Data abstraction and program development

Reinhold Friedrich Hille

University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/compsciwp

Recommended Citation

https://ro.uow.edu.au/compsciwp/59

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
PART I: ABSTRACT DATA TYPES AND ALGORITHMS

Chapter 1 ALGORITHMS and DATA ABSTRACTION

1.1 Algorithms
1.2 On Programming with Functions
1.3 Data Abstraction
1.4 The Specification of Data Types

Chapter 2 LINEAR DATA TYPES

2.1 The Linear List
   2.1.1 Abstract Definition of the Linear List
   2.1.2 Algorithms for Lists
   2.1.3 Discussing Computational Complexity
   2.1.4 Two List Implementations in Pascal
   2.1.5 Pascal Implementations of some List Algorithms
      2.1.5.1 Chained Implementation
      2.1.5.2 Array Implementation

2.2 The Stack
   2.2.1 A Standard Implementation of the Stack
   2.2.2 Abstract Definition of the Stack
   2.2.3 Functional Implementation of the Stack
   2.2.4 Some Mathematical Properties of Stacks

2.3 The Queue
   2.3.1 Example: Circular Buffer
   2.3.2 Abstract Definition of the Queue
   2.3.3 Functional Implementation of the Queue

2.4 About Removing Recursion
2.5 Maintaining Ordered Lists
   2.5.1 Abstract Definition and Implementation of the Priority Queue
   2.5.2 Efficient Searching, Binary Search
   2.5.3 The Complexity of Binary Search
   2.5.4 Application to Sorting
       2.5.4.1 Insertion Sort
       2.5.4.2 Selection Sort
   2.5.5 Merging Ordered Lists

Chapter 3 NONLINEAR DATA TYPES

3.1 Binary Trees
   3.1.1 Mathematical Background
   3.1.2 Abstract Definition of the Binary Tree
   3.1.3 Tree Traversal Algorithms
   3.1.4 Algorithms using binary Trees
       3.1.4.1 Sorting and Searching
       3.1.4.2 Recognition of Expressions

3.2 Generalised Lists
   3.2.1 Abstract Definition
   3.2.2 List Algorithms
   3.2.3 Representation of other Data Types as Lists
   3.2.4 Dynamic Storage Management

3.3 Types for particular Applications
   3.3.1 The Matrix as an Abstract Data Type
   3.3.2 The Symbol Table as an Abstract Data Type

PART II: DATA REPRESENTATION AND STORAGE

Chapter 4 STORAGE and RETRIEVAL of INFORMATION

4.1 Access to Data
   Comparison of various Methods
   Linear and Binary Search,
   Retrieval via Directories
   One-step Access Methods.

4.2 Address Transformation
   4.2.1 Hash Tables, Hash Functions
   4.2.2 Collision Resolution
   4.2.3 Examples

Chapter 5 FILE ORGANISATION

Secondary Storage Devices and Data Representation

5.1 Linear Files
5.1.1 Sequential Files
5.1.2 Indexed Sequential Files

5.2 Files with Direct Access
5.2.1 B-Trees
5.2.2 Dynamic Hashing
5.2.3 The Grid File
Preface

This book is intended for students of computer science at the second year university level who have completed an introductory programming course and who have also had an introduction to basic machine architecture.

It integrates data abstraction and the use of abstract data types in programming into a course on data structures and algorithms. The use of recursive definition of data types serves as an easy introduction to the concept of recursion and of recursive functions. But more importantly, the extensive use of data abstraction provides a useful guide to the modularisation of programs. Choosing a suitable modularisation is the main problem in top down design or stepwise refinement of programs.

There has been extensive discussion about these issues in the literature. Over the past ten to fifteen years it has become increasingly clear that programming must be liberated as much as that is possible from the burden of implementation detail. By this we mean that low level machine features should be kept out of programming for as long as possible.

Programming should be done in two stages:

(a) The abstract design of data types and algorithms, the actual problem solving stage;

(b) Translation of data types and algorithms into an implementation language.
The first stage begins with the identification of the information which is available and some decision about the operations which will be performed on that information. In other words, programming begins with an abstract view of the data to be used by the program. This is the most important phase of programming, as all decisions about modularisation of the program result from it.

This book demonstrates how the use of abstract data types can lead to the design of programs which are modularised in such a way that they are easy to understand and simple to modify. Jackson stated in his book on program design that the structure of the data should be reflected in the structure of the program. This becomes a natural consequence of the use of abstract data types in program development.

During the past decade much work has been done in the field of data abstraction, and the time has come to put some of this to practical use in everyday program design. The main ideas presented in the first part of the book have their roots in the LISP culture and are also found in some other languages and programming environments such as Alphard, CLU, Simula, Smalltalk.

In the definition of abstract data types we follow the notation used by J.V. Guttag and co-authors [see e.g. Comm ACM 20(1977)396-404] because it has very strong intuitive appeal apart from being mathematically rigorous.

The operations belonging to an abstract data type are defined by a set of axioms which can be formulated in terms of recursion and the alternative construct. Even though recursive functions play a central
role, it should be understood that this does not necessarily require the implementation of all operations as recursive functions. We demonstrate in an informal way how many recursive function definitions can easily be put into iterative form.

The first part of the book gives an introduction into the specification, use, and implementation of some frequently used data types. Algorithms which work with these data types are defined in terms of the primitive operations. The design of data types and algorithms is kept separate from their implementation. However, implementation examples in Pascal are presented in sequence with abstract designs, and in some cases alternative implementations are discussed.

The functional programming style ensures that the number of global variables is kept to an absolute minimum. Avoidance of side effects ensures that higher functions can be defined in very compact form. However, in some applications side effects are used deliberately to simplify the final programs, particularly in applications which deal with the recognition of certain structures stored on secondary devices, i.e. where reading operations become important.

The second part concentrates on implementation issues which result from efficiency considerations, in particular the efficiency of storage and retrieval of data. The issues discussed there are more closely related to machine architecture and the student requires some minimal knowledge of hardware concepts. Such topics as hashing and file organisation are considered there.

The reader is assumed to be familiar with the programming language
Pascal, or at least some similar block structured language, preferably one which supports recursion. Pascal has been chosen as the implementation language because it is now used in introductory programming courses at most tertiary institutions around the world.

It is hoped that the reader will find the book useful both as a guide for the development of a personal programming style and as an aid to understanding some of the important algorithms in computing science.
Chapter 1  ALGORITHMS AND DATA ABSTRACTION

This chapter introduces the main concepts of part I of the book. First it tries to clarify the meaning of the word algorithm. Reasons are given why the concept of a function is important and useful in the definition of abstract data types. Finally, a useful notation for the definition of abstract data types is introduced by using simple examples.

For many years courses on data structures have occupied a central position in the computer science curriculum. With the appearance of volume I of Knuth's book "The Art of Computer Programming" it was widely recognised that the same data structuring techniques can be applied in almost any implementation language. Emphasis was placed on the design of efficient algorithms which were formulated in a mixture of natural language and certain pseudo programming languages.

This approach places heavy emphasis on the procedural aspect of algorithms. The meaning of the data used and manipulated by the algorithms is not described sufficiently. The advantages of using abstract data types were appreciated more widely only since the mid seventies. In many cases the definition of an algorithm loses clarity when its data types are not described or specified in abstract form and only their implementation in terms of machine words or even the data types provided by a programming language like Algol.

One faces a similar problem when trying to define a suitable framework in which to carry out top down design. The point is that a precise definition of an algorithm can be given only when the data on which it
operates have been defined, that means data objects and operations.

The definition of an algorithm is equivalent to the definition of a function in mathematics. The definition of a function consists of the definition of its domain and range and the function rule (which describes how for every point of the domain the corresponding point in the range can be found). This can be done either by specifying a list of pairs of points in domain and range, or as an algorithm which defines the necessary computations. Obviously, it is best to define the rule in terms of the operations characteristic of the data and not, for example, in terms of some machine operations.

In the final analysis, this means that we define a computational model specifically for a problem we are trying to solve. The operations we define for a given set of data types form that model. At the other extreme, we have computational models like the Turing Machine, a theoretical device used in the analysis of the inherent complexity of certain computational problems. For everyday programming, this computational model would be completely impractical.

The first and most important step in programming is the design of the data types and the main algorithms. The concentration on abstract data types and their use as a computational model has distinct advantages which will still manifest themselves in the second stage, the implementation in a particular programming language.

a) The separation of the operational from representational issues results in data independence. The correctness of a program which uses a particular data type is not affected by changes in the
representation of the type.

h) The operations can be defined in rigorous mathematical fashion. Consequently, the implementor knows exactly what it is, that is to be implemented. Other people working on the same project know precisely how to use the data type and the correctness of their part of the program can be proved independently.

c) Because the abstract definition of data types leave out all representational detail, algorithms which are defined in terms of abstract data types are usually much easier to understand.

1.1 Algorithms

In mathematics, the word algorithm has a well defined meaning. It stands for a general procedure which can be used to find a solution for every member of a given class of problems. Once an algorithm has been found, the mathematician is no longer interested in this class of problems and concentrates on others.

The task of the computer programmer overlaps with that of the mathematician. It consists of inventing and formulating the algorithm as well as finding an efficient and useful representation of it in terms of a programming language.

To get an idea of what we mean by an algorithm, consider the following well known examples from mathematics. They have been chosen because the operations of the data types used are well known.
* Long division of two natural numbers which are given in decimal representation;

* Expansion of the square root of a natural number into a decimal fraction;

* Decomposition of a rational function of a real variable into partial fractions;

* Construction of the solution of a set of linear equations with real coefficients.

When we speak of a general procedure for constructing a solution, we mean a procedure in which all steps are defined in every detail. There is no room for creative freedom on the part of the executive.

This imposes strict requirements on the definition of algorithms. It must be possible to write down the entire process as a finite text. We cannot allow, as one might do for example in the description of the algorithm for the resolution of a set of linear equations, that the details of the required arithmetic operations are not defined and must be supplied by the user of the algorithm. Of course, it is clear to the mathematician how to extend the definition to complete the algorithm. In the computing situation such details must be supplied with the algorithm if they are not already defined elsewhere, as for example in the implementation language.

There are terminating and non terminating algorithms. For example, Euclid's algorithm to compute the greatest common divisor of two natural numbers always terminates after a finite number of steps and
delivers a result. On the other hand, the algorithm for the expansion of the square root of a natural number into a decimal fraction does not always terminate because the square root of a number may be irrational.

We are interested mainly in terminating algorithms, although it is often possible to force the termination of an otherwise non-terminating algorithm by introducing an additional condition.

We formulate algorithms as functions. The function rule (text of the algorithm) is composed out of the functions which represent the primitive operations of the data types. The following section is intended to give a brief introduction into programming with functions and some related issues arising from their implementation in a procedural programming language such as Pascal.

1.2 On Programming with Functions

Programming with functions which are free of side effects is also called applicative programming.

A function is said to have a side effect when it alters the value of at least one of its arguments (parameters). In a procedural language such as Algol, Pascal, etc. side effects may be used deliberately to allow functions to return more than one data item resulting from the computation. For procedures this is the only way to return the results of their computations unless they simply use global variables.

Pascal, like all other members of the Algol family of languages, provides two different mechanisms for the passing of parameters to
functions and procedures. Arguments may be passed by value or by
reference. Passing by value means that the parameter behaves like a
local variable of the function. Any change of its value during the
execution of the function is not "visible" to the calling function or
procedure. A variable \( x \) is called local to the function \( f \) if it is
accessible to \( f \) but not to the calling function. Passing by reference
means that no new copy of the variable substituted for the formal
parameter is passed on to the function and the representation of the
variable exists only in the calling routine. In Pascal the parameters
which are passed by reference are called variable parameters. A func-
tion which has only value parameters cannot have a side effect as
described above.

Clearly, side effects can become a source of great difficulty in
instances when existing pieces of software must be altered to suit
different purposes. In the presence of side effects it may be impos-
sible to alter part of a program without being forced to alter much of
the remainder.

We distinguish between two kinds of side effects:

a) Alteration of the values of function arguments. This makes it
impossible to substitute functions as arguments of other functions.
A calling function cannot alter the value to be returned from a
called function. As we shall see later, side effects would also
make it much more difficult to define operations of abstract data
types.
b) Alteration of the values of global variables which are not mentioned explicitly as arguments of functions. This second type of side effect would make it extremely difficult to give formal specifications of functions.

For a procedure a side effect is the only mechanism by which it can return the result of its computation. A function returns its result in the place where it is named (called) in the text of the program. For example

\[ y := 1 + \sin(x); \]

shows how the function \( \sin(x) \) is used inside an expression.

To illustrate the differences between the use of functions and procedures we use the simple problem of selecting the largest of two given integers. We call this operation \( \text{MAX} \) for maximum. Because we must make comparisons to determine which is the larger of the two we assume the existence of the infix operator "\( > \)" for integers, whose value is true if the left operand is greater than the one on the right, otherwise false. As a Pascal procedure, \( \text{MAX} \) would look like this:

```pascal
procedure MAX(i, j: integer; var m: integer);
begin
  if i > j then m := i
  else m := j
end;
```

The parameters \( i \) and \( j \) behave like local variables of \( \text{MAX} \) and any alteration of their values from within \( \text{MAX} \) is not visible to the calling routine. The result of the computation is returned in the variable parameter \( m \). Consequently, \( m \) does not exist as a variable inside
MAX and every reference to it is re-directed to the global variable substituted for \textit{m} in the calling routine.

The more parameters a function has, the more difficult it may become to follow what goes on. Let us investigate the opposite and try to reduce the number of parameters given to MAX. If it is not necessary to preserve the original values of \textit{i} and \textit{j}, then we can get by with two parameters only. The value of the maximum can be assigned to one of them, which must of course be a variable parameter:

\begin{verbatim}
procedure MAX(i : integer; var j : integer);
begin
  if i > j then j := i
end;
\end{verbatim}

Now \textit{MAX} does nothing when the value of \textit{j} is already the maximum, otherwise the value of \textit{j} is replaced by a copy of the value of \textit{i}. It is clear that the information originally contained in the parameters of such a procedure is lost after the execution.

To understand the procedure call

\begin{verbatim}
MAX(x, y);
\end{verbatim}

where \textit{x} and \textit{y} are integer variables, we must be aware of the convention that the result is pre-turned in the second parameter.

If \textit{MAX} is written as a function, the maximum becomes the value of the function after termination. The function call

\begin{verbatim}
m := MAX(x, y);
\end{verbatim}

is unambiguous because it states explicitly where the result appears. The maximum appears in the place at which the function is called.
function MAX(i,j : integer) : integer;
begin
  if i > j then MAX := i
  else MAX := j
end;

By composing MAX with itself we can write a new function MAX3 to find the maximum of three integers:

function MAX3(i,j,k : integer) : integer;
begin
  MAX3 := MAX(MAX(i,j),k)
end;

More complicated functions can be composed out of existing ones. Note that composition of functions is possible only in the absence of side effects.

In terms of procedures The same result can be achieved through two successive procedure calls, but the resulting text is not as easy to understand.

procedure MAX3(i,j,k : integer; var m : integer);
var
  l : integer;
begin
  MAX(i,j,l);
  MAX(l,k,m)
end;

The procedure MAX3 returns the result of its computation in the variable parameter m. The body of MAX3 consists of two calls of the procedure MAX, which can find the maximum of two integers. An intermediate variable l is required to hold the result from the first procedure call,

MAX(i,j,l);

which finds the maximum of i and j and returns the result in l. The
second call of the procedure MAX finds the maximum of 1 and k and returns the result in m, where it becomes available to the calling routine. Note that it is more difficult to understand what goes on here as there are several conventions which are not expressed explicitly. One of them is that the procedures return the result of their computations in the last parameter. Matters are complicated further by the use an intermediate variable. Note that every side effect forces the introduction of a variable to record the effect.

To reduce the number of parameters we can again use the same device as before. If it is not necessary to retain the original values of all parameters then we can simply alter the value of one of them to return the result.

```plaintext
procedure MAX3(i, j : integer; var k : integer);
begin
    MAX(i, j);
    MAX(j, k)
end;
```

To understand what goes on here we must know that MAX(x, y) also returns the maximum in its second parameter, y.

The following two versions of the procedure MAX3 also return the maximum of three integers which are passed to them, whereas the last version is not guaranteed to return the correct result but does so in some cases. Which are they?

```plaintext
procedure MAX3(i, j : integer; var k : integer);
begin
    MAX(i, k);
    MAX(j, k)
end;
```
procedure MAX3(i, j : integer; var k : integer);
begin
   MAX(j, k);
   MAX(i, k)
end;

The following is the incorrect version:

procedure MAX3(i, j : integer; var k : integer);
begin
   MAX(i, k);
   MAX(i, j)
end;

1.3 Data Abstraction

By data abstraction we mean the process of defining abstract data types. One concentrates on the essential properties of the data and omits all implementation detail. An abstract data type consists of a set of domains, one of which is the designated domain (the set of objects of this type), a set of primitive operations defined on these domains, and a set of axioms (rules) which define the primitive operations. The word *primitive* means non-derivative.

To understand what we mean by primitive operations, consider the example of natural numbers. To define the arithmetic operations of addition and multiplication for natural numbers, we require only the two primitive operations PREDecessor(Natno) and SUCCesor(Natno), which we represent as functions. We need two other functions. One of them recognises *zero* which is the first element of the sequence of natural numbers. That is the Boolean function

\[ \text{ISZERO(Natno)} \rightarrow \text{Boolean}, \]

which returns true when it's argument is the number zero. The other
function,

\[
\text{ZERO} \rightarrow \text{Natno},
\]

returns the first element of the sequence of natural numbers.

The sequence of natural numbers \(\{0,1,2,\ldots\}\) is assumed to be given. Therefore, the functions \(\text{PRED}\) and \(\text{SUCC}\) are defined by the existence of the sequence.

We now have four primitives for the data type \text{Natno}:

\[
\begin{align*}
\text{ZERO} & \rightarrow \text{Natno} \\
\text{ISZERO(Natno)} & \rightarrow \text{Boolean} \\
\text{PRED(Natno)} & \rightarrow \text{Natno} \\
\text{SUCC(Natno)} & \rightarrow \text{Natno}
\end{align*}
\]

We have already defined what the domains and co-domains of these functions are. This is the syntactic part of the definition of the data type \text{Natno}. The names of domains always begin with a capital letter and the rest of the name is in small letters. For function names we use capital letters only. After the name of a function we give in parentheses as list of the names of data types which make up the domain of the function. \text{ZERO} is the constant function. It always returns the same result and therefore does not require any argument. The arrow indicates that something is returned by the function which is an element of the data type named as the range of the function.

Without yet writing down the axioms which define the function rules, we want to demonstrate how derived functions can be defined in terms of the primitive functions.
Let us define addition for natural numbers. Note that we must be careful with the definition of the PREdecessor function, as it is not defined for the first natural number.

For the syntactic part of the definition of the function ADD we use the same notation as above. It is:

\[ \text{ADD}(\text{Natno}, \text{Natno}) \rightarrow \text{Natno}. \]

This tells us that the domain of ADD is the set of all pairs of natural numbers. Sometimes this is called the Cartesian product of the set Natno with itself.

Besides the syntactic part of the definition there must be a semantic part in the form of axioms (rules) which define what the function does. To define ADD we must give a rule or a set of rules to describe how the result can be derived from the argument(s).

The problem is trivial if one of the arguments, for example the second, is ZERO. Then the result is simply the value of the first argument. In all other cases we try to reduce the problem to the trivial case by using the fact that the result will be the same if we add \( x \) to the predecessor of \( y \) and take the successor of the result. We continue to make this substitution until the second of the arguments of ADD is ZERO.

For all \( x, y \) in Natno let

\[ \text{ADD}(x, \text{ZERO}) = x \]

\[ \text{ADD}(x, y) = \text{SUCC}(\text{ADD}(x, \text{PRED}(y))). \]
Instead of writing two axioms which deal with the two distinct cases, we can use the alternative construct and write it all into one axiom. This already looks very much like a recursive implementation of the function.

\[
\text{ADD}(x, y) = \\
\quad \begin{cases} 
\text{if ISZERO}(y) \text{ then } x \\
\text{else SUCC}(\text{ADD}(x, \text{PRED}(y)))
\end{cases}
\]

With the aid of addition we can define multiplication for natural numbers. The function

\[
\text{MULT}(\text{Natno}, \text{Natno}) \to \text{Natno}
\]

The product of its two arguments. If the second argument is zero, the result must be zero. Otherwise we must add the first argument to the result of multiplying the first argument by the predecessor of the second argument.

\[
\text{MULT}(x, y) = \\
\quad \begin{cases} 
\text{if ISZERO}(y) \text{ then ZERO} \\
\text{else ADD}(\text{MULT}(x, \text{PRED}(y)), x)
\end{cases}
\]

These two examples demonstrate how algorithms can be defined as derived functions in terms of the primitives of a given data type.

Now we complete the definition of the data type \text{Natno} for which we have the primitive operations

\[
\text{ZERO} \to \text{Natno} \\
\text{ISZERO}(\text{Natno}) \to \text{Boolean} \\
\text{PRED}(\text{Natno}) \to \text{Natno} \\
\text{SUCC}(\text{Natno}) \to \text{Natno}
\]
Since these four functions represent primitive, i.e. non derivative operations, they can be defined only in terms of each other. To the syntactic part already known we add the semantic part as follows:

```plaintext
type Natno
  ZERO \rightarrow Natno
  ISZERO(Natno) \rightarrow Boolean
  PRED(Natno) \rightarrow Natno
  SUCC(Natno) \rightarrow Natno
for all x in Natno
  ISZERO(ZERO) = true
  ISZERO(SUCC(x)) = false
  PRED(SUCC(x)) = x
end Natno
```

The function ZERO defines the starting point of the sequence. For that value the predicate ISZERO must return the value true. This is stated by the first axiom. The second axiom uses the fact that zero cannot be the successor of any other natural number. Note that the last axiom, which defines the way in which PRED and SUCC work together, is written as PRED(SUCC(.)) because that is defined for all natural numbers. SUCC(PRED(.)) is not defined for zero.

1.4 The Specification of Abstract Data Types

Programming consists of two stages:

1. the design of the required data types and the algorithms (functions);

2. the translation of the data types and functions into the implementation language.

The separate treatment of the two stages has the advantage that the
programmer can concentrate on the solution of the problem and the correctness of the algorithms without having to worry about how one might express the solution in terms of a particular programming language with all its special syntax requirements.

In the previous section we gave a first example of a way in which the specification of an abstract data type can be written. In this section we state rules for specifying data types and give a further example.

To implement a data type we must do two things. We must map its domains into the domains provided by the implementation language and we must express the operations of the abstract data type in terms of the operations provided by the implementation language.

But our concern here is the specification of abstract data types for which we require some rules. We shall try to keep these rules as simple as possible without restricting generality. We limit ourselves to the smallest possible number of constructs for writing axioms. It is well known that all computable functions can be defined in terms of alternation and recursion. Therefore, we allow only these two constructs. In the example of the previous section we already saw the use of alternation (if ... then ... else). To be able to compose functions we must forbid side effects.

The axioms which define how the functions work together are written in the form of equations. All occurring variables are only universally qualified.

To summarise, we have the following rules:
1. Side effects are not allowed.

2. All variables are universally qualified.

3. The alternative construct \texttt{if P then S1 else S2} is permitted.

4. Recursion is permitted.

The use of the alternative construct requires predicates which we represent as functions of type \texttt{Boolean}. Consequently, this type is part of the definition of every abstract data type.

The definition of a data type consists of two parts:

(a) the syntactic part which states the names of the domains and ranges of the operations as well as the names of the functions;

(b) the semantic part which consists of the axioms that define the effect of the operations.

At this stage we are not interested to know \textit{how} the functions achieve their results, but rather \textit{what} the results are. The question of \textit{how} is essentially procedural and will be answered in the implementation of the operations. The same data type with all its operations can always be implemented in many different ways. To consider any of these in the abstract definition would only confuse the issue.

\textit{Example}

As an example or a composite type let us define the \texttt{Array} of \texttt{Item}. This example was given by John McCarthy who was instrumental in designing and implementing the functional programming language LISP.
The designated domain is **Array** and the component domain is **Item**.

One can think of arrays as mappings from an index set into a set of values. One usually requires two operations for arrays, to **STO**r\(e\) and to **RE**\(T\)rieve items. The index specifies a particular component of the array. A third operation, **NEW**, returns a new (empty) array.

```plaintext
type Array[Item]
  NEW -> Array
  STO(Item, Array, Index) -> Array
  RET(Array, Index) -> Item

for all \(x\) in Item, \(i, j\) in Index, \(a\) in Array let
  RET(NEW, \(i\)) = UNDEFINED
  RET(STO(\(x, a, i\)), \(j\)) =
    if EQUAL(\(i, j\)) then \(x\)
    else RET(\(a, j\))
end Array
```

The first line names the two domains **Array** and **Item**. The designated domain is **Array**. Although the domain **Index** is part of this data type, it need not be included in this definition. Normally, one uses a subset of the natural numbers for the index, but other data types may also be used, as for example enumeration types in Pascal. We simply assume that the domain **Index** is defined somewhere and that the function **EQUAL** is defined on the Cartesian product of **Index** with itself.

**NEW** is the constant function and requires no argument. It returns an object of type **Array**, which must be the new (empty) array. **STO** requires three arguments: the item to be stored, the array in which storage is to take place, and the index of the position at which storage is to be effected. **STO** returns the updated array. Storage of a value at some place in the array implies that the value previously stored there is lost. **RET** takes two arguments, the array from which
to retrieve and the index. It returns the value last stored at the indicated place.

The semantic part begins with universal quantification of the variables. The first axiom tells us that nothing can be retrieved which has not previously been stored. The detail of UNDEFINED belongs to the implementation. For example, the run time system of Pascal gives an error message when the retrieval of a previously undefined item from an array is attempted. The second axiom specifies how STO and RET work together by defining RET in terms of STO. The operation EQUAL belongs to Index. EQUAL(i,j) = true means that i has the same value as j. The axiom tells us that RET(b,j) must return the value x if that was the last value stored at index j. It also specifies that storage operations elsewhere have no effect on the value to be retrieved from position j.
Chapter 2. LINEAR DATA TYPES

In this chapter we introduce the simplest composite data type, the linear List of items (elements) and two of its special cases, the stack and the queue. We think of a list as a sequence of items because every item has a well-defined place in the list. The special cases of Stack and Queue arise because of restrictions which are placed on the mode of access to the list, i.e., the ways in which items may be added or deleted.

Several algorithms for the manipulation of these data types are introduced and defined as functions composed out of the primitive operations. We give recursive definitions of the data types and also of algorithms (functions) which manipulate the data types. In this way the reader is introduced to recursion in a natural way.

We begin with a simple recursive definition of the list. From the definition we derive a minimal number of primitive operations for lists. Then we define a number of useful algorithms for lists in a notation similar to that used in the list axioms. The algorithms are defined as recursive functions that are easy to understand because their structure resembles very closely the structure of the definition of the list.

In the examples for implementation we first use the fact that programming languages like Pascal support recursion and the functions so defined can easily be implemented as recursive functions.

Finally, there is a section which explains in an informal way how recursion on linear structures like lists can be replaced by
iteration.

2.1 The Linear List

Linear lists of items occur in various forms in many applications. If all items are atomic (i.e. they are not composite themselves) and of the same type, we call the list simple. There are also more complicated lists in which the items may again be lists. We call such lists generalised lists.

2.1.1 Abstract Definition of the Linear List

We base the discussion of this data type on the following definition.

Definition

A linear list is either empty or it consists of an item (the head of the list) and a list (the rest or tail of the list).

Fig. 1

A list consisting of head and tail

The meaning of a data type is defined by its primitive operations. We want to keep the set of primitive operations as small and simple as possible. While it would be tempting to invent more and more different operations which may all be very useful, we seek a minimal set which allows the definition of all other desirable operations as derived
functions.

According to the definition given above the list consists of two components, an item and another list. Hence, we require two access functions, also called selector functions because they select parts of the data object. One of them returns the head of the list, the other the rest. We also require a constructor function to build a list out of the two components.

Let us call these functions HEADL, RESTL, and CONSL. The functions HEADL and RESTL have the domain List, whereas the domain of CONSL is the Cartesian product of Item and List. HEADL is of type Item, RESTL and CONSL are of type List. We express this in the usual way:

\[
\begin{align*}
\text{HEADL(List)} & \rightarrow \text{Item} \\
\text{RESTL(List)} & \rightarrow \text{List} \\
\text{CONSL(Item, List)} & \rightarrow \text{List}
\end{align*}
\]

Let \( i \) be an item and \( l \) be a list. Then the two axioms

\[
\begin{align*}
\text{HEADL(CONSL}(i, l)) & = i \\
\text{RESTL(CONSL}(i, l)) & = l
\end{align*}
\]

tell us everything about the three functions, i.e. how to construct lists and how to take them apart.

This set of primitive functions for lists is not sufficient. For instance, it is not clear what should happen when the argument of
HEADL is the empty list. A predicate is required which can recognize empty lists. We call this function ISNEWL. As already explained in chapter 1 we must have a function which returns the "starting point" of this data type, the empty list. The name of this function is NEWL.

The three domains, List, Item, and Boolean are part of this data type. List itself is the designated domain. Note that the definition of the type Item can be left open. We simply assume that it is defined somewhere else. The definition of the data type List is complete without explicit mention of Item.

We can now understand the complete definition of the data type List[Item].

type List[Item]
  NEWL -> List
  ISNEWL(List) -> Boolean
  HEADL(List) -> Item
  RESTL(List) -> List
  CONSL(Item, List) -> List
for all i in Item, 1 in List let
  ISNEWL(NEWL) = true
  ISNEWL(CONSL(i, 1)) = false
  HEADL(CONSL(i, 1)) = 1
  RESTL(CONSL(i, 1)) = 1
end List

The first two axioms tell us that ISNEWL must return the value true for new (empty) lists, otherwise false. For the head of an empty list nothing meaningful can be returned. Therefore, there is no axiom to define HEADL(NEWL).

The situation is different in the case of the RESTL. Its range is List. There appear to be two choices.
(1) We could leave RESTL undefined for empty lists

(2) or we could argue that the rest of an empty list must be the empty list again.

It is preferable to leave RESTL(NEWL) undefined rather than to introduce a special convention for this case. With additional rules to cover special cases one always runs the risk of introducing inconsistencies or contradictions elsewhere. However, in this case the risk is minimal and in fact, this solution is used in the programming language LISP.

We adopt the first of the two choices as we see no need to rely on the additional rule in any of the list algorithms of this chapter.

The last two axioms tell us how the functions CONSL, HEADL, and RESTL work together. When a list has been constructed out of the item i and the list l, the value of HEADL for that list must be the item i, and the value of RESTL must be l.

2.1.2 Algorithms for Lists

Before we show how lists can be implemented let us define some algorithms for lists based on these primitive operations.

We can define algorithms for an abstract data type without referring to an actual implementation because we know exactly how the primitive operations work together. This is one of the benefits of data abstraction. One can write complex programs which use the abstract data types and discuss their correctness even before the data types them-
selves have been implemented.

The implementation of the data types of such a program can be altered without affecting the program itself. That means changes can be localised and the maintenance of such software is comparatively easy.

There are many well known algorithms for linear lists which are used frequently in applications, sometimes explicitly, sometimes as part of larger segments of code. Here we give some instructive examples.

The definitions of the functions as given below should be considered to be definitions of their behaviour only, although it is quite feasible to implement them exactly how they are defined as recursive functions. However, this is by no means necessary as we shall see in the section on removal of recursion. It is our purpose to discuss how such functions can be defined in a way that does not rely on a particular programming language.

LAST

HEADL returns the first item contained in a list. It may be necessary to access the item at the other end of a list. LAST returns the last item contained in the list which is its argument. As in the case of HEADL nothing meaningful can be returned if the argument is the empty list. LAST can be defined only for lists which contain at least one item. Consequently, there are only two cases to be considered:

(1) The list contains one item only, i.e. the rest of the list is empty and the head is also the last item.
(2) The list contains more than one item. The last item in the list is also the last item in the rest of the list.

The definition of a function contains two parts just like the definition of an abstract data type. In the syntactic part we simply state the names of the function and its domain and range. The semantic part is written in terms of the alternative construct and recursion, using the primitive operations and possibly some other derived functions. Primitive operations of subsidiary data types may also be used.

LAST(List) -> Item
LAST(1) =
  if ISNEWL(RESTL(1)) then HEADL(1)
  else LAST(RESTL(1))

If the list contains only one item, that is the one to be returned by LAST. This case is recognised by the fact that the rest is the empty list.

If on the other hand the rest of the list is not empty, then we make progress by asking for the last item in the rest of the list. Eventually the recursion will lead to a list of one item only.

Note that we would be in trouble if we allowed the rest of an empty list to be defined. In that case we might try to return the head of an empty list. Since LAST is not defined for empty lists, any functions using it must ensure that LAST is not called with the empty list as argument. In other words, it is the responsibility of the calling function to guard against that.
LENGTH

The problem is to determine the length of a list of items, i.e. the number of items it contains. To count the number we require an additional data type, Natural Number, for which we assume the existence of the functions ZERO, SUCC, as defined in the first chapter.

We distinguish between the following cases;

(1) The empty list has length zero.

(2) In all other cases the length of the given list is greater by one than (the successor of) the length of the REST.

LEN(List) \rightarrow \text{Natno}

LEN(1) =
    if ISNEWL(1) then ZERO
    else SUCC(LEN(RESTL(1)))

We always write the termination case first. Obviously, the length of an empty list is zero, which is the value returned by the function ZERO. When the length of the given list 1 is greater than zero, we simply take the successor of the length of the rest of the list which we must determine first. Hence the recursive call of LEN on the rest of the list. Eventually the list which is passed to recursive calls of LEN will be empty as each of the recursive calls receives only the rest of the list passed to at the previous recursive call. Then all recursive calls will terminate by returning the appropriate values to their calling functions, and the first instantiation of the function LEN will return a natural number which gives the length of the list.
Given an item \( x \) and a list \( l \), the problem is to determine whether \( x \) is contained in \( l \). In the worst case, namely when \( x \) is not in the list, all items will be inspected. If the item is present, the search can stop as soon as it has been found. We must distinguish three cases:

1. the list is empty, ISIN returns false;
2. \( x \) is equal to HEADL(1), ISIN returns true;
3. \( x \) is not equal to HEADL(1) and the list is not empty, the search must be continued in the rest of the list.

For the data type Item we require the operation EQUAL. All other operations come from the type List.

\[
\text{ISIN(Item, List) } \rightarrow \text{ Boolean}
\]

\[
\text{ISIN}(x, 1) = \\
\text{ if ISNEWL}(1) \text{ then false} \\
\text{ else if EQUAL}(x, \text{HEADL}(1)) \text{ then true} \\
\text{ else ISIN}(x, \text{RESTL}(1))
\]

**MAXIMUM**

A slightly different problem is to find the largest element in a given list. We give the name MAX to the function. The maximum of an empty list is undefined. Hence we have only two cases to consider here:

1. There is only one item in the list. That item is the maximum.
2. There is more than one item. MAX must return either the head of the list or the maximum of the rest of the list, whichever is
greater. For this purpose we use the auxiliary function GR which returns the greater of two items.

\[
\text{MAX(List) } \rightarrow \text{Item}
\]

\[
\text{MAX(1) = if ISNEWL(RESTL(1)) then HEADL(1) else GR(HEADL(1), MAX(RESTL(1)))}
\]

The function GR(Item, Item) is defined as follows:

\[
\text{GR(Item, Item) } \rightarrow \text{Item}
\]

\[
\text{GR(x, y) } = \begin{cases} 
\text{x} & \text{if } x > y \\
\text{y} & \text{else}
\end{cases}
\]

All other operations are list primitives. First, MAX tests whether l consists of one item only. That must be the maximum. Otherwise the list contains more items and the head must be compared with the maximum of the rest.

PUTLAST

This function inserts an additional item at the end of a list. The domain of PUTLAST is the Cartesian product of Item and List. The range is the data type List.

Case Analysis:

(1) If the list is empty it is easy to insert a new last item because that will also be the first item. Hence, this case can be handled by CONSL.
In all other cases we make progress by attempting to insert into the rest of the list. Afterwards we must make sure that the head of the list is replaced.

\[
\text{PUTLAST(Item, List) } \rightarrow \text{ List}
\]

\[
\text{PUTLAST}(i, l) =
\begin{cases}
  \text{CONSL}(i, l) & \text{if ISNEWL}(l) \text{ then}
  \\
  \text{CONSL(HEADL}(l), \text{PUTLAST}(i, \text{RESTL}(l))) & \text{else}
\end{cases}
\]

APPEND

PUTLAST placed an additional item at the end of a given list. APPEND attaches another list at the end of a given list.

Given two lists \(x\) and \(y\), the function APPEND must return a list which contains all elements of \(x\) in their original sequence followed by all elements of \(y\), also in their original sequence. Let us consider all possible cases.

1 \(x\) is empty:

1.1 \(y\) is empty: the value of APPEND\((x, y)\) is NEWL

1.2 \(y\) is not empty: the value to return is \(y\)

2 \(x\) is not empty:

2.1 \(y\) is empty: the value to return is \(x\)

2.2 \(y\) is not empty: we must try to append REST\((x)\) and \(y\) and afterwards replace the head of \(x\).

This translates easily into
All algorithms must then be formulated as if they were to be performed on such a machine. Rather than translating everything down to the level of zeros and ones we shall use a model which is defined at a much higher level. We are justified in doing so, because a particular operation or computational steps of such a high level model can usually be carried out in a fixed number of low level steps on a Turing Machine or, for that matter, on a real machine. We just have to be careful that the size of the problem does not have any effect on the complexity of the operations which we assume to be "basic".

We shall assume the existence of an abstract machine which can execute the primitive operations of the abstract data type List. In terms of this model we shall discuss the computational complexity of the function REVERSE.

The complexity of an algorithm is expressed as a function of the size of the problem. Let us call this function \( f(n) \), where \( n \) is a measure of the size of the problem. If we are dealing with lists, then the size is given by the number of elements in the list.

Complexity theory is interested only in the asymptotic behaviour of the function \( f(n) \), i.e. how fast it grows as \( n \) becomes large. One compares the growth of \( f(n) \) with that of well known types of functions by forming quotients and determining whether limits exist for large \( n \).

If \( f(n) \) grows like a quadratic polynomial in \( n \), then the quotient of \( f(n) \) over the square of \( n \) remains constant as \( n \) becomes very large. We say that the complexity is of the order of \( n \) squared, which is written \( O(n^2) \). This notation is sometimes called the "big oh" notation. The number of steps is given by a constant times \( n \) squared. That constant
may be large, but the important point is that it is a constant and the actual size is largely immaterial.

The Complexity of REVERSE

Let us assume that the function REVERSE is actually implemented just as it is defined it in the previous section, i.e. as a recursive function which uses another recursive function, PUTLAST.

\[
\text{REVERSE(} \text{List} \text{)} \rightarrow \text{List}
\]

\[
\text{REVERSE(} l \text{)} =
\]

if ISNEWL(\text{I}) \text{ then } 1
else PUTLAST(HEADL(\text{I}), \text{REVERSE(RESTL(\text{I}))})

To discuss the complexity of REVERSE we simply count the number of times the function CONSL is executed in one invocation of REVERSE on a list of length n. This is a simplification, and the actual complexity differs from ours by a constant factor because each recursive invocation of PUTLAST makes calls to ISNEWL, HEADL, and RESTL, except the last one, which calls only ISNEWL and CONSL. However, as far as the asymptotic behaviour is concerned, we still obtain the same result but with a different constant.

REVERSE calls itself recursively n times if the length of the list is n. That involves n calls of the function PUTLAST. Consequently, PUTLAST is invoked in turn on lists of length \(n-1\), \(n-2\), \(\ldots\), 2, 1, 0, in that order.

PUTLAST uses CONSL \(k+1\) times on a list of length \(k\). Hence, there will be \(n + (n-1) + \ldots + 2 + 1\) applications of CONSL. The sum of this arithmetic progression comes to \(n(n+1)/2\). That is the number of
computational steps of REVERSE as defined on a list of length n.

The complexity REVERSE is \( f(n) = \frac{n(n+1)}{2} \). As \( n \) becomes large the quadratic term will dominate the first order term of \( f(n) \). The complexity of REVERSE in this form is \( O(n^2) \), the square of \( n \).

Now we define a new function REVl which can do the job in \( O(n) \). To do this we introduce a device known in functional programming as an accumulating parameter. The meaning will become clear from the definition of the function. The idea is as follows. REVl takes the list \( l \) apart, one item at a time, always taking the head off the remaining list. Each item is then inserted at the head of the accumulating parameter which is another list. Consequently, the new list contains the items in reverse order.

\[
\text{REVl}(\text{List}, \text{List}) \rightarrow \text{List}
\]

\[
\text{REVl}(a, l) = \begin{cases} 
\text{if ISNEWL}(l) \text{ then } a \\
\text{else REVl(CONSL(HEALD(l), a), RESTL(l))}
\end{cases}
\]

The list \( a \) is the accumulating parameter. When REVl terminates, \( a \) contains the elements of \( l \) in reverse order in front of any items it may have contained initially. Each recursive call of REVl is made with one more item removed from the list \( l \) and placed onto the front of the list \( a \), which can be done by one call of CONSL. Hence, the complexity of this function is \( O(n) \).

Now we can re-define REVERSE as

\[
\text{REVERSE}(l) = \text{REVl(NEW, l)},
\]
in order to hide the accumulating parameter which will initially be
the empty list.

2.1.4 Two List Implementations in Pascal

The implementation details of the list primitives are determined by
the representation of the domains. The choice may be dictated by the
application or it may simply be a matter of taste. We can represent a
list as a chain of records which are linked together with pointers.
This allows us to use Pascal's dynamic storage allocation. We obtain
the storage space for a list node only when we actually want to insert
an item. Reference to the list is made by pointing to the first
record. The items are stored in records. The records are often called
list nodes.

A list node must not be confused with an item. They are two different
things. A node contains an item. An object consisting of one list node
is a list and not an item.

The necessary type declarations in Pascal are:

```pascal
type
  itemtype = (* to be filled in *);
  listtype = ^node;
  node = record
    item : itemtype;
    rest : listtype
  end;
```

The type declarations are the mapping of the two domains List and Item
into the data types of Pascal. The definition of `itemtype` is left open
because items can be defined independently elsewhere. A node, imple-
mented as a Pascal record, contains an item and a pointer to the rest of the list. The empty list is represented by the nil pointer. A non-empty list is represented by a pointer to the first record.

**Implementation of the Functions**

The function **NEWL** simply returns the nil pointer:

```pascal
function NEWL : listtype;
begin
  NEWL := nil
end;
```

The task of **ISNEWL** is to decide whether l has that value:

```pascal
function ISNEWL(l : listtype) : listtype;
begin
  ISNEWL := (l = nil)
end;
```

Once the function **CONSL** has been implemented, the functions **HEADL** and **RESTL** follow, because in the axioms they are defined in terms of **CONSL**. **CONSL** accepts an item and a list and produces a list which consists of these two components. To do this, it must obtain a new list node from the dynamic storage allocation system. The additional item will be stored in the new node. After that the new record is linked to the front of l. **CONSL** returns the pointer to the new node, and thereby the entire list.
function CONSL(i : itemtype; l : listtype) : listtype;
var
tmp : listtype;
begin
  new(tmp);
  tmp^.item := i;
  tmp^.rest := l;
  CONSL := tmp
end;

For the implementation of HEADL and RESTL recall the axioms:

\[
\begin{align*}
\text{HEADL}(&\text{CONSL}(i, l)) = i \\
\text{RESTL}(&\text{CONSL}(i, l)) = l
\end{align*}
\]

The implementation of CONSL above and these two axioms define how
HEADL and RESTL must be implemented.

function HEADL(l : listtype) : listtype;
begin
  HEADL := l^.item
end;

Note that HEADL is not defined for empty lists, as implied by the list
axioms. However, the function HEADL(CONSL(x,y)) has the value x.

RESTL simply alters the list pointer to point to the second node of
its argument.

function RESTL(l : listtype) : listtype;
begin
  RESTL := l^.next
end;

Note that this function fails if its argument is the empty list. See
also the discussion of the list axioms.
An alternative Implementation

Sometimes it may be more convenient to implement a list as an array of items. Because arrays are not dynamic in Pascal, every list requires the same amount of storage space in the machine.

We must ensure that the array is always large enough, so that there will be no overflow problems. As an additional safeguard one might want to introduce additional tests to ensure that there is enough room left when an item is to be inserted. This would introduce a bounded data type requiring additional primitive operations and axioms to define them.

It is usually impossible to decide upon design of an abstract data type what action should take place when an overflow occurs. Overflow is a catastrophic situation and would require dramatic action such as re-compilation of the program to allow for more storage space. In any case, the responsibility for taking corrective action lies with the program that uses the data type and not with the data type itself. Therefore, features such as error recovery are best left out of the definition of abstract data types.

Even though the first list implementation uses dynamic storage allocation it’s advantage over the array implementation in this respect is not as great as it might seem at first sight. After all, dynamic storage allocation still must operate within the bounds of a finite machine.

In the array implementation of the list it is useful to keep track of the number of items in the list. We require two variables to represent
the list, the array of items and an integer variable to represent the counter. They can be built into a record which represents the list so that one can refer to the list by using one variable only.

In this implementation the function NEWL must deliver such a record complete with array and counter. The function ISNEWL(List) will test the value of the counter in its argument.

If we simply stored the list elements beginning at position one and always adding a new item at that position, then we would have to move all items by one place each time another one is added to the front of the list. To avoid this we might turn the list around so that the last item is always at the first position. Then the head of a list of length n is at the n-th position.

The following constant and type declarations in Pascal are required.

```pascal
const
  maxlen = (* something large enough *);

type
  itemtype = (* left open *);
  listtype = listrec;
  listrec = record
    items : array[1..maxlen] of itemtype;
    count : integer (* see the explanation *)
  end;
```

The function NEWL requires a local variable to hold the value of the pointer to the new record it must obtain through the Pascal procedure `new`. 
function NEWL : listtype;
var
tmp : listtype;
begin
  new(tmp);
  tmp^.count := 0;
  NEWL := tmp
end;

function ISNEWL(l : listtype) : boolean;
begin
  ISNEWL := l^.count = 0
end;

CONSL simply stores the additional item i in the given list l by
incrementing count and storing i at that location.

function CONSL(i : itemtype; l : listtype) : listtype;
begin
  with l^ do begin
    count := count + 1;
    items[count] := i
  end;
  CONSL := l
end;

The implementations of HEADL and RESTL are clear from the foregoing.

function HEADL(l : listtype) : itemtype;
begin
  HEADL := l^.items[count]
end;

function RESTL(l : listtype) : listtype;
begin
  with l^ do
    if count > 0 then
      count := count - 1;
  RESTL := l
end;

Strictly speaking, RESTL is a deletion operation. If the original
value of the argument 1 should not be destroyed by RESTL(1), it must make a copy of its argument without the first item and not just alter the value of 1. This is important, for example, when one wants to define derived functions in which the same argument is passed to RESTL more than once. As RESTL is written above, it has the side effect of altering its list argument. The same consideration applies to CONSL.

In the next chapter we shall use side effects deliberately to simplify the definitions of algorithms for handling arithmetic expressions.

In the first implementation of the list as a chain of records, RESTL and CONSL did not alter their arguments. The original lists are still available after the execution of these functions. RESTL does not actually delete an item. It simply sets a pointer to the successor of the head. Similarly, CONSL does not alter the value of the pointer to its list argument. It only links an additional record onto the front of the list and returns a pointer to that.

2.1.5 Pascal Implementations of some List Algorithms

This section is in preparation.
Implementation examples of several of the list algorithms will be given together with a discussion of some issues related to efficiency.

It is now a simple matter to implement the list algorithms of section
2.1.2 as Pascal functions by simply substituting from either of these two implementations of the list. However, some of the implementations would turn out to be rather clumsy and wasteful. For example, the function LEN would only have to read the counter in the second implementation to find the length of a list instead of taking the list apart to find out how many elements it contained.

Similarly, the function LAST would only have to read the item at the first array position, which is always the last item in a non-empty list in this implementation.

But for lists implemented as chains of records we can actually implement the algorithms as they are defined in terms of the primitives without too much loss of efficiency.

2.1.5.1 Chained Implementation

We are given a list \( l \) of items and the task is to replace the first occurrence of the item \( x \) by the item \( y \). If \( x \) is not in the list, no action is taken.

\[
\text{REPLACE}(\text{Item}, \text{Item}, \text{List}) \rightarrow \text{List}
\]

\[
\text{REPLACE}(x, y, l) =
\begin{align*}
& \text{if ISNEWL}(l) \text{ then } l \\
& \text{else if EQUAL}(x, \text{HEADL}(l)) \text{ then } \text{CONSL}(y, \text{RESTL}(l)) \\
& \text{else } \text{CONSL}(\text{HEADL}(l), \text{REPLACE}(x, y, \text{RESTL}(l)))
\end{align*}
\]

If the list is empty, there is nothing to be done. If the head is equal to \( x \) then it is replaced by \( y \), otherwise we make progress by trying to replace \( x \) by \( y \) in the rest of the list before we put the head back on.
As a Pascal function REPLACE looks like this:

function REPLACE(x, y : itemtype; l : listtype) : listtype;
begin
  if ISNEWL(l) then REPLACE := l
  else if EQUAL(x, HEADL(l)) then REPLACE := CONSL(y, RESTL(l))
  else REPLACE := CONSL(HEADL(l), REPLACE(x, y, RESTL(l)))
end;

There will be several examples here
====================================

2.1.5.2 Array Implementation

There will be several examples here
====================================

Exercises for this section in preparation.
2.2 The Stack

The access to the data type List[Item] as defined in the previous section is not restricted in any way. In principle, access is possible to any part of the list even though we did not define primitive operations to retrieve any item other than the head of the list. This must not be seen as a limitation on the modes of access to list elements. Different combinations of the functions HEADL and RESTL are sufficient to access any part of the list.

Whether a data object belongs to the type Stack or not is determined by the way in which it is used and not so much by the way in which it is defined. As we shall see, a stack of items is really much the same as a list of items. The difference is in the limitations which are imposed on its uses.

It is possible to store, retrieve and delete items. The rules are that the item last stored is the first to be retrieved or deleted. A stack is a "LIFO" store, where LIFO stands for "Last In First Out". Access to other items below the top of the stack is not permitted. A storage device is a stack only because of this restriction. We think of a stack as a collection of items which are placed on top of each other so that only the topmost item can be reached at any time and that additional items must be placed on top.

2.2.1 A Standard Implementation

The most frequently used implementation of a stack uses the access operations PUSH and POP, where PUSH(Item, Stack) means to place an
additional item on top of the stack. POP usually does two things and is therefore not a single valued function. It alters the value of the stack by deleting the topmost item. It also returns the value of that item. If POP(Item, Stack) is implemented as a procedure, it must alter the values of both its arguments. PUSH(Item, Stack) \rightarrow Stack can be implemented as a function without side effect, returning an updated copy of the stack. Quite often, both of them are implemented as procedures.

The store of the stack can be implemented as an array of items. When the index of the array is declared to range from zero to size, it is usual to let the index top (often called the stack pointer) always refer to the first free position in the array just above the item last added to the stack. That item is at the position top-1. The stack pointer is incremented after an insertion. After a deletion (decrement of the stack pointer) the topmost item is still available at array[stacktop]. The state empty is equivalent to top = 0, whereas full corresponds to top = size.

This is only a convention and the index might as well range from 1 to size. Then the stackpointer top could refer to the position of the item last stored. We have chosen the first variant, in which top refers to the next available location. Of course, top = 0 indicates the empty stack, whereas top = size+1 means the stack is full.

In Pascal, the necessary declarations could be as follows:
const  
size = ...; (* according to requirements *)

type  
itemtype = ...; (* depending on the application *)
stacktype = array[1..size] of itemtype;

var  
s : stacktype;
top : integer; (* one above the topmost item *)

The procedures PUSH and POP can be implemented as

procedure PUSH(x : itemtype; var stack : stacktype;
               var stacktop : integer);
begin
  stack[stacktop] := x;
  stacktop := stacktop + 1
end;

procedure POP(var x : itemtype; var stack : stacktype;
              var stacktop : integer);
begin
  stacktop := stacktop - 1;
  x := stack[stacktop]
end;

PUSH does not test whether the stack is already full before it goes ahead and attempts to insert the next item. Similarly, POP does not test whether the stack is empty before attempting to delete. We believe that the responsibility for this lies with the program that uses the stack. Also, POP has the responsibility of returning the item that was on top. The result of popping an empty stack cannot be defined.

The implementation is somewhat clumsy because it requires two parameters, stack and stacktop, to refer to one stack. PUSH must alter stacktop and the array. Its first parameter, x, can be a value parameter, but the second, the array, is altered from within the procedure.
POP produces two results, a new value for x and a new value for stack-top. It is not necessary for POP to actually delete an item from the array, it just alters the pointer so that the topmost item is no longer available. The array is passed to POP as a variable parameter because Pascal would make a copy of it if it were passed as a value parameter.

PUSH returns an updated stack. The result in this implementation is that stacktop must be altered apart from entering the new item in the array.

When written as a function, PUSH returns the new value of stacktop. The function POP returns the value of the topmost item which, as a side effect it also removes from the stack.

PUSH must update the array and stacktop.

```pascal
function PUSH(x : itemtype; stacktop : integer;
             var stack : stacktype) : integer;
begin
  stack[stacktop] := x;
  PUSH := stacktop + 1
end;
```

```pascal
function POP(var stacktop : integer;
             var stack : stacktype) : itemtype;
begin
  stacktop := stacktop - 1;
  POP := stack[stacktop]
end;
```

If we want to write stack operations as pure functions without side effect, we must split POP into two operations. Functions must be single valued, i.e. they can return only one result. One of the two new functions could return the topmost item. We call it
TOP(Stack) -> Item.

The other alters the stack by deleting the topmost Item. It can be called

POP(Stack) -> Stack.

There is no problem with PUSH, whose syntax can be

PUSH(Item, Stack) -> Stack.

The absence of side effects from these functions allows us to define the abstract data type Stack[Item].

2.2.2 Abstract Definition of the Stack

We can give a recursive definition of a stack which is very similar to that of a list.

Definition

A stack is either empty or it is composed of an item and a stack.

As was the case with lists, we require five operations, one to make a new stack, one to recognise a new stack, one to construct a stack out of it’s two components, and two selectors for the two components. The operations are:

NEWSTACK to return a new (empty) stack,
ISNEWSTACK(Stack) to test whether the stack is empty,
PUSH(Item, Stack) to add an Item to the stack,
TOP(Stack) to retrieve the item last stored,
POP(Stack) to delete the item last stored.

The resulting formal specification of the type Stack is:
type Stack[Item]
    NEWSTACK -> Stack
    ISNEWSTACK(Stack) -> Boolean
    PUSH(Item, Stack) -> Stack
    TOP(Stack) -> Item
    POP(Stack) -> Stack

for all i in Item, s in Stack let
    ISNEWSTACK(NEWSTACK) = true
    ISNEWSTACK(PUSH(i, s)) = false
    TOP(NEWSTACK) = {UNDEFINED}
    TOP(PUSH(i, s)) = i
    POP(NEWSTACK) = NEWSTACK
    POP(PUSH(i, s)) = s

end Stack

This is the same as the definition of the data type List of Item. We just have to rename TOP as HEADL, POP as RESTL, and PUSH as CONSL. Therefore, we need not write new implementations for the stack when we already have the implementation of a list. We just rename the operations. From the implementation of the list is as a chain of linked records we obtain directly the same implementation for the stack.

The reason for this is that the stack is just a linear list with restricted access. There are no operations for it beyond the ones provided as primitives, for if we had further access functions to be able to retrieve items from further down, then this data type would no longer be a stack.

2.2.3 Functional Implementation of the Stack

In the example at the beginning of this section we implemented the stack as an array of items. The difficulty there was that we could not pass the stack to the functions as one argument. We had to pass the array as well as the stack pointer. To return an updated stack we had to return the updated array as well as the new value of the stack.
To overcome this the array and the stack pointer can be built into a Pascal record.

In Pascal problem will arise with functions whose task it is to return a stack. They are:

\[
\begin{align*}
\text{NEWSTACK} & \rightarrow \text{Stack} \\
\text{PUSH(Item, Stack)} & \rightarrow \text{Stack} \\
\text{POP(Stack)} & \rightarrow \text{Stack}
\end{align*}
\]

In Pascal, functions are not supposed to return structured data types, only scalar types. For this reason we decided to declare the type Stack as a pointer to a record containing the array and the integer Stacktop.

The declarations are:

```pascal
type
  itemtype = (* left open *)
  stacktype = 'stackrec;
  stackrec = record
    store: array[1..size] of itemtype;
    top: integer (* stacktop *)
  end;
```

NEWSTACK must return an empty stack. It returns a pointer to the new record.

```pascal
function NEWSTACK : stacktype;
var
  result: stacktype;
begin
  new(result);
  result^.top := 0;
  NEWSTACK := result
end:
```
ISNEWSTACK simply has to test whether the stackpointer contained in it's argument is zero.

```pascal
function ISNEWSTACK(st : stacktype) : boolean;
begin
  ISNEWSTACK := (st. top = 0)
end;
```

PUSH must store the value of it's first argument (Item) in the array contained in it's second argument (Stack). It returns a pointer to the updated stack.

```pascal
function PUSH(i : itemtype; st : stacktype) : stacktype;
begin
  with st do begin
    store[top] := i;
    top := top + 1
  end;
  PUSH := st
end;
```

TOP and POP together simply reverse the effect of PUSH.

```pascal
function TOP(st : stacktype) : itemtype;
begin
  with st do
    TOP := store[top]
end;
```

```pascal
function POP(st : stacktype) : stacktype;
begin
  with st do
    top := top - 1;
    POP := st
end;
```

Because of the use of pointers these functions actually alter the values of their arguments. However, again because of the use of pointers, it is still possible to compose the functions as if there
were no side effects. Therefore, the possibility of defining most
algorithms as derived functions in terms of the primitives has been
retained.

We also have the possibility to test the correctness of this implemen-
tation directly by substituting the functions into the axioms for the
data type Stack. For this purpose we have written the following short
main program. If the functions are implemented correctly, then for
each axiom the boolean value true should be returned.

(* Assume the following variable declarations *)

var
  item : itemtype;
  stack1, stack2 : stacktype;

begin (* main program *)
  item := ••• ; (* something appropriate *)
  stack1 := POP(PUSH(item, NEWSTACK));
  stack2 := NEWSTACK;
  writeln(stack1~ = stack2~);
  stack1 := ••• ; (* some non-empty stack *)
  stack2~ := stack1~;
  stack1 := POP(PUSH(item, stack1));
  writeln(stack1~ = stack2~);

Since we want to compare the values of the records, we cannot write
the axiom

POP(PUSH(item, stack)) = stack

directly as a Pascal statement. That would only verify that we are
pointing to the same record. We must compare the values of two
records, which were equal before the composite function POP(PUSH( ... )) was applied to one of them.
2.2.4 Some Mathematical Properties of Stacks

With a stack one can generate the members of a class of permutations in the following way. From an input stream of items, one item at a time is pushed onto the stack. At any time, before the next action there are two choices:

(1) remove the item on top of the stack and add it to the output stream,

(2) push the next item onto the stack.

We must only ensure that no attempt is made to POP an empty stack. Each element is pushed and popped exactly once. Therefore, the number of operations PUSH must equal the number of operations POP. The sequence of operations PUSH and POP must be such that at no time the number of POPs exceeds the number of PUSHes.

Such a sequence of operations is called well-formed.

To answer the question of how many permutations of the sequence of numbers \{1, 2, ..., N\} can be generated with a stack, we must count the number of well-formed sequences of 2N operations PUSH and POP.

Let us denote PUSH by 1 and POP by 0. Thus \{110100\} is well-formed, but \{10010110\} is not. The second sequence becomes not well-formed for the first time at the position of the third element, where the number of zeros exceeds the number of ones by one.

Instead of counting the stack permutations directly, we can make use of the one-to-one relationship which exists between the well-formed
sequences and the stack permutations.

For \( N = 3 \) there are five different well-formed sequences, and therefore five of the six permutations of three elements can be generated with a stack. It is easy to verify that the sequences and the corresponding permutations are:

\[
\begin{align*}
\{101010\} & \{1, 2, 3\} \\
\{101100\} & \{1, 3, 2\} \\
\{110010\} & \{2, 1, 3\} \\
\{110100\} & \{2, 3, 1\} \\
\{111000\} & \{3, 2, 1\}
\end{align*}
\]

Apparently, it is impossible to generate the permutation \( \{3, 1, 2\} \) with a stack. That is so because we would have to move the third element of the input sequence ahead of the other two without also reversing their order.

In general, stack permutations \( \{p_1, p_2, \ldots, p_N\} \) do not contain subsequences of three elements \( p_i, p_j, p_k \) such that \( p_j < p_i \) and \( p_k > p_i \).

Since each of the permutations is generated by a different well-formed sequence of operations, it is sufficient to count the well-formed sequences. To do that we use the device of first counting all sequences of \( N \) ones and \( N \) zeros. Then we subtract from that the number of non-well-formed sequences of \( N \) ones and \( N \) zeros. They are easier to count than the well-formed sequences.

Assume that we have \( N \) symbols "1" and \( 2N \) slots in which to place them. The gaps will later be filled with the zeros. That can be done in \( \binom{2N}{N} \) different ways because we select \( N \) of a total of \( 2N \) slots.
Here, \( C(i, j) \) is the binomial coefficient \( \frac{i!}{j! (i-j)!} \).

The number of sequences of \( N \) ones and \( N \) zeros, which are not well-formed is the same as the number of sequences of \( N-1 \) ones and \( N+1 \) zeros by the following argument. Take any non well-formed sequence of \( N \) ones and \( N \) zeros. In it mark the index \( j \) at which for the first time the number of zeros exceeds the number of ones. From that point onwards, that is, for all indices \( i > j \), re-write the sequence by replacing ones by zeros and vice versa. That turns the sequence into one containing \( N-1 \) ones and \( N+1 \) zeros.

This mapping can be inverted. Take any sequence of \( N-1 \) ones and \( N+1 \) zeros. There will always be an index \( j \) at which the number of zeros exceeds the number of ones by one for the first time, reading from left to right. For all indices \( i > j \) again reverse the symbols, thereby obtaining a non well-formed sequence of \( N \) ones and \( N \) zeros. Hence, the mapping is one-to-one and onto. That concludes the argument.

It is now the simple matter to count these sequences in the same way as before. The result is \( C(2N, N-1) \). Therefore, the total number of stack permutations of a sequence of \( N \) elements is

\[
C(2N, N) - C(2N, N-1) = \frac{C(2N, N)}{1+N}.
\]

As we shall see in the next chapter, this is also the number of binary trees of \( N \) nodes.

2.2.5 An Application: Postfix Evaluator
The definition of the postfix expression will be given here together with the algorithm using a stack.

Exercises for this section in preparation.
2.3 The Queue

The data type queue occurs very often in information processing. For example, reading from a sequential file is the same as working through the contents of a queue. A buffer between a producer and a consumer process is a queue. The text of a Pascal program appears to a compiler as a queue of symbols of the language.

Many algorithms make explicit use of queues, which means that their definitions rely on the definition of a queue. We shall see some examples in the next chapter when we define the traversal algorithms for trees.

The queue is a storage device from which the item inserted first is always released first. A queue is sometimes called a FIFO (First In, First Out) store.

It is customary to call the item due to be released the head of the queue. The remainder is still a queue and is called the tail or rest of the queue.

2.3.1 Example: The Circular Buffer

We begin with a particular implementation example of a queue. One of the most frequent uses of a queue is the buffer. Buffers are required as communication devices between producer and consumer processes. The producer process adds data items to the buffer (the end of the queue). It can do so as long as the buffer is not full. The consumer process takes data from the buffer, thereby consuming the contents, and can do so as long as the buffer is not empty. Only one of the processes can
have access to the buffer at a time. Switching between the two processes may take place at any time, not only when the buffer is either full or empty. Therefore, it is essential to maintain information about the locations of the two ends of the queue.

The buffer may be represented as an array of items. Data items are stored contiguously (without gaps). Because the buffer is not always full, it is necessary to use two indices to mark the beginning and the end of the occupied region. Because insertions take place at the rear and deletions at the front of the queue, the occupied region will migrate through the array. This would make it necessary to move the buffer contents forward when the end of the array is reached. To avoid this costly relocation of data items, we allow the queue to wrap around the array. That is, the rear of the queue may be ahead of the front as shown in the diagram below. The index front always points to the location immediately ahead of the queue, whereas the index rear points at the location containing the last item in the queue. This arrangement allows distinction between the states empty and full without the using an additional variable.

Diagrams depicting states of a ring buffer.

After some insertions and deletions a state may be reached in which rear < front. The end of the queue has wrapped around to the front. The buffer may be in one of three different states: empty, partially full, or full. In the empty state both indices are the same. The
head of the queue is deleted simply by increasing the front index by one. However, before a deletion can be carried out, a check must be made whether front = rear. If that is the case, the queue is already empty and the deletion cannot go ahead.

After front has been incremented (modulo array size), the head item of the queue is still available at array[front].

The number of items in the queue is the same as the difference between the two indices, modulo array size.

When the queue is full, front = (rear + 1) mod array size. On insertion we must first ensure that the queue is not full. Therefore, we first increment the value of the rear index modulo array size. If after that rear = front, the queue is already full and the intended insertion cannot go ahead. The original value of the rear index must be restored.

For the Pascal implementation we require the following declarations.

```pascal
const
  size = ... ; (* depending on the application *)
var
  buf: array[0..size] of item; (* storage area *)
  front, rear: integer; (* to mark the data region *)
```

The operations INSERT and DELETE can be written as Pascal procedures as follows.
procedure INSERT(item : itemtype; front : integer;
    var rear : integer;
    var buffer : buffertype);

var
    save : integer; (* to save the original value of rear *)
begin
    save := rear;
    rear := (rear + 1) mod (size + 1);
    if not (rear = front) then
        buffer[rear] := item
    else begin
        rear := save;
        EXCEPTION(full)
    end
end;

The variables front and item can be passed to INSERT by value. Rear
and buffer must be variable parameters as they are updated from within
the procedure.

procedure DELETE(item : itemtype;
    var front : integer;
    rear : integer);
begin
    if front = rear then
        EXCEPTION(empty)
    else
        front := (front + 1) mod (size + 1)
end;

"EXCEPTION" can be a procedure call to deal with exceptions as they
arise. Details depend on the actual application and cannot be decided
without knowledge about it.

The buffer is passed as a variable parameter to the procedure INSERT
so that at the procedure call the array is not copied. Front must be
a variable parameter, because it is altered from within the procedure.
The else part "EXCEPTION" is not strictly required because a deletion
from an empty buffer can be deemed not to have taken place at all and
the conditional statement guards the action against the state "empty".

The "trick" of this implementation is that the last available place in the array is never filled. This allows distinction between the states full and empty at the time when insertions or deletions are attempted. If we were to use all places in the array, this distinction would no longer be possible without an additional variable. For example, DELETE finds front = rear. This could have come about by filling the array completely. Hence, DELETE must find out whether the queue is empty or full. The necessary information can be maintained in a boolean variable.

2.3.2 Abstract Definition of the Queue

The implementation given above was obviously intended for efficient operation of a ring buffer. It is evident that the details of this implementation were dictated by the properties of the machine for which it was made. In such cases there is the danger that the characteristic properties of the data type are obscured to some extent by the measures taken "to make life easy for the machine". For example, the wrap-around feature has nothing to do with with the intrinsic properties of a queue. The essence of the queue lies in the access to it.

There is a logical inconsistency built into this implementation. The item just deleted is still available at array[front]. There is no operation to retrieve the head of the queue. We already encountered a similar situation in the case of the stack, where we decided to separate POP(Stack) and TOP(Stack). Here we shall again have two
operations HEADQ(Queue) to retrieve the head and DELQ(Queue) to delete the head of the queue. There will be five operations in all.

ADDQ(Item, Queue) -> Queue, to add an item to a queue,

HEADQ(Queue) -> Item, to return the head of the queue,

DELQ(Queue) -> Queue, to delete the head of the queue,

NEWQ -> Queue, to make a new queue,

ISNEWQ(Queue) -> Boolean, to test whether a queue is empty.

The names describe the operations. This is simply for readability. The axioms which specify the queue do not rely on any naming scheme.

We write down the complete definition before discussing it.

type Queue[Item]
    NEWQ -> A Queue
    ISNEWQ(Queue) -> A Boolean
    ADDQ(Item, Queue) -> A Queue
    HEADQ(Queue) -> A Item
    DELQ(Queue) -> A Queue
for all i in Item, q in Queue let
    ISNEWQ(NEWQ) = true
    ISNEWQ(ADDQ(i, q)) = false
    HEADQ(ADDQ(i, q)) =
        if ISNEWQ(q) then i
        else HEADQ(q)
    DELQ(ADDQ(i, q)) =
        if ISNEWQ(q) then q
        else ADDQ(i, DELQ(q))
end Queue

The semantic part of the definition must state that the head is that element of all those still in the queue which was the first to be inserted and will therefore be the first to be deleted. If the queue contains only one element, as would be the case when the previous
insertion was into an empty queue, then the head is the first and last element at the same time. In all other cases the value of HEADQ is not affected by the last insertion.

Similarly, DELQ is defined to remove always that item of all those still in the queue which was added first. If q was empty before the last insertion, then we simply restore q by DELQ(ADDQ(i, q)). If on the other hand the queue was not empty before the last insertion, then the order of ADDQ and DELQ can be reversed without affecting the result, because additions and deletions take place at opposite ends.

2.3.3 Functional Implementation of the Queue

Clearly, the implementation of the operations depends on the chosen representation of the data object. In the first section of this chapter we gave the definition and a Pascal implementation of the data type List[Item]. Instead of implementing the queue in terms of the predefined data types of Pascal, we shall write an implementation in terms of the data type List.

We find that the functions NEWL, ISNEWL, HEADL, and RESTL can be used directly to represent the queue functions NEWQ, ISNEWQ, HEADQ, and DELQ. The only difference is in the function ADDQ, which we can represent by PUTLAST.

From our understanding of these data types we already know that this is correct. But the definition of data types with the use of axioms allows us to prove the correctness of this implementation in a particularly simple formal way. We just have to show that the axioms for
the queue are satisfied when we replace the queue functions by appropriate list functions that may be primitive or derived functions.

To do this we write down the axioms for the queue and substitute for the queue functions the corresponding list functions. The list axioms must then be used to show that the queue axioms are satisfied.

We define:

\[
\begin{align*}
\text{NEWQ} &= \text{NEWL}, \\
\text{ISNEWQ}(\text{Queue}) &= \text{ISNEWL}(\text{List}), \\
\text{HEADQ}(\text{Queue}) &= \text{HEADL}(\text{List}), \\
\text{DELQ}(\text{Queue}) &= \text{RESTL}(\text{List}), \\
\text{ADDQ}(&\text{Item, Queue}) = \text{PUTLAST}(\text{Item, List}).
\end{align*}
\]

The axioms

\[
\begin{align*}
\text{ISNEWQ}(&\text{NEWQ}) = \text{true} \\
\text{ISNEWQ}(&\text{ADDQ}(i, q)) = \text{false}
\end{align*}
\]

are satisfied immediately when the corresponding list axioms are satisfied and when we use

\[
\text{ADDQ}(i, \text{NEWQ}) = \text{CONSL}(i, \text{NEWL}).
\]

Now we prove

\[
\begin{align*}
\text{HEADQ}(&\text{ADDQ}(i, q)) = \text{if ISNEWQ}(q) \text{ then } i \\
&\text{else } \text{HEADQ}(q).
\end{align*}
\]

Substitution of the list functions yields

\[
\begin{align*}
\text{HEADL}(&\text{PUTLAST}(i, 1)) = \text{if ISNEWL}(1) \text{ then } i \\
&\text{else } \text{HEADL}(1).
\end{align*}
\]
insertion was into an empty queue, then the head is the first and last element at the same time. In all other cases the value of HEADQ is not affected by the last insertion.

Similarly, DELQ is defined to remove always that item of all those still in the queue which was added first. If q was empty before the last insertion, then we simply restore q by DELQ(ADDQ(i, q)). If on the other hand the queue was not empty before the last insertion, then the order of ADDQ and DELQ can be reversed without affecting the result, because additions and deletions take place at opposite ends.

2.3.3 Functional Implementation of the Queue

Clearly, the implementation of the operations depends on the chosen representation of the data object. In the first section of this chapter we gave the definition and a Pascal implementation of the data type List[Item]. Instead of implementing the queue in terms of the predefined data types of Pascal, we shall write an implementation in terms of the data type List.

We find that the functions NEWL, ISNEWL, HEADL, and RESTL can be used directly to represent the queue functions NEWQ, ISNEWQ, HEADQ, and DELQ. The only difference is in the function ADDQ, which we can represent by PUTLAST.

From our understanding of these data types we already know that this is correct. But the definition of data types with the use of axioms allows us to prove the correctness of this implementation in a particularly simple formal way. We just have to show that the axioms for
the queue are satisfied when we replace the queue functions by appropriate list functions that may be primitive or derived functions.

To do this we write down the axioms for the queue and substitute for the queue functions the corresponding list functions. The list axioms must then be used to show that the queue axioms are satisfied.

We define:

\[
\begin{align*}
\text{NEWQ} &= \text{NEWL}, \\
\text{ISNEWQ}(\text{Queue}) &= \text{ISNEWL}(\text{List}), \\
\text{HEADQ}(\text{Queue}) &= \text{HEADL}(\text{List}), \\
\text{DELQ}(\text{Queue}) &= \text{RESTL}(\text{List}), \\
\text{ADDQ}(\text{Item, Queue}) &= \text{PUTLAST}(\text{Item, List}).
\end{align*}
\]

The axioms

\[
\begin{align*}
\text{ISNEWQ}(\text{NEWQ}) &= \text{true} \\
\text{ISNEWQ}(\text{ADDQ}(i, q)) &= \text{false}
\end{align*}
\]

are satisfied immediately when the corresponding list axioms are satisfied and when we use

\[
\text{ADDQ}(i, \text{NEWQ}) = \text{CONSL}(i, \text{NEWL}).
\]

Now we prove

\[
\text{HEADQ}(\text{ADDQ}(i, q)) = \begin{cases} 
\text{if ISNEWQ}(q) \text{ then } i \\
\text{else HEADQ}(q).
\end{cases}
\]

Substitution of the list functions yields

\[
\text{HEADL}(\text{PUTLAST}(i, l)) = \begin{cases} 
\text{if ISNEWL}(l) \text{ then } i \\
\text{else HEADL}(l).
\end{cases}
\]
We treat the two cases separately. First:

\[ \text{HEADL(PUTLAST(i, NEWL))} = i. \]

We substitute from the definition of PUTLAST:

\[ \text{HEADL(CONSL(i, NEWL))} = i, \]

which is satisfied because of the definition if the list. Now we must prove the general case:

\[ \text{HEADL(PUTLAST(i, 1))} = \text{HEADL(1)}. \]

When 1 is not the empty list, we have

\[ \text{PUTLAST(i, 1)} = \text{CONSL(HEADL(1), PUTLAST(i, RESTL(1))} \]

and substitution into the line above gives

\[ \text{HEADL(CONSL(HEADL(1), PUTLAST(i, RESTL(1))}} = \text{HEADL(1)}. \]

Because of HEADL(CONSL(x, y)) = x, the left hand side reduces to HEADL(1).

That completes this proof.

Next, we must show that

\[ \text{DELQ(ADDQ(i, q))} = \text{if ISNEWQ(q) then NEWQ else ADDQ(i, DELQ(q))} \]

Again, we treat the two cases separately. First the empty queue:

\[ \text{DELQ(ADDQ(i, NEWQ))} = \text{NEWQ}, \]
where we substitute the list functions and obtain

\[ \text{RESTL}(\text{PUTLAST}(i, \text{NEWL})) = \text{NEWL}. \]

By the definition of \text{PUTLAST} this becomes

\[ \text{RESTL}(\text{CONSL}(i, \text{NEWL})) = \text{NEWL}, \]

in accordance with the list axioms.

Now the general case:

\[ \text{RESTL}(\text{PUTLAST}(i, q)) = \text{PUTLAST}(i, \text{RESTL}(q)). \]

When \( q \) is not empty, we substitute

\[ \text{PUTLAST}(i, q) = \text{CONSL}(\text{HEADL}(q), \text{PUTLAST}(i, \text{RESTL}(q)) \]

into the left hand side and obtain

\[ \text{RESTL}(\text{CONSL}(\text{HEADL}(q), \text{PUTLAST}(i, \text{RESTL}(q))). \]

Because \( \text{RESTL}(\text{CONSL}(x, y)) = y \), this reduces to

\[ \text{PUTLAST}(i, \text{RESTL}(q)) \]

as was to be proved.

Thus, we have an implementation of the queue as a list, which we have proved correct in terms of the axioms defining the two data types. The Pascal version can be obtained by substitution of the Pascal versions of the list functions together with the representation of the data objects used there.
Additional Queue Functions

Some authors include in the set of queue primitives the operation to append two queues, which is frequently required. We take the view that this is not strictly a primitive operation because it is easily defined in terms of the primitives which we already have.

In the section on lists we described a function to append two lists:

\[
\text{APPEND(List, List) } \rightarrow \text{ List.}
\]

A similar function for queues can be constructed out of the queue primitives.

\[
\text{APPENDQ(Queue, Queue) } \rightarrow \text{ Queue.}
\]

The central idea is the same. In

\[
\text{APPENDQ}(x, y)
\]

we work through the second argument, \(y\), always adding the head of \(y\) to \(x\), and continue with the rest of \(y\). That is, we treat \(x\) as an accumulating parameter. That gives the following definition of the function \(\text{APPENDQ}\):

\[
\text{APPENDQ}(x, y) = \begin{cases} 
\text{if ISNEWQ}(y) & \text{then } x \\
\text{else APPENDQ(ADDQ(HEADQ(y), x), DELQ(y))} 
\end{cases}
\]

In the next section on the maintenance of ordered lists we shall see a variant of the queue found in some scheduling applications, the priority queue. There, not the first item, but the item of highest priority is deleted first from the queue.
Exercises for this section in preparation.
2.4 On Removing Linear Recursion

Programming languages which support recursive function calls use a stack. The stack is also used for ordinary function or procedure calls. When the execution of a function or procedure begins, a stack frame containing the parameters and local variables as well as a reference to global variables and some other "house keeping" information is pushed onto the stack. When the function terminates the last frame is removed from the top of the stack to expose the frame which belongs to the calling function whose execution will then continue.

The obvious way to remove recursion would be to simulate the runtime stack and do all the necessary book keeping in the function itself. However, in certain simple cases one can get away without a stack. Simple List functions are linearly recursive because the data type list is linear itself. A linearly recursive function is called tail end recursive if the last action taken by the function before it terminates is a recursive call of itself. In that case it is not necessary to use an additional stack frame for the recursive call and the frame belonging to the first invocation of the function can be re used for all recursive calls. We shall see that it is not difficult to write a tail end recursive function in iterative form by using a loop.

There are several reasons why one might want to remove recursion. The implementation language may not support recursion or one might want to take advantage of the fact that the iterative version runs faster than the recursive one.

The function Length of section 2.1.2 was defined as:
LEN(list) -> Natno

LEN(1) =
    if ISNEWL(1) then ZERO
    else SUCC(LEN(RESTL(1)))

Before evaluation of a function can begin, the values of all arguments must be determined. In this case the recursive call of LEN appears as the parameter of another function. Therefore, the recursive call is not the last action taken by the function. LEN is not truly tail end recursive. In the else part of LEN, RESTL is evaluated first, then LEN, and finally SUCC.

Let us unfold some of the recursive steps. The last action taken by the recursive function is to find the successor of the result of the recursive call LEN(RESTL(1)). When we unfold one step we find:

SUCC(SUCC(LEN(RESTL(RESTL(1)))))

Eventually, on termination of the recursive scheme, RESTL will return the empty list and we have

SUCC(SUCC(SUCC( ... LEN(NEWL) ... )))

Because we are now defining an iterative function, we must maintain an auxiliary variable to contain the intermediate result which will be updated each time around the loop. We call it result. The while loop is appropriate here because the test for the empty list must be made before any action can be taken. The action of computing the successor of an intermediate result is carried out once for each recursive call. Recursive calls are made as long as the list argument of the function is not the empty list.
Initially, result is zero. The while loop must run as long as the end (the empty list) has not been reached. That means, the condition of the while loop must be the negation of the termination condition of the recursive function. At each step we update result by taking its successor and we update the list by taking the rest.

\[
\text{LEN}(l) = \\
\begin{cases} 
\text{result} \leftarrow \text{ZERO} \\
\text{while not ISNEW}(l) \\
\quad \text{result} \leftarrow \text{SUCC}(\text{result}) \\
\quad l \leftarrow \text{RESTL}(l) \\
\text{return}(\text{result})
\end{cases}
\]

In the semantic part we express the updating of variables (assignment) by the symbol "\leftarrow". Because we are writing in iterative form, we must be able to group statements together to form compound statements. All statements enclosed by a matching pair of braces belong to a compound statement. The body of the function LEN is a compound statement, so is the body of the while loop.

LEN uses the type Natno with its primitives ZERO and SUCC for the counting. The while loop is not executed at all when l is the empty list. If the list contains N items, then the loop is executed N times as N items must be discarded to make the list empty. This will be the value of count which is returned in the end.

Maximum

In the section on lists we defined a function MAX to return the largest item contained in a simple list.
MAX(List) -> Item

MAX(1) =
    if ISNEWL(RESTL(1)) then HEADL(1)
    else GR(HEADL(1), MAX(RESTL(1)))

If we unfold one recursive step, we obtain

GR(HEADL(1),
    GR(HEADL(RESTL(1)),
        MAX(RESTL(RESTL(1)))
    )
).

If we number the items in the list as 1, 2, ..., we can write this as

GR(I, GR(I, GR(I, ..., HEADL(RESTL( ... (RESTL(1)) ...) ... 1 2 3
, ... ,
where HEADL(RESTL(RESTL( ... (RESTL(1)) ...) is simply 1, the last element of the list. We have

GR(I, GR(I, GR(I, ..., GR(I, I) ... )))
1 2 3 N-1 N

Instead of selecting the greater of the last two elements, then selecting the greater of that and the one before, and so on, we can begin with the first element and replace it only when the next element (head of the next sublist) is greater. The loop must be executed as long as the remaining list contains more than one element.

Here is the text of the iterative form of MAX:
Instead of using RESTL(l) at each step, we can update l to RESTL(l) once before entering the loop for the first time, after having saved HEADL(l) in result.

```
MAX(l) =
    {result <- HEADL(l)
     l <- RESTL(l)
     while not ISNEWL(l) {
       result <- GR(result, HEADL(RESTL(l)))
       l <- RESTL(l)
     }
     return(result)
```

This reduces by two the number of function calls per loop.

Exercises for this section in preparation.
2.5 Maintaining Ordered Lists

There are two main reasons for keeping the items of a list in some sort of lexicographical order:

(a) so that an ordered listing or report can be printed that is easy to read for people,

(b) to allow efficient searching in applications that involve frequent searches of large lists.

The domain Item in the data type List[Item] must be a well-ordered set. That means a relation is defined on the set of items, which allows us to decide which one of two different given items "comes before the other" or "is smaller".

If a frequently occurring operation is to find the largest (or smallest) element contained in a list, then this list may be implemented as a priority queue.

A priority queue is simply an ordered list with restricted access. It is a storage device in which the item of highest priority is always the next to be deleted or retrieved. If the priority queue is implemented as an ordered linear list, then finding the item with the highest priority is particularly simple because it is at the head of the list.

For the priority queue we require functions to return the maximum, to delete the maximum, and to insert an additional item. We give them the names MIN, DELMAX, and INSERT.
If the priority queue is implemented as a list in descending order, the function to find the maximum is simply

\[ \text{MAX(Prqueue)} \rightarrow \text{Item} \]

\[ \text{MAX}(p) = \text{HEADL}(p). \]

The function

\[ \text{INSERT(Item, Prqueue)} \rightarrow \text{Prqueue} \]

guarantees that the items in the queue are always in lexical order by inserting the new item in the correct place. It requires the predicate

\[ \text{ISGREATER(Item, Item)} \rightarrow \text{Boolean.} \]

\[ \text{ISGREATER}(x, y) = \begin{cases} 
\text{true} & \text{if } x > y \\
\text{false} & \text{otherwise}
\end{cases} \]

The definition of the function \text{INSERT} for priority queues follows from the following case analysis.

1. Finding the correct place for an additional item is easiest when either the list is empty or when the new item is greater than the maximum of the list. In both cases the insertion is done by \text{CONSL(Item, List)}.

2. In all other cases we must insert into the rest of the list and then replace the head.

\[ \text{INSERT(Item, Orderedlist)} \rightarrow \text{Orderedlist} \]

\[ \text{INSERT}(i, l) = \begin{cases} 
\text{CONSL}(i, l) & \text{if } \text{ISNEWL}(l) \\
\text{CONSL}(\text{MAX}(l), \text{INSERT}(i, \text{RESTL}(l))) & \text{else if } \text{ISGREATER}(i, \text{MAX}(l)) \\
\text{CONSL}(\text{MAX}(l), \text{INSERT}(i, \text{RESTL}(l))) & \text{else}
\end{cases} \]
The first two cases can be joined:

\[
\text{INSERT}(i, 1) = \begin{cases} 
\text{CONSL}(i, 1) & \text{if ISNEWL}(l) \text{ or ISGREATER}(i, \text{MAX}(l)) \\
\text{CONSL}(&\text{MAX}(l), \text{INSERT}(i, \text{RESTL}(l))) & \text{else}
\end{cases}
\]

2.5.1 Abstract Definition of the Priority Queue

The previous section describes the implementation of the priority queue as a simple list. INSERT was written as a derived function. In this section we give two different abstract definitions of the priority queue. The first one defines the priority queue by itself, whereas the second defines it as a list with an additional (derived) function to insert additional items.

From the foregoing discussion it is clear that we require five operations to maintain a priority queue:

- NEWPQ: return an empty ordered list,
- ISNEWPQ: test whether a given ordered list is empty,
- ADDPQ: insert an additional item,
- MAXPQ: return the greatest item in the ordered list,
- DELPQ: delete the greatest item in the ordered list.

The essence of the priority queue lies in the three functions ADDPQ, MAXPQ, and DELPQ. DELPQ reverses the action of ADDPQ only when the item to be inserted is greater than the maximum already in the list. Otherwise, provided that the list is not originally empty, the order
of the two operations can be reversed, i.e.

$$\text{ADDPQ}(i, \text{DELPQ}(p)) = \text{DELPQ}(\text{ADDPQ}(i, p)),$$

where \(i\) is an item and \(p\) is a priority queue.

If the priority queue is empty, we cannot delete before inserting, and
the rule above no longer holds.

Note that \(\text{DELPQ}\) and \(\text{ADDPQ}\) cannot be inverses of each other, because
the range of \(\text{DELPQ}\) is not the same as the domain of \(\text{ADDPQ}\), which takes
two arguments, the new element and the list.

The function \(\text{MAXPQ}\) must return the item \(i\) last added to the priority
queue by \(\text{ADDPQ}\) only if \(i\) is greater than the maximum prior to the last
insertion. In all other cases, \(\text{MAXPQ}\) must return whatever was the max-
imum before the last insertion. In

$$\text{MAXPQ}(\text{ADDPQ}(i, p)) = \ldots$$

the case of the empty queue \(p\) is special and must be treated
separately because \(\text{MAXPQ}(\text{NEW})\) is undefined.

Here is the complete definition of the priority queue.
type Prqueue[Item]

NEW -> Prqueue
ISNEWPQ(Prqueue) -> Boolean
MAXPQ(Prqueue) -> Item
DELPQ(Prqueue) -> Prqueue
ADDPQ(Item, Prqueue) -> Prqueue

for all i in Item, p in Prqueue let
ISNEWPQ(NEW) = true
ISNEWPQ(ADDPQ(i, p)) = false
MAXPQ(NEW) = {UNDEFINED}
MAXPQ(ADDPQ(i, p)) = if ISNEWPQ(p) then i
else if ISGREATER(i, MAXPQ(p)) then i
else MAXPQ(p)
DELPQ(ADDPQ(i, p)) = if ISNEWL(p) then NEW
else if ISGREATER(i, MAXPQ(p))
then else ADDPQ(i, DELPQ(p))

end Prqueue.

In section 2.3.3 we demonstrated how the queue can be implemented in terms of the abstract data type List[Item]. Below we give an implementation of the priority queue as a simple list. We require only one additional operation:

ADDPQ(Item, Prqueue) -> Prqueue

which we can define as a derived function for the list. The list is maintained such that the first item is always the maximum. The remaining functions can be implemented as follows:

NEWPQ = NEWL
ISNEWPQ(Prqueue) = ISNEWL(List)
MAXPQ(Prqueue) = HEADL(List)
DELPQ(Prqueue) = RESTL(List)

The function ADDPQ is defined as
for all i in Item, p in Prqueue let
ADDPQ(i, p) = if ISNEWL(p) then CONS(i, p)
else if ISLESS(i, MAXPQ(p)) then CONS(i, p)
else CONS(MAXPQ(p), ADDPQ(i, DEl.(p)))

The correctness proof of this implementation is left as an exercise.

This algorithm for ADDPQ is based on a linear search of the list to find the correct place for insertion. Consequently, it may take \( N \) steps of computation in the worst case to insert an additional item.

If it is very important to be able to insert efficiently, then the priority queue might for example be implemented as a binary tree. This has the advantage over a linear representation that insertions require a number of computational steps which grows logarithmically rather than linearly with the size of the queue, provided that the tree does not degenerate into a linear list. In fact, the priority queue implemented as a binary tree will require re-balancing from time to time.

2.5.2 Efficient Searching, Binary Search

One of the main reasons for maintaining an ordered list is to be able to carry out an efficient search of the list, either by a human operator or by machine. The simple linear search is appropriate in situations in which the list to be searched is not too long. The average search time is proportional to the length of the list. However, when the list to be searched is very long, the time required for a linear search may be prohibitive and one might have to look for more efficient search techniques. In applications in which the contents of the
list are seldom changed, that is, searches are predominant, the binary search technique can be employed.

For example, a telephone book in which the names of subscribers to the telephone service are not listed in alphabetical order would require a linear search which could take a very long time indeed. A person using a telephone book can find a name quickly by the binary search procedure. One takes an initial guess about the position of the name in the book and checks the name found at that position for its alphabetical order relative to the name sought. On the basis of that information, one makes a second guess, and so on, until either the name has been found or it has become clear that the name is not contained in the list.

This search procedure relies on a particular representation of the ordered list which allows direct access to any indicated place in order to decide in which half of the list to continue the search. The array is such a structure because access to items in arrays is by index.

Because the search region is always divided in two, the search is called "binary". The search may terminate when either the item in question has been found or the search region has become too small to be subdivided further. Alternatively, one may always allow the search to continue until a search region of minimal size has been found. On termination the indices for the lower and upper bounds of the region will both point to the item if it has been found in the list or they will point to the place where the item should be if it were in the list.
Below we define the binary search recursively for lists without assuming a particular representation. Besides the arguments "Item" and "List" we require a third argument to define an offset for the position in the list, should the search continue in the upper half.

**Case Analysis:**

1. If the length of the list is one, return the offset;

2. Find X, the value of the item half way through the list;
   
   2.1 If Item < X then search in the front part of the list
   
   2.2 else search in the rear part;

BINARYSEARCH(Item, List, Integer) -> Integer

BINARYSEARCH(i, 1, p) =
if LENGTH(1) = 1 then p + 1
else if i > ITEM(LENGTH(1) div 2, 1) then
   BINARYSEARCH(i, REAR(1, LENGTH(1) div 2), p + LENGTH(1) div 2)
else
   BINARYSEARCH(i, FRONT(1, LENGTH(1) div 2), p)

This uses the auxiliary functions LENGTH(List) as previously defined, FRONT(List, Integer), and REAR(List, Integer). FRONT(1, p) returns the first part of the list 1 up to and including the item at position p, REAR(1, p) returns the last part of the list 1 beginning with the item at position p + 1.

For the example of an efficient implementation in Pascal we represent an ordered list of integers as an array. Instead of writing a recursive version, we use a while loop which is executed as long as the length of the list is greater than one, i.e. the index "lower" is less
than the index "upper". We require the following declarations:

```pascal
const
size = ...; (* depending on the application *)
type
orderedlist = array[1..size] of integer;
var
1 : orderedlist;
```

We write the binary search as a function which accepts two arguments, the item for which to search and the list in which to search. It returns the index of the position at which the item was found or at which it would be if it were in the list.

The indices of the lower and upper bounds of the search region are maintained in the variables lower and upper.

```pascal
(* Binary Search, first version *)
function BINARYSEARCH(key : integer;
var 1 : orderedlist) : integer;
var
lower, middle, upper : integer;
begin
lower := 1;
upper := size;
while lower < upper do begin
  middle := (lower + upper) div 2;
  if key > 1[middle] then
    lower := middle + 1
  else
    upper := middle
end;
BINARYSEARCH := lower
end;
```

In the computation of the index middle we truncate the result of the division. Therefore, the test on the next line must be whether key is greater than 1[middle], in which case the lower bound is set to one
place above the middle. Otherwise it may be in the region lower to middle, including the boundaries. Hence, upper := middle. The truncation of the division result together with the assignment of lower to middle + 1 if the key is greater than 1[middle] assures that on termination both indices (lower and upper) point to the same spot. It would be wrong to write the while loop as

```python
while lower < upper do begin
    middle := (lower + upper) div 2;
    if key < 1[middle] then upper := middle - 1
    else lower := middle
end;
```

because the search would not terminate properly. At first sight this seems to be correct, but the following counter example shows the flaw.

The sequence {4, 7} is stored in an array between positions lower = 1 and upper = 2. We are searching for the key 7. The first time around the loop we compute middle = (1 + 2) div 2 = 1. The key is not less than 1[middle] and we set lower = middle. Hence, lower remains at 1 and the loop does not terminate.

2.5.3 The Complexity of the Binary Search

In this implementation of the binary search two tests are carried out each time around the loop. The loop condition is tested and the test inside the loop is carried out.

It may happen that the key is found very early on and still the binary search pattern continues until the two indices meet. One might try to avoid this apparent waste and terminate the search as soon as the key
is found. That required an additional test inside the loop for equality between the key and the entry \(l[\text{middle}]\). The loop terminates when lower is no longer less than upper. Therefore, we set lower = middle and upper = middle as soon as the key has been found.

(* Binary Search, second version *)

function BINARYSEARCH(key : integer;
var 1 : orderedlist) : integer;
var
lower, middle, upper : integer;
begin
lower := 1;
upper := size;
while lower < upper do begin
    middle := (lower + upper) \div 2;
    if key = \(l[\text{middle}]\) then begin
        lower := middle;
        upper := middle
    end
    else if key > \(l[\text{middle}]\) then
        lower := middle + 1
    else
        upper := middle
end;
BINARYSEARCH := lower
end;

In this implementation three tests are made each time around the loop, and it turns out that the search is somewhat more expensive in the average case.

If there are \(N\) items in the list then it may be halved \(x\) times, where \(x\) is the smallest integer greater than \(\log N\). Hence, the first version always makes \(2 \log N\) comparisons. As there is a 50% chance to find an before the last subdivision, the second version takes on the average \(3 (\log N - 1)\) comparisons. As \(N\) becomes large the first version turns out to be more efficient.
2.5.4 Application to Sorting

This section deals with the problem of transforming an initially unsorted list into a sorted one. There are many algorithms which do this, some are more efficient than others. The time taken by the simpler algorithms (e.g. insertion sort and selection sort) grows quadratically with the length of the list, whereas the time taken by the faster sorting algorithms grows with the size of the list like \( N \log N \), where \( N \) is the length.

We begin by defining two simple sorting algorithms, insertion sort and selection sort.

2.5.4.1 Insertion Sort

Because we already have the definition of a function to insert items into an ordered list, it is very easy to define a function which sorts a given sequence of items by that method. We shall again use the idea of an accumulating parameter as introduced for the list function REVERSE. The accumulating parameter will become the sorted list.

At each step we must remove the head from the given unsorted list and insert it at the right place into the accumulating parameter. If the given list is empty, we can return the value of the accumulating parameter.

\[
\text{SORTI}(\text{List}, \text{List}) \rightarrow \text{List}
\]

\[
\text{SORTI}(a, l) = \begin{cases} 
\text{if ISNEWL}(l) \text{ then } a \\
\text{else } \text{SORTI} \left( \text{INSERT}(\text{HEADL}(l), a), \text{RESTL}(l) \right) 
\end{cases}
\]
We hide the accumulating parameter \( a \) by defining \( \text{INSERTIONSORT} \) as

\[
\text{INSERTIONSORT}(l) = \text{SORT1}(\text{NEWL}, l)
\]

### 2.5.4.2 Selection Sort

In similar fashion we can define the selection sort if we have operations \( \text{MAX} \) and \( \text{DELMAX} \), which find and delete, respectively, the largest element of a list, and whose syntax is:

\[
\begin{align*}
\text{MAX}(\text{List}) & \rightarrow \text{Item} \\
\text{DELMAX}(\text{List}) & \rightarrow \text{List} \\
\text{SELSORT}(\text{List}) & \rightarrow \text{List}
\end{align*}
\]

\( \text{SELSORT} \) uses \( \text{MAX} \) to select the largest element of the given unsorted list. \( \text{CONSL} \) can place it at the head if an accumulating parameter. \( \text{DELMAX} \) deletes it from the unsorted list. Then all other elements still in the unsorted list are smaller and must consequently be added to the front of the accumulating parameter. For \( \text{MAX} \) we can use the function defined in section 2.1.2.

First we define a function \( \text{SELSORT1} \) with an accumulating parameter \( a \), whose value can be returned when the given unsorted list \( l \) is empty.

\[
\text{SELSORT1}(a, l) = \begin{cases} 
\text{if ISNEWL}(l) \text{ then } a \\
\text{else } \text{SELSORT1} \left( \text{CONSL} \left( \text{MAX}(l), a \right), \text{DELMAX}(l) \right).
\end{cases}
\]

Finally, we hide the accumulating parameter as before by defining
In the chapter on non-linear data types we shall see other sorting algorithms which are more efficient than the ones defined in terms of linear data types. They make direct or indirect use of the binary tree whose depth is proportional to the logarithm of the number of items stored in it.

2.5.5 Merging Ordered Lists

A problem which occurs frequently in information processing is that of merging two ordered lists. It usually occurs in the form of the file merging problem in which the task is to merge two ordered files into one ordered file. We shall define a function which accepts two ordered lists and merges them into one ordered list. We think of the ordered lists as queues. We can immediately see two ways of solving the problem.

(1) The result is a queue to which always the smaller of the heads of the two input queues is added until both input queues are empty. If one queue is empty, we simply append the other to the already existing partial result.

(2) The elements of one of the lists can be inserted into the other. If the operation ADDQ can be carried out in constant time, then the first method is much more efficient than the second. The merging operation requires \( m + n \) comparisons if \( m \) and \( n \) are the lengths of the two queues.
The number of comparisons required by the second method would be proportional to \( m \times n \), because \( n \) elements must be inserted into a list whose length is \( m \) initially and grows to \( m + n \).

Using the first method, the function \( \text{MERGE} \) can be defined as:

\[
\text{MERGE} (\text{Orderedlist}, \text{Orderedlist}) \rightarrow \text{Orderedlist},
\]

or

\[
\text{MERGE} (\text{Queue}, \text{Queue}) \rightarrow \text{Queue}.
\]

Let \( X \) and \( Y \) be the two ordered lists (queues). There is the special case in which one of the lists is empty:

(1) \( X \) is empty: \( \text{MERGE}(X, Y) = Y \),

(2) \( Y \) is empty: \( \text{MERGE}(X, Y) = X \).

This case will usually arise during the merge operation. It is the termination case for our algorithm. We simply append the non-empty queue to the result. In the general case we must select the smaller of the two minima and add it to the output.

(2) neither \( X \) nor \( Y \) is empty:

(a) \( \text{MIN}(X) < \text{MIN}(Y) \)

(b) \( \text{MIN}(X) > \text{MIN}(Y) \)

In case (a) we must add \( \text{MIN}(X) \) to the output and then append the result of merging the remainder of \( X \) with \( Y \). Case (b) is the same as (a) with \( X \) and \( Y \) interchanged.
The first step of the general case is to add the smaller of the two heads to an empty queue.

The recursive scheme is easiest to express when we always add to the new queue and then append to it the result of merging the rest of the two queues. In that way the termination case fits in as defined, namely the non empty queue is returned by the recursive call of \textsc{merge} and is then appended by the calling incarnation of \textsc{merge}.

The algorithm is

\begin{verbatim}
MERGE(Queue, Queue) -> Queue

MERGE(X, Y) =
    if ISNEWQ(X) then Y
    else if ISNEWQ(Y) then X
    else if HEADQ(X) < HEADQ(Y) then
        APPENDQ(ADDQ(HEADQ(X), NEWQ),
                 MERGE(DELQ(X), Y))
    else
        APPENDQ(ADDQ(HEADQ(Y), NEWQ),
                 MERGE(DELQ(Y), X))
\end{verbatim}

For a Pascal implementation of this algorithm we consider the situation in which the two input queues \(X\) and \(Y\) are ordered files of items, which are to be merged into one ordered output file. Reading from a sequential file is a queue operation, so is writing to the end of a sequential file. In Pascal, the queue operation \(\text{HEADQ}(X)\) can be represented by inspection of the buffer for the file \(X\). The function \(\text{DELQ}(X)\) can be represented by the procedure \text{get}(X)\), which deletes the current element from its file argument. The operation \(\text{ADDQ}(i, X)\) is represented by the procedure \text{write}(X, i)\) which writes the item \(i\) on the file \(X\).
Because we are dealing with files, we construct an iterative version of MERGE directly from the recursive definition. MERGE is tail-end recursive, which means the recursive call is the last action taken by the function, unless we have the termination case. To write such functions in iterative form, we simply negate the termination condition and use it as the condition in a while loop.

Now we must transform

\[
\text{APPENDQ(}
\text{ADDQ(HEADQ(X), NEWQ),}
\text{MERGE(DELQ(X), Y))}.
\]

The resulting chain of recursive calls ends when X (or Y, respectively) is empty. In the meantime, we simply add the head of X to the output file and continue the operation with the rest of X. To append Y to the output file simply means to add one element of Y after another to it. Therefore, this part of the recursive algorithm is equivalent to

\[
\text{while not ISNEW(X) do}
\text{ADDQ(HEAD(X), Z).}
\]

The other case, when HEAD(Y) < HEAD(X), is the same with X replaced by Y. We can now put the two together and bring the case distinction into the loop:
while not (ISNEW(X) or ISNEW(Y)) do
  if HEAD(X) < HEAD(Y) then begin
    ADDQ(HEAD(X), Z);
    DELQ(X)
  end
  else begin
    ADDQ(HEAD(Y), Z);
    DELQ(Y)
  end

As soon as one of the lists has become empty, we can simply move the other one across to the output list. We use a boolean variable finished to control the loop. When one of the lists is empty, finished = true.

The type and variable declarations below define the mapping of the data objects into the data objects of Pascal. The definition of Item-type is left open, the ordered lists are files of items:

```pascal
type
  Itemtype = ...;
  Orderedlist = file of Itemtype;
var
  x, y, z : Orderedlist;
```

After re-writing the recursive function in iterative form with a while loop, the body of the operation becomes:
procedure MERGE(var x, y, z : Orderedlist);
  var
    finished : boolean;
  begin
    reset(x); reset(y); rewrite(z);
    finished := false;
    while not finished do begin
      if eof(x) then begin
        IDVE(y, z);
        finished := true
      end
      else if eof(y) then begin
        IDVE(x, z);
        finished := true
      end
      else if x^ < y^ then begin
        write(z, x^);
        get(x)
      end
      else begin
        write(z, y^);
        get(y)
      end
    end;
  end;

Because all additions to the resulting ordered list, z, take place via a procedure, and because files or file pointers are not returned from functions in Pascal, we write this operation as a procedure. We also require a procedure MOVE(a, b), which which reads the ordered list a and adds it's contents item by item to the ordered list b.

procedure MOVE(var a, b : Orderedlist);
begin
  while not eof(a) do begin
    write(b, a^);
    get(a)
  end
end
Exercises for this section in preparation.