.t. one another. In other words, we need a relative way for measuring algorithm performance.

Because of these two important disadvantages, we do not pursue the experimental track any further. Instead we present a technique that takes *all* possible inputs into account and that allows us to *compare* two performance studies. The basic idea of the technique is to consider an algorithm in terms of the size n of its input and to determine the *number of computational steps* the algorithm needs for an input of that size. For example, we might be interested in how many computational steps the famous QuickSort algorithm needs in order to sort n numbers. Clearly, this amount depends on n. Hence we try to determine a function $f_A(n)$ that gives us a count for the number of computational steps needed to execute an algorithm A on inputs of size n. This function is known as the *performance characteristic of the algorithm*. Notice that the function can be a constant function. For instance, $f_{\texttt{car}}(n) = 1$ for the `car` algorithm (which can be applied to any list of length n) because `car` is always equally fast. However, in most cases, $f(n)$ will not be constant. Given two algorithms A and B and given their performance characteristics $f_A(n)$ and $f_B(n)$, then comparing both algorithms is a matter of (relatively simple) mathematics. For example, if we have two algorithms A and B such that $f_A(n) = 5n$ and $f_B(n) = 10n$ then we can say that A is twice as fast as B.

Let us now try to establish such a performance characteristic. Let us consider the following Scheme procedure `greatest` which takes a list of positive numbers and which returns the greatest element of that list. Our goal is to come up with a function $f_{\texttt{greatest}}(n)$ which gives us an expression (depending on n, the length of the list that gets bound to the `lst` parameter) that allows us to calculate the amount of computation needed to execute the algorithm.

```
(define (greatest l)
  (define (iter el l)
    (cond
       ((null? l) el)
       ((< el (car l)) (iter (car l) (cdr l)))
       (else (iter el (cdr l)))))
  (iter 0 l))
```

We will try to analyse as precisely as possible the number of computational steps that this procedure performs, given a list `l` = (`cons a d`) of length n (i.e., the length of `d` is $n - 1$). We will use T(`(greatest l)`) to denote the number of computational steps needed to perform the call to `greatest` with `l`

as its argument. Hence, $f_{\texttt{greatest}}(n) = T(\texttt{(greatest l)})$ where `l` is a list of length n. When calling `greatest`, a call to `iter` immediately follows. Hence, $T(\texttt{(greatest l)}) = T(\texttt{(iter 0 l)}) + 1$. Inside `iter`, a `null?` test is executed and subsequently (if the list is not empty), the car of the list is compared to the current `el`. In Scheme, all this requires 3 computational steps. If this comparison succeeds, a recursive call is made after having called `car` and `cdr` (i.e. 3 steps) or a recursive call is made after having called only `cdr` (i.e. 2 steps). This means that, either $T(\texttt{(iter el l)}) = 6 + T(\texttt{(iter a d)})$, or $T(\texttt{(iter el l)}) = 5 + T(\texttt{(iter el d)})$. In the very last step of the recursion, all we have to do is the `null?` test which is a single computational step. As a result, we have two extreme situations. In the worst-case scenario, the comparison fails in every step of the recursion. This only happens when the list contains its numbers in ascending order, such that a new maximum is found in every call of `iter`. Consequently, we have $T(\texttt{(iter el l)}) = 6n + 1$, and thus $f_{\texttt{greatest}}(n) = 6n + 2$. In the best-case scenario, the comparison succeeds in every step of the recursion. It occurs when no element in the list is greater than 0 such that the `car` procedure will never be used in recursive calls. In that case, $T(\texttt{(iter el l)}) = 5n + 1$, and thus $f_{\texttt{greatest}}(n) = 5n + 2$.

This clearly illustrates that we have three kinds of analyses to measure the time efficiency of an algorithm in terms of the number of computational steps performed by the algorithm:

Worst-case Analysis In a worst-case analysis, we try to estimate the number of execution steps performed by an algorithm by selecting the worst possible choice of every option offered by the algorithm. This of course, depends on the input provided to the algorithm. A worst-case analysis assumes the worst possible imaginable input. It means that every `if` test or `cond` in the algorithm is scrutinized and the worst possible branch (in terms of number of computational steps to be executed) is selected. For the majority of the algorithms presented in this text, a worst-case analysis will be presented. Given a worst-case algorithm performance characteristic $f(n)$, we can say that the algorithm will never perform more poorly than $f(n)$ for inputs of size n.

Best-case Analysis A best-case analysis is an analysis of the algorithm in which we select the most optimistic branch in every possible `if` test and `cond` the algorithm has to execute. This is the branch with the least number of potential computational steps to be taken. Clearly, *in general* this is not very useful an analysis. Given an algorithm A for which the best-case performance characteristic is $f(n) = 1$ but for which the characteristic gives $f(n) = 2^n$ for *all other inputs* except for the best case, then we clearly have a very poor algorithm even though it is very fast for *some* case (i.e. the best case). Therefore, a best-case analysis is not a very useful analysis. Nevertheless, as we will see, some algorithms (notably sorting algorithms) do have interesting best-case analyses. If the chances on the best-case input are reasonably high, then one might consider selecting the algorithm anyhow.

Average-case Analysis One might ask the question "Ok, say we have a best-case analysis and a worst-case analysis. What would be the algorithm's performance characteristic for the *average* input?". Unfortunately, answering this question is not an easy task. In order to perform an average case analysis, one needs to find out what kinds of inputs are likely to occur and what kinds of inputs are very unlikely to occur. This usually requires a good deal of knowledge on probability theory and the resulting mathematical analysis is often not even worth the effort as the outcome is usually not fundamentally different from a worst-case analysis. However, in some occasions, the worst-case is so rare that an average case analysis is more representative.

Given these considerations, most of the algorithms presented in this text will be analyzed using a worst-case analysis. We present an average-case analysis for a few algorithms for which the probabilistic distribution of the input is easy to determine. We also present a few best-case analyses when the outcome of a best-case analysis is fundamentally different from the worst-case analysis *and* when the input for that best-case execution of the algorithm is likely to occur. But again, most of our time will be devoted to worst-case analyses.

1.5.2 The Big Oh Notation

Looking back at the analyses of $T($`(iter el l)`$)$ presented in the previous session, we can make the following observations:

- We are counting the number of basic computational steps and *not* the execution time of those steps. But in a real Scheme implementation it might as well be the case that the execution of the $<$ predicate requires much more time than the execution of the `null?` predicate. So, maybe "6" would be a better estimate than "5", at least for some implementations of Scheme. Hence, hiding both the execution time for $<$ and the time needed to execution `null?` in the factor "5" is not very precise.

- Suppose that we replace the `null?` test in the algorithm by an (`eq?` '()) test and suppose that our Scheme implementation runs the (`eq?` '()) test faster than the `null?` test. If we would do our analysis of the new version of the algorithm (i.e. the one using (`eq?` '())) we would get a factor $7n$ instead of $6n$ in our final formula (since evaluating '() requires 1 computational step as well) while the actual execution time would be faster than the version of the algorithm that gave us $6n$. Again, this argument shows us that the numbers "6" or "7" are not very meaningful.

- The same thing can be said about a number of constant steps. Suppose that the first call of `iter` would be performed using the car of the list as the first maximum (instead of `0`). In that case, our worst-cast analysis would give $6n + 3$ instead of $6n + 2$. Is this additional constant factor "1" really relevant in the way we look at this algorithm? Suppose that

yet another variant of the algorithm would call two procedures before starting the iteration (e.g. a call to `display` followed by a call of `car` as before). Now our analysis would give $6n + 4$. Again we can ask ourselves the question whether the difference between $6n + 2$, $6n + 3$ and $6n + 4$ is big enough to justify this kind of precise reasoning. Especially when n gets larger and larger, we see that the role of the constant factor (i.e. 2, 3 and 4) gets smaller and smaller. When applying the algorithm to lists containing thousands of elements, the role of the constant gets neglectable.

Given these arguments, it seems like the only reasonable thing to say is that the performance characteristic of our algorithm is "something of the form $f(n) = an + b$ where a and b are relatively small numbers that do not depend on n (i.e. they are constant)". It seems that a and b are not really meaningful: b is not relevant for non-small values of n and a's exact value depends on so many technical factors (i.e. factors that depend on a particular Scheme implementation) that its precise determination is both imposible and useless. Therefore, the only thing we can say is that the execution time of our algorithm is linearly dependent on the size of its input list. The longer the list, the longer the algorithm runs in a linear sense: lists that are k times as long will require k times as much time. What we really want to grasp is *how* $f(n)$ grows as n gets larger.

Perhaps it is more instructive to have a look at some numbers. The numbers in the table of figure 1.5 show the different growth rates for different types of $f(n)$ *for large* n. In order to see the impact of the different types of functions (as opposed to small difference in constants like a and b *within* e.g. the linear case), we advise the reader to compare n^3 with $n.log(n)$ for $n = 1024$. Even though in practice, the input for algorithms is typically much bigger than 1024 (e.g. many people have an iTunes play list that contains 5000 songs), the case $n = 1024$ already reveals that using an $n.log(n)$ algorithm really pays off w.r.t. using an n^3 or even an n^2 algorithm. From the table it clearly shows that a version of iTunes in which the "sort" is implemented by an $n.log(n)$ algorithm is much faster than a version in which it is implemented by an n^2 algorithm!

In what follows, we present a number of mathematical tools that allow us to express this kind of coarse grained reasoning. They allow us to express that the functions $f_1(n) = n$, $f_2(n) = 2n + 5$ and $f_3(n) = 15n - 44$ are all "the same". We need such tools because we want to express that these functions are *not* the same as $g(n) = n^2$. Indeed, for large inputs (i.e. large n) it is not hard to see that n^2 is *considerably* larger than any combination that looks like $an + b$. In order to fully grasp this, we invite the reader to have a look at figure 1.4 which displays the relative growth of four functions. Two of them are linear (i.e. of the form $an+b$) and the other two are quadratic (i.e. of the form an^2+bn+c). We clearly see that for large n (notice that the figure merely shows the growth untill $n = 25$, a fairly small input size) the quadratic functions grow *much* faster than the linear ones (which hardly differ from the x-axis). The point of our reasoning is that n^2 grows *much* faster than n or $5n$. Hence, n and $5n$ are "the same" while $5n$ and n^2 are not.

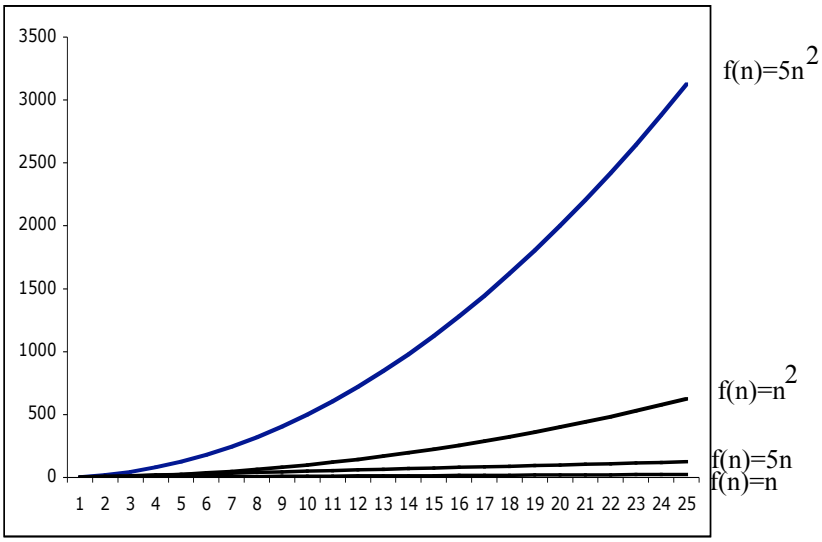

Figure 1.4: Comparing the growth of function

Let us now have study the mathematical tools that allow us to express this notion of "sameness". The following sections will present three different views on how to express that two functions are "roughly the same" or, conversely, "considerable different". All three of them have in common that they make an *asymptotic comparison* of the functions; i.e., they compare the functions "for large n" thereby omitting the smaller n. The reason for this is as follows. Suppose we have an algorithm the performance characteristic of which is $f(n) = 10n$ and suppose we have another one for which the characteristic is $g(n) = n^2$. Following the above reasoning, we agree that the g characteristic is significantly worse than the f characteristic. Applying the algorithms to inputs of huge sizes (e.g. $n = 10^7$) makes this difference painfully tangible. Nevertheless, it is easy to see that $n^2 \leq 10n$ for all $n \leq 10$. In other words, g is better than f for some small number of uninteresting n. Therefore, it would be a mathematical error to say that g *is* bigger than f because this is only a true statement if $g(n) > f(n)$ for *all* n. We therefore only compare g and f "only for large n", i.e. for $n \geq n_0$ where n_0 is a certain constant whose exact value is not really relevant.

Big Theta

We now present the first of a set of mathematical tools that allow us to say that a function $f_1(n)$ is behaving "similar" to a function $f_2(n)$. We will do so by looking at a set of functions $\Theta(f_1(n))$ which is the set of all functions that are roughly the same as $f_1(n)$. Once we have defined this set, we can then say that

n	$log(n)$	$\sqrt{n}$	n	$n.log(n)$	n^2	n^3	2^n
2	1	2	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	3	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	6	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.85×10^{19}
128	7	12	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	23	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	1.79×10^{308}

Figure 1.5: Growth of Functions of n

$f_2(n)$ is similar to $f_1(n)$ by writing $f_2(n) \in \Theta(f_1(n))$.

Here is the definition:

$$\Theta(g(n)) = \{f | \exists c_1, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\}$$

In order to understand this definition formally, we refer to figure 1.6 which shows an imaginary function $g(n)$ and an imaginary function $f(n)$ which is in $\Theta(g(n))$. The idea of n_0 is to ignore "initial" irregularities for small n in the behaviour of $f(n)$ and $g(n)$ because we are only interested how $f(n)$ and $g(n)$ relate to each other in general, for big values of n.

The above definition formally defines what it means for an f to be "roughly the same as" g. Figure 1.6 shows the situation graphically. As we can see, the idea of Big Theta is to determine two constants c_1 and c_2 such that g lies between "a c_1-fold of f" and "a c_2-fold of f". In other words, apart from the factors c_1 and c_2, we can say that $f(n)$ is "roughly the same as" $g(n)$. More precisely, we say that $g(n)$ *is an asymptotically tight bound to* $f(n)$.

Big Theta is a very convenient mathematical tool to express that a function f is roughly the same as g. Unfortunately it is not always easy to prove this property. An even bigger problem is that Big Theta is sometimes too precise a mathematical tool. Remember from section 1.5.1 that we are often concerned with a worst-case analysis and this does not always fit Big Theta since our algorithm *might* have a best-case analysis performance characteristic that is of another Bit Theta. For instance, in section 5.3.2, we will present a sorting algorithm called insertion sort. Given a vector of length n, it will turn out to be the case that the sorting algorithm is $\Theta(n^2)$ if we perform a worst-case analysis (this will be the situation where the vector is completely unsorted or even sorted in reverse order). However, performing a best-case analysis on the same algorithm will reveal that the algorithm is $\Theta(n)$ (the best case will be the situation where the vector is already sorted before applying the sorting algorithm). Hence, it will be technically incorrect to say that the insertion sort algorithm "is roughly n^2" or "is roughly n" for *all* inputs if we use the Θ

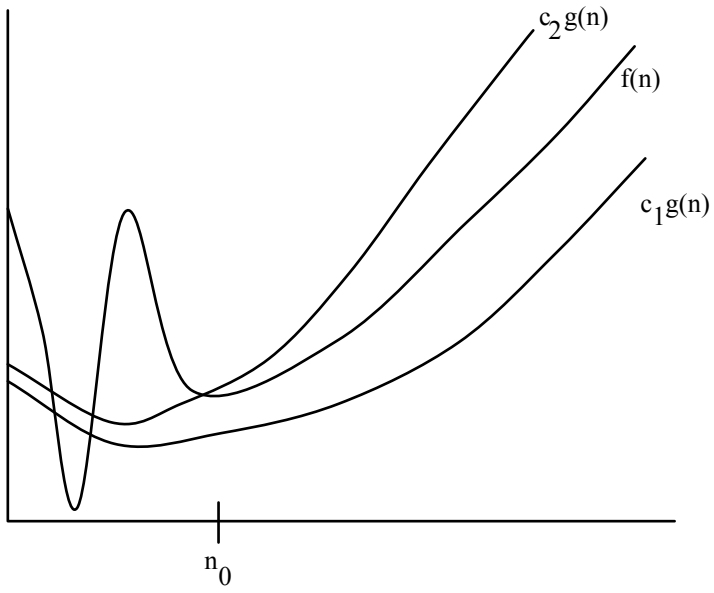

Figure 1.6: $f(n) \in \Theta(g(n))$

tool. The tool seems to be too precise if we want to express the performance characteristic irrespective of the properties (i.e. worst-case or best-case) of the input.

Big Oh

The problem with Θ is that it requires us to approximate functions both with an upper bound as well as a lower bound. The Big Oh notation is less strict in that it only requires us to find upper bounds. Here is the definition:

$$O(g(n)) = \{f | \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

Now let us reconsider the problem with the insertion sort described above. If we were to say that "insertion sort *is* an $O(n^2)$" algorithm, this would not be a mathematical error. After all, the worst-case analysis of insertion sort tells us that it "is an n^2 algorithm" while the best-case analysis tell us that it "is an n algorithm". Therefore we are *not* allowed to say that insertion sort *is* $\Theta(n^2)$ since $n \notin \Theta(n^2)$. But we *are* allowed to say that "insertion sort is an $O(n^2)$ algorithm" since both $n^2 \in O(n^2)$ and $n \in O(n^2)$. Figure 1.7 shows what it means for a function $f(n)$ to be $O(g(n))$. Notice that we are no longer speaking about the functions being "roughly the same". O merely gives an *asymptotic upper bound* and is therefore *much* less precise than Θ. For example, it would

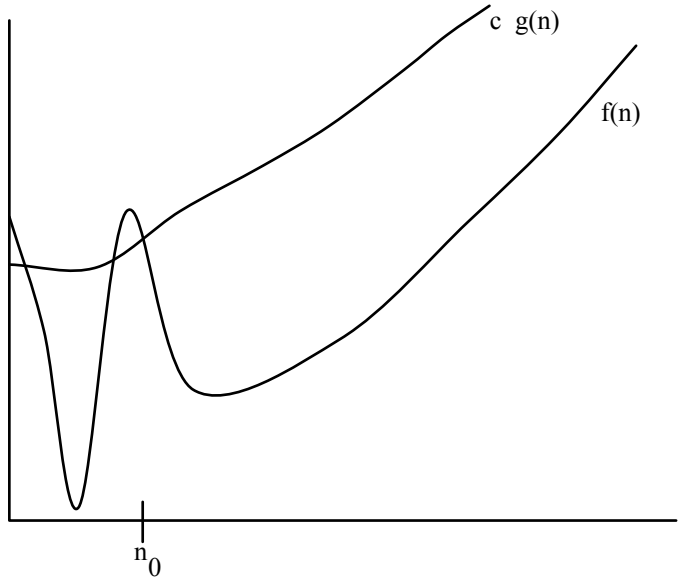

Figure 1.7: $f(n) \in O(g(n))$

be perfectly valid to say that $n \in O(2^n)$ since 2^n is clearly an upper bound for n. Hence, whenever we want to use $O(g)$ to estimate the performance of an algorithm we will try to be as precise as possible by selecting a g as small as possible. Informally, $f(n) \in O(g(n))$ means that the worst-case for f will be no worse than g.

Big Omega

Instead of looking for asymptotic *upper* bounds for functions, we might also look for asymptotic *lower* bounds. This is the role of Big Omega. It is defined as follows:

$$\Omega(g(n)) = \{f | \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

Figure 1.8 shows us what it means for a function $f(n)$ to be in $\Omega(g(n))$. Informally it means that f will be at least as big as g for large n. Since Big Omega is a lower bound, we mainly use it to speak about best-case analyses for algorithms. Just like O, Ω is not very precise a tool. For example, it is mathematically correct to say that $2^n \in \Omega(n)$ since n is indeed a lower bound for the exponential function. Needless to say, this is pretty meaningless a statement. Remember that we said that a best-case analysis of the insertion sort algorithm will reveal that the algorithm performs "like n" in the best case (i.e. when the data is already sorted). More formally we can now say that insertion sort is an

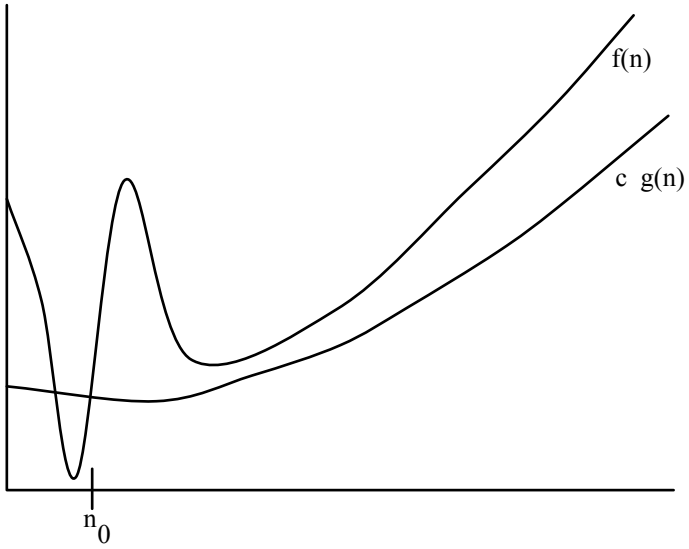

Figure 1.8: $f(n) \in \Omega(g(n))$

$\Omega(n)$ algorithm: given any input, it will always take more execution time than n. Stated otherwise, $f(n) \in \Omega(g(n))$ expresses the fact that the best case for f will not be better than g.

Theorem: Relations between Θ, Ω and O

For the sake of completeness, we relate the three notations with each other. This is done formally by the following theorem:

For any two functions f and g, $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$

This is actually not a surprising result: the theorem merely states that a function f is *tightly* bound by g if and only if f has g both as a *lower* bound and as an *upper* bound.

Remarks

From the previous discussion, we conclude that Θ is the most precise instrument for measuring the performance of algorithms. Nevertheless, since Θ is not always easy to determine and pathological input cases (e.g. a few specific input values) can cause a good candidate for Θ not to hold in general. Therefore, we will usually opt for O in which we will try to be as precise as possible by taking the smallest f such that an estimate $O(f(n))$ holds for our algorithm. After all, saying that our algorithm is in $O(n)$ is more precise than saying that it is in

$O(n^2)$. Moreover, O usually fits a worst-case analysis better and this is what we are usually interested in. O gives us a rough classification of algorithms. However, blindly using O to compare algorithms is not always without danger.

- One of the practical consequences of using the O notation is that constant coefficients do not matter. Indeed, in the definition of O, a constant c has to be found to find an upper bound for $f(n)$. This means that, whenever we have $f'(n)$ that is a constant factor bigger than $f(n)$ (in other words: $f'(n) = a.f(n)$) then it suffices to take $c' = c.a$ in order to have a constant c' that is an upper bound for $f'(n)$. Hence $f(n)$ and $a.f(n)$ belong to the same Big Oh. However, sometimes caution is required. Clearly, $10^{10}n$ is much bigger than n^2 for most of the practically occurring n. Hence, an algorithm with quadratic time might *in some cases* be more beneficial than a linear algorithm, even though $O(n)$ is theoretically better than $O(n^2)$. Nevertheless, as computers get faster, the constant coefficient gets less relevant (remember that we count the *amount* of steps and not the absolute time) and the quadratic algorithm gets less attractive.

- A second word of caution is needed when comparing two algorithms with performance characteristics that are of the *same* order of growth. As we will see in section 3.4.2, using some clever tricks here and there in one's source code can make the code twice as fast. Even though both algorithms are of the same order (e.g. $O(n/2)$ and $O(n)$), one algorithm can be substantially faster than another one. Comparing the performance characteristic of two different algorithms is much more instructive when their performance characteristics are of different orders of growth than when they are of the same order of growth.

Simplification Rules

The following rules make it much simpler to come up with an expression for $O(f_{\texttt{A}}(n))$ for a given Scheme procedure `A`. Their proof follows directly from the definition of O.

1. Constants in front of terms (i.e. coefficients) play no role: $O(an^k) = O(n^k)$. This was already shown above.

2. A corollary of this is that $O(c) = O(1)$ for all constants c. An algorithm the performance characteristic of which is $O(1)$ executes in constant time, i.e. it is equally fast for all possible inputs.

3. Another corollary is that the basis for logarithms is of no importance. Since $log(a^b) = b.log(a)$, we can prove that $log_a(x) = \frac{log(x)}{log(a)}$. Hence, $O(log_a(x)) = O(log(x))$ since coefficients are not important.

4. Only dominant terms are important: If $f(n) = t_1 + t_2$ where the order of t_1 is higher that the order of t_2 then $O(f(n)) = O(max(t_1, t_2)) = O(t_1)$.

At this point we refere back to the table in figure 1.5 which shows us the orders that occur most often. We conclude that for O:

$$1 < log(n) < \sqrt{(}n) < n < n.log(n) < n^{k-1} < n^k < 2^n < n!$$

Finding O for Scheme procedures

Now that we have developed the mathematical machinery necessary to reason about the performance $O(f_A(n))$ of an algorithm A, it remains to be discussed, given a concrete Scheme procedure `A`, how we can — based on the source text of `A` — come up with the function $f_A(n)$. We will do this in two phases. First, we consider Scheme's primitive computational elements one by one and estimate how much computational effort they require. Second we will see how to combine this knowledge in order to determine the performance characteristic of a compound Scheme procedure, the body of which is composed out of these expressions. Hence, the idea is to distill the performance characteristic of a Scheme procedure by combining the performance characteristics of the expressions that make up the body of the procedure.

1.5.3 Analyzing Scheme Expressions

Let us systematically consider Scheme's expression types in order to come up with a systematic methodology for finding $f_A(n)$ given the source text of the body of the Scheme procedure `A`. The following list enumerates all Scheme's expression types and estimates their performance characteristic in terms of the size n of the input of the procedure to which they belong:

- $O(f_{\texttt{(define v E)}}(n)) = O(f_{\texttt{E}}(n))$. This is because a `define` expression contains a subexpression `E`. The number of steps needed for executing the `define` expression itself is $O(1)$. However its subexpression `E` can be computationally intensive. Therefore, the order of growth of the entire expression is $O(f_{\texttt{E}}(n))$ where $f_{\texttt{E}}(n)$ is `E`'s performance characteristic. For example, the expression (`define x 3`) will be $O(1)$. However, the expression (`define x (fac n)`) requires $O(n)$ computational steps because $f_{\texttt{fac}} \in O(n)$. In other words, the efficiency of the compound `define` expression depends on the efficiency of its subexpression.

- $O(f_{\texttt{(set! v E)}}(n)) = O(1) + O(f_{\texttt{E}}(n)) = O(f_{\texttt{E}}(n))$. The reasoning for `set!` is exactly identical to that for `define`.

- $O(f_{\texttt{(set-car! p E)}}(n)) = O(f_{\texttt{(set-cdr! p E)}}(n))$ which is the same as $O(f_{\texttt{p}}(n) + f_{\texttt{E}}(n))$ and thus $O(max(f_{\texttt{p}}(n), f_{\texttt{E}}(n)))$. In the case of `set-cdr!` and `set-car!`, there are two subexpressions: one to calculate the pair `p` in which to set the car or cdr, and another one to calculate its new value designated by the expression `E`. Hence, we have to add the performance characteristics of both subexpressions. Furthermore, we only have to take the greatest term of this sum.

- $O(f_{\texttt{(lambda lst body)}}(n)) = O(1)$. Evaluating an anonymous `lambda` form only takes one computational step in order to *create* the procedure object.

- $O(f_{\texttt{(if c t e)}}(n)) = O(max\{f_{\texttt{c}}(n), f_{\texttt{t}}(n), f_{\texttt{e}}(n)\})$. Evaluating an `if` expression causes the condition `c` to be evaluated, followed by the evaluation of `t` or `e`. Hence, the performance characteristic is $f_{\texttt{c}}(n) + f_{\texttt{t}}(n)$ or $f_{\texttt{c}}(n) + f_{\texttt{e}}(n)$. Hence, we have to take the maximum of these three performance characteristics.

- $O(f_{\texttt{(let ((v1 e1) (v2 e2) ... (vn en) b1 b2 .. bk)}}(n))$ $= O(max_i(f_{\texttt{ei}}(n)) + max_j(f_{\texttt{bj}}(n))) = O(max_{i,j}\{f_{\texttt{ei}}(n), f_{\texttt{bj}}(n)\})$. In the case of a `let` expression that consists of several subexpressions, we have to add the performance characteristics of *all* the subexpressions. However, we know that only the dominant term is relevant in such an addition. Therefore, taking the maximum of the performance characteristics of the subexpression suffices.

- $O(f_{\texttt{(cond ((c1 a1) ... (cn an)))}}(n)) = O(max_i\{f_{\texttt{ci}}(n), f_{\texttt{ai}}(n)\})$. For `cond`, the same reasoning applies.

- $O(f_{\texttt{(begin e1 ... en)}}(n)) = O(f_{\texttt{e1}}(n)+...+f_{\texttt{en}}(n)) = O(max_i(f_{\texttt{ei}}(n)))$. Similarly, the performance characteristic of a `begin` expression is the dominant term of the sum of the performance characteristics of its subexpressions.

- $O(f_{\texttt{(prim a1 ... an)}}(n)) = O(1) + O(f_{\texttt{a1}}(n)) + ... + O(f_{\texttt{an}}(n))$ for $\texttt{prim} \in \{\texttt{+}, \texttt{-}, \texttt{and}, \texttt{or}, \texttt{eq?}, \texttt{eqv?}, \texttt{vector-ref}, \texttt{vector-set}, \texttt{car}, \texttt{cdr}, ...\}$. Calls of most *primitive* procedures such as $+$, $-$, etc result in a performance characteristic that is in $O(1)$ since they only require one computational step. On top of this, the arguments have to be evaluated which gives rise to performance characteristics $f_{\texttt{ai}}(n)$. Again, the dominant term in the sum of all these subexpressions survives. However, caution is required when using primitive procedures that work with Scheme's built-in list data structure. Scheme features quite a lot of primitives that are used to convert data values into lists and vice versa. Examples are `list->vector`, `vector->list`, `string->list` and `list->string`. Needless to say, these do not exhibit $O(1)$ behaviour but depend on the size of the vector, string or list that is passed as an argument. They exhibit an $O(n)$ behaviour where n is the length of the input list or the input vector.

- $O(f_{\texttt{(f a1 ... an)}}(n)) = O(f_{\texttt{f}}(n) + f_{\texttt{a1}}(n) + ... + f_{\texttt{an}}(n))$. Determining the performance characteristic for a *non-primitive* procedure call has to be done by adding (i.e. selecting the dominant term) of the performance characteristics of all subexpressions, augmented with the performance characteristic of the procedure `f` that is actually executed. The

following section explains how to establish this characteristic for arbitrary Scheme procedures.

An important Scheme expression that was omitted in this list is the so called *named* `let`. In the following example, a named `let` is used to implement an iterative version of the good old factorial procedure:

```
(define (fac n)
  (let fac-loop
    ((result 1)
     (factor n))
    (if (= factor 0)
      result
      (fac-loop (* result factor) (- factor 1)))))
```

We will use the named `let` construct quite frequently in this text. What can we say about the performance characteristic of the named `let` construct given the performance characteristics of its subexpressions? Unfortunately, the situation is not simple. Fortunately there is a way to get rid of the named `let` construct when establishing performance characteristics. The idea is to replace the named `let` by two subexpressions. The first subexpression defines a local procedure whose purpose it is to implement the loop expressed with the named `let`. The second expression launches the loop by calling that local procedure. For the `fac` example given above, this transformation results in the following code:

```
(define (fac n)
  (define (fac-loop result factor)
    (if (= factor 0)
      result
      (fac-loop (* result factor) (- factor 1))))
  (fac-loop 1 n))
```

This code has the benefit that it is a (recursive) procedure that does not use the named `let` construct. Establishing its performance characteristic is easy as soon as we know how to establish the performance characteristic of a procedure. This is the topic of the following section. We conclude that it is always possible to transform a named `let` expression into two subexpressions; one to define a local procedure and another one to call that procedure.

1.5.4 Analysing Scheme Procedures

Now that we have a precise definition of the performance characteristic of Scheme's primitive computational building blocks, the final topic of this chapter is to present a technique that allows us to distill the performance characteristic of a scheme procedure `P` based on the expression `B` that forms the body of `P`. We distinguish between recursive procedures and non-recursive procedures.

Non-recursive Procedures If `P` is a non-recursive procedure (i.e. a procedure that does not call itself directly or indirectly) then the situation is simple: we simply analyse the body expression `B` using the rules presented in section 1.5.3. In case the body has multiple subexpressions then the performance characteristic is nothing but the sum of the performance characteristics of those subexpressions. For example, consider the following Scheme procedure.

```
(define (weird vector)
  (define len (vector-length vector))
  (if (odd? len)
    (a-linear-function len)
    (a-quadratic-function len)))
```

Applying what we know gives us a performance characteristic $f_{\texttt{weird}}(n) \in O(n^2)$ where n is the length of the input vector. The reasoning is as follows: the characteristic for the procedure is the sum of the characteristics of the subexpressions. However, we know from the simplification rules that only the dominant term is of importance in such a sum. Therefore, we only have to consider the maximum of the performance characteristics of the subexpressions. There are two such subexpressions: the `define` expression and the `if` expression. The `define` subexpression gives $O(1)$ since its subexpression consists of a call to a primitive procedure of $O(1)$ (i.e. `vector-length`). The performance characteristic for the `if` expression is the maximum of the performance characteristics of its subexpressions. Assuming that $f_{\texttt{a-quadratic-function}}(n) \in O(n^2)$ and $f_{\texttt{a-linear-function}}(n) \in O(n)$, we thus get $f_{\texttt{weird}}(n) \in O(n^2)$.

Recursive Procedures If `P` is a recursive procedure because its body `B` calls `P` again[2], then the situation is much more complex. Let us consider our good old `fac` again:

```
(define (fac n)
  (if (= n 0)
    1
    (* n (fac (- n 1)))))
```

When applying the above rules, one soon comes up with the fact that $f_{\texttt{fac}}(n) = 1 + f_{\texttt{fac}}(n-1)$. In other words, in order to come up with the performance characteristic for a recursive procedure, we need the performance characteristic of the procedure in its turn. Solving such mathematical "recursive equations" can be quite simple. However, in many cases solving the equation is all but trivial. A precise mathematical derivation of performance characteristics of recursive functions is therefor outside the

[2]Remember that there is a difference between a recursive *procedure* and a recursive *process*. A recursive procedure is a procedure that syntactically calls itself. This can give rise to both a recursive as well as to an iterative process.

scope of this text. To avoid such complex analysis, we will take a rather intuitive approach in this text by applying the following rule of thumb:

Rule of Thumb: If we have a recursive procedure that takes an argument the input size of which depends on n, then first determine the performance characteristic $O(b(n))$ for the body of the procedure without taking the recursive procedure calls into account. Then determine an estimate $r(n)$ for the number of recursive calls (depending on n) that will be made. The performance characteristic for the entire recursive procedure will be $O(b(n).r(n))$.

Examples

Example 1: For our good old `fac` shown above, it should come as no surprise that the characteristic of its body is $O(1)$ since that body merely consists of primitive procedure applications. The number of times `fac` will be called is determined exactly by the size of its input n since every call with argument `n` (except for the last one) gives rise to exactly one call with argument `n-1`. In other words $b(n) \in O(1)$ and $r(n) \in O(n)$. Hence $f_{\texttt{fac}} \in O(1.n) = O(n)$.
Example 2: As a second example, let us have a look at procedures for computing Fibonacci numbers. The following code excerpt shows two Scheme procedures for computing Fibonacci numbers. `fib1` is the canonical recursive algorithm. `fib2` is an iterative algorithm.

```
(define (fib1 n)
  (if (< n 2)
    1
    (+ (fib1 (- n 1)) (fib1 (- n 2)))))

(define (fib2 n)
  (define (iter n a b)
    (if (= n 0)
      b
      (iter (- n 1) b (+ a b))))
  (iter n 0 1))
```

Let us now try to come up with the performance characteristics $f_{\texttt{fib1}}(n)$ and $f_{\texttt{fib2}}(n)$ for `fib1` and `fib2` respectively.

- **The `fib1` case:** `fib1` is a recursive procedure. Following our rule of thumb, we have to find $b(n)$ and $r(n)$. Clearly, $b(n) = 1$ since (apart from the recursive calls), the body of `fib1` solely consist of applying primitive procedures such as $+$, $<$ and $-$. In order to come up with $r(n)$, we observe that every call to `fib1` gives rise to two recursive procedure calls. Notice that the number of recursive calls generated by `(fib1 (- n 2))` will be smaller than `(fib1 (- n 1))`. Hence, it is fair to say that the number of recursive calls generated by `(+ (fib1 (- n 1)) (fib1 (- n 2)))` is less than the number of recursive calls generated by `(+ (fib1 (- n 1))`

`(fib1 (- n 1))`. If we were to replace the body of `fib1` by the latter expression, we would have a recursive procedure in which every procedure call produces *exactly* two recursive calls. This means that $r_{\texttt{fib1}}(n) \leq 2^n$. As a consequence, $f_{\texttt{fib1}} \in O(2^n)$.

We can ask ourselves whether we can be more precise than this. As a matter of fact we can, but this requires more complex mathematical reasoning which is beyond the scope of this text. It is possible to prove that the number of recursive calls made by `fib1` is actually a bit smaller than 2^n, but not much. It can be shown that the number of calls is always proportional to ϕ^n where $\phi = 1.61$ which is the famous golden ratio. In other words, we can prove that $r_{\texttt{fib1}}(n) = \phi^n$. Hence, we can say that $f_{\texttt{fib1}} \in \Theta(\phi^n)$.

- **The `fib2` case:** `fib2` calls `iter` and `iter` is a recursive procedure as well (even though it generates an iterative process). Again, we have $b(n) = 1$ since the body of `iter` only consists of applications of primitive Scheme procedures. It should be clear that $r(n) = n$ since every call to `iter` either stops or gives rise to one more recursive call with an argument that is exactly 1 smaller than the argument of the original call. In other words, $f_{\texttt{fib2}} \in O(1.n) = O(n)$.

Multiple Parameters

Until now, we have only considered performance characteristics for Scheme procedures that take one argument the size of which is given by n. But what does it mean to establish a performance characteristic for algorithms that take multiple arguments? Consider for example the following algorithms that calculate the sum and product of two Scheme numbers `n` and `m`.

```
(define (sum n m)
  (if (= n 0)
    m
    (+ 1 (sum (- n 1) m))))

(define (times n m)
  (if (= m 1)
    n
    (sum n (times n (- m 1)))))
```

What can we say about $f_{\texttt{sum}}$ and $f_{\texttt{product}}$? Since these procedures take two arguments n and m, the performance characteristics will be mathematical functions that depend on two arguments as well. Hence, we are looking for two functions $f_{\texttt{sum}}(n, m)$ and $f_{\texttt{times}}(n, m)$.

Clearly, `sum` is a procedure whose body is in $O(1)$ provided that we ignore the recursive call. Hence, $b_{\texttt{sum}}(n, m) = 1$. Since it breaks down its first parameter `n` until it reaches zero, it obviously executes the recursion n times. Hence we can say that $r_{\texttt{sum}}(n, m) = n$. By applying our rule of thumb, we conclude that

$f_{\texttt{sum}}(n, m) \in O(n)$. Looking at `times`, we observe that its body consists of a call to `sum`. Hence, $b_{\texttt{times}}(n, m) = n$. Furthermore, `times` is called recursively m times, resulting in $r_{\texttt{times}}(n, m) = m$. Again, our rule of thumb yields the overal result for `times` which is $f_{\texttt{times}}(n, m) \in O(n.m)$.

1.5.5 Performance Measure 2: Memory Consumption

Most of the algorithms presented in this text operate on data structures that already reside in our computer's central memory. For example, a sorting algorithm sorts the entries of an iTunes playlist. The playlist is implemented as a data structure (e.g. a list or a vector) residing in main memory. Furthermore, most of the algorithms do not require *additional* memory on top of the memory that is already occupied by the data structure on which the algorithm operates. Algorithms that meet this property are said to be *in-place*. We only discuss very few algorithms which are not in-place. For those algorithms that are not in-place, we will estimate the amount of memory that is needed for the execution of the algorithm. Luckily, we can use the same mathematical tools (i.e. Ω, O and Θ) to express the amount of memory. For example, we might say that a certain algorithm consumes an mount of memory that is $O(n)$. This means that the algorithm itself will (while it is running) reserve a number of memory cells on top of the memory cells that are occupied by the input. The amount of memory is directly proportional to n.

Examples

On first sight, the implementation of the Fibonacci algorithm `fib1` shown above does not consume any memory apart from the argument itself (i.e. it is in-place). However, since the procedure generates a recursive process, this means that every recursive call has to remember what to do after returning from that recursive call. From the procedure body, we see that a call to `fib1` with argument `n` will call `fib1` with argument `n-1` after it has called `fib1` with argument `n-2` (or the other way around, depending on the order in which Scheme evaluates the arguments of $+$). This means that the recursion depth of `fib1` will be n. Given the fact that `fib1` generates a true recursive process, our Scheme interpreter will have to remember n times what to do after returning from the chain of recursive calls. In other words, `fib1` requires $\Theta(n)$ memory to execute. Hence, `fib1` is *not* an in-place algorithm.

Now let's consider the implementation of `fib2`. In contrast to `fib1`, `fib2` generates an iterative process. Even though every call to `fib2` generates a call to `fib2` in its turn (apart from the final call), the Scheme evaluator does not have to remember anything about what to do when returning from the latter call. The only thing that remains to be done after returning from the latter call, is to return from the former call. This means that we can safely return from the very last call to the very first call while ignoring all intermediate calls. No additional memory is needed to remember partial results from intermediate

calls. Hence, `fib2` requires $\Theta(1)$ memory which means that it is a true in-place algorithm.