Chapter 3

Data Types

This book defines two kinds of *data types*: *abstract data types* and *concrete data types*.

We use the conventional definition of *abstract data type*:

Definition 3.1. *An* abstract data type *is a data type defined by the possible operations on instances of the type, without reference to the way in which these instances are represented in the memory of a computer.*

As a simple example of an abstract data type, consider a *stack*. (Chapter 13 offers more detail about this abstract data type.) The operations on a stack are:

- Test whether the stack is empty.
- Push an object onto the stack.
- Retrieve the top of the stack; that is, given that the stack is not empty, return the top item on the stack without changing the stack.
- Pop the stack; that is, given that the stack is not empty, remove the top item on the stack.

In the literature, there are two ways of defining the operations on an abstract data type, namely *imperatively* and *functionally*.

In the imperative definition of an operation, it is important to preserve the *identity* of the instance of the type. An operation such as *push* or *pop* must *modify* the instance of the abstract data type.

In contrast, in the *functional* definition, no instance of the abstract data type is ever modified. Operations such as *push* or *pop return a new instance* of the type without modifying the instance passed to them as an argument.

In this book, most of the abstract data types described employ the imperative definition.

A data type that is not abstract is a *concrete data type*. Here is the definition of a concrete data type:

Definition 3.2. *A* concrete data type *is a data type defined by the layout of an instance of the type in the memory of a computer.*

A very simple example of a concrete data type is a *cell of a simply linked list*, which is a pair of consecutive *words*. (For more detail about this idea, see Chapter 6.) Another example of a concrete data type is an *array*, which is a collection of consecutive words in the memory of a computer.

There are data types that can be defined either as concrete or abstract data types. Take, for example, a *simply linked list*. It can be described in both ways, concretely or abstractly. As an abstract data type, a simply linked list would have the following operations:

- Test whether the list is empty.
- Return the first element of the list, provided that the list is not empty.
- Return the list of elements other than the first, provided that the list is not empty.

That definition is *functional* because the last operation returns an instance different from the type without modifying the instance passed as an argument.

As a concrete type, a list is either empty, and thus represented by the value `NIL`, or a *cell* (or, rather, a *pointer* to a cell) with two fields. One field, the

head, contains the first element of the list. The other field, the *tail*, contains a list of the elements other than the first element of the list.

While the first definition does not mention the physical layout of a non-empty list in memory of a computer, in practice, there is only one reasonable[1] representation, and that reasonable representation corresponds precisely to the second definition. Consequently, this book always treats this type as concrete.

[1] In the era of programming languages (such as Fortran) that did not support the possibility of defining *structures*, it was common to represent all the cells of a list by means of two arrays in parallel. One array contained the *head* fields of all the cells, and the other array contained the *tail* fields of all the cells. The tail field would contain either the index into the arrays of the following cell or 0 for the empty list.