4. What characteristics must a sequence possess if its inverse is also a sequence?
5. Write a Z predicate which states that a given sequence of characters s is a palindrome; that is, it spells the same backwards as it does forwards.

CHAPTER 10

A third specification: Project allocation

Aims

To build on the material of the previous chapters by developing a further specification which uses sequences, and to introduce the concept of schema composition. can be applied to web service composition !

Learning objectives

When you have completed this chapter, you should be able to:

- feel more confident in identifying appropriate applications of sequences in your specifications;
- develop specifications which use combinations of all the discrete mathematical structures which you have met in this book;
- use schema composition to create new operation schemas from existing ones.

10.1 Introduction

In the previous chapter, we developed a specification which used a sequence to model a queue of people. In this chapter, we will take this a stage further by using sequences in combination with functions to model a more complex situation, namely the allocation of undergraduate projects on a university degree course.

10.2 Allocation of undergraduate projects: the problem

A university requires a computerised system to manage the allocation of the individual projects undertaken by its final-year degree students. Each student

must be allocated to a personal supervisor from the lecturing staff. Each lecturer has a maximum number of students which s/he is required to supervise. Each student and each lecturer must list their areas of interest (in descending order of their enthusiasm for the topic!) and the system must attempt to allocate students to supervisors in such a way that the maximum contentedness with the result is created. Contentedness is a difficult concept; we all have an intuitive idea of how much of it we possess at any given time, but it is not easy to quantify in the context of a potentially large group of students and lecturers. Inevitably, some people will be more content with the allocation than others! To make things worse, we are trying to specify contentedness in a formal language, where we must give a precise definition. This means that we have to simplify and make compromises. We will arbitrarily decide that the priority is to allocate the student currently under consideration his/her most desired choice of topic from those which are available (i.e. those which are on the 'areas of interest' list of at least one lecturer who has not already got a full complement of students to supervise), the student being allocated to the lecturer who has this topic highest on his/her list of preferences. If more than one lecturer has the topic at the same level of priority, an arbitrary choice of supervisor can be made from these lecturers. We are thus putting the students' wishes above those of the lecturers.

10.3 Basic types and the system state schema

These are

$[PERSON]$ the set of all people
$[TOPIC]$ the set of all academic areas of interest

ProjectAlloc

$$studInterests, lecInterests : PERSON \pfun \operatorname{iseq} TOPIC$$
$$allocation : PERSON \pfun PERSON$$
$$maxPlaces : PERSON \pfun \mathbb{N}$$

$$\operatorname{dom} studInterests \cap \operatorname{dom} lecInterests = \{\}$$
$$\operatorname{dom} allocation \subseteq \operatorname{dom} studInterests$$
$$\operatorname{ran} allocation \subseteq \operatorname{dom} lecInterests$$
$$\operatorname{dom} maxPlaces = \operatorname{dom} lecInterests$$
$$\forall lec : \operatorname{dom} maxPlaces \bullet \#(allocation \rhd \{lec\}) \leqslant maxPlaces\ lec$$

The functions *studInterests* and *lecInterests* map individual students and lecturers respectively to their preference lists. The lists are represented by injective sequences of topics, which means that a given topic cannot appear in a given list more than once; that is, there are no duplicates. The lists are assumed to be sorted into descending order of preference for the topics.

The set of all students in the system is

$$\operatorname{dom} studInterests$$

and the set of all lecturers in the system is

$$\operatorname{dom} lecInterests$$

The function *allocation* relates the students who have been allocated a project to their supervisors. It is a function because this means that any given student can be related to at most one supervisor.

The function *maxPlaces* relates each lecturer to the maximum number of students which they are required to supervise. This specification will not allow any lecturer to supervise more than their maximum number of students. We allow the possibility that a lecturer may be in the system with *no* places as his/her maximum, so that the system may contain the information as to the lecturer's interests for some future allocation.

The predicate

$$\operatorname{dom} studInterests \cap \operatorname{dom} lecInterests = \{\}$$

specifies that a person in the system cannot be both a student and a supervisor.

The predicates

$$\operatorname{dom} allocation \subseteq \operatorname{dom} studInterests$$
$$\operatorname{ran} allocation \subseteq \operatorname{dom} lecInterests$$

specify that those students and lecturers who have been allocated to projects must be students and lecturers in the system.

The predicate

$$\operatorname{dom} maxPlaces = \operatorname{dom} lecInterests$$

specifies that *all* lecturers in the system have a designated maximum number of supervisions which they may undertake.

The initial state is as follows:

InitProjectAlloc

$$ProjectAlloc'$$

$$lecInterests' = \{\}$$
$$studInterests' = \{\}$$

These values for $lecInterests'$ and $studInterests'$ imply that all of the sets in the abstract state are empty, and all of the state invariant predicates are satisfied, so that this is a valid state for the system.

10.4 Operations

We will now specify the successful cases of operations to add a student to the system, add a lecturer to the system, allocate a supervisor to a student, deallocate a supervisor from a student, remove a topic from a lecturer's preference list and output the set of all lecturers available for the supervision of a given topic.

Adding a student to the system

The inputs required for this operation are a student and that student's list of preferences for his/her project topic. The list is assumed to be organised in descending order of the student's preference for the topics therein.

AddStudent

Δ *ProjectAlloc*
$s? : PERSON$
$ts? : \text{iseq } TOPIC$

$s? \notin \text{dom } studInterests \cup \text{dom } lecInterests$
$studInterests' = studInterests \cup \{s? \mapsto ts?\}$
$lecInterests' = lecInterests$
$allocation' = allocation$
$maxPlaces' = maxPlaces$

(checking that s? is not a lecturer!)

The student must not already be in the system, as either a lecturer or a student!

$$s? \notin \text{dom } studInterests \cup \text{dom } lecInterests$$

The appropriate maplet is added to the *studInterests* function.

$$studInterests' = studInterests \cup \{s? \mapsto ts?\}$$

Note that it would be sufficient to have the precondition

$$s? \notin \text{dom } studInterests$$

because

$$s? \notin \text{dom } lecInterests$$

is implied by the postcondition

$$studInterests' = studInterests \cup \{s? \mapsto ts?\}$$

and the 'after' state invariant

$$\text{dom } studInterests' \cap \text{dom } lecInterests' = \{\}$$

In fact, we don't need an explicit precondition at all, as the above postcondition implies that either

$$s? \mapsto ts? \in studInterests$$

in which case the operation has no effect on the state, or

$$s? \notin \text{dom } studInterests$$

This is because *studInterests* is a function, so $s?$ cannot be mapped to more than one range element. However, in an implementation of the system, we would like the system to produce a message to tell the user when such events occur, so we will leave the explicit precondition in the schema, and write appropriate error case schemas to produce a total version of the operation.

Adding a lecturer to the system

The inputs required for this operation are a lecturer, the list of topics which that lecturer is prepared to supervise and the maximum number of students which the lecturer may supervise. Again, the list of topics is assumed to be organised in descending order of the lecturer's preference for the topics.

AddLecturer

Δ *ProjectAlloc*
$l? : PERSON$
$ts? : \text{iseq } TOPIC$
$maxAlloc? : \mathbb{N}_1$

$l? \notin \text{dom } studInterests \cup \text{dom } lecInterests$
$lecInterests' = lecInterests \cup \{l? \mapsto ts?\}$
$maxPlaces' = maxPlaces \cup \{l? \mapsto maxAlloc?\}$
$studInterests' = studInterests$
$allocation' = allocation$

The lecturer must not already be in the system, as either a lecturer or a student!

$$l? \notin \text{dom } studInterests \cup \text{dom } lecInterests$$

The appropriate maplet is added to the *lecInterests* function

$$lecInterests' = lecInterests \cup \{l? \mapsto ts?\}$$

and to the *maxPlaces* function.

$$maxPlaces' = maxPlaces \cup \{l? \mapsto maxAlloc?\}$$

Exercises 10.1

1. Write a predicate which specifies a state where there are no unallocated students.
2. Write a predicate which specifies a state where all students in the system are unallocated.
3. Write a schema for an operation to remove a student from the system.
4. Write a schema for an operation to remove a lecturer from the system.
5. Write a Z expression for the set of all the students allocated to a given supervisor *s*.
6. Write a Z expression for the set of all students with the same supervisor as a given student *p*.

Allocating a student to a supervisor

The input to this operation is the student to be allocated. The operation must allocate this student to a supervisor in such a way that the student gets to do the highest priority topic from his/her list for which a supervisor is available. ('Available' means that the topic appears in the preference list of at least one supervisor who still has places left for supervisions.) Remember that preference lists are assumed to be sorted into descending order of preference. Additionally, the student is to be allocated to the lecturer who has this topic highest on his/her list of preferences. If more than one lecturer has the topic at the same level of priority, an arbitrary choice of supervisor will be made from these lecturers.

Allocate

Δ *ProjectAlloc*
$s?$: *PERSON*

$s? \in \text{dom } studInterests$
$s? \notin \text{dom } allocation$

$\exists sup$: dom *lecInterests*; t : *TOPIC*; i, j : $\mathbb{N}$
| *maxPlaces sup* > # (*allocation* $\rhd$ {*sup*})
$\wedge\ i \mapsto t \in studInterests\ s?$
$\wedge\ j \mapsto t \in lecInterests\ sup$
• (
$\forall lec$: dom *lecInterests*; k : $\mathbb{N}$ | *maxPlaces lec* > # (*allocation* $\rhd$ {*lec*})
• (
$(k \mapsto t \in lecInterests\ lec \Rightarrow k \geqslant j)$
$\wedge$
$(\text{ran}(1 \,..\, i-1 \lhd studInterests\ s?) \cap \text{ran}(lecInterests\ lec) = \{\})$
)
$\wedge\ allocation' = allocation \cup \{s? \mapsto sup\}$
)
$studInterests' = studInterests$
$lecInterests' = lecInterests$

The student must be in the system, and must not be already allocated to a supervisor.

$s? \in \text{dom } studInterests$
$s? \notin \text{dom } allocation$

The other preconditions and postconditions of interest are specified by the rather complex quantified expression above, which we will look at piece by piece.

There is a supervisor who has places left for project supervision and who has one of the students' topics on his/her list of preferences:

$\exists sup$: dom *lecInterests*; t : *TOPIC*; i, j : $\mathbb{N}$
| *maxPlaces sup* > # (*allocation* $\rhd$ {*sup*})
$\wedge\ i \mapsto t \in studInterests\ s?$
$\wedge\ j \mapsto t \in lecInterests\ sup$

There are no supervisors available who have this topic at a higher priority in their list of preferences than the given supervisor

$\forall lec$: dom *lecInterests*; k : $\mathbb{N}$ | *maxPlaces lec* > # (*allocation* $\rhd$ {*lec*})
• (
$(k \mapsto t \in lecInterests\ lec \Rightarrow k \geqslant j)$

and there are no supervisors available who are interested in any of the topics which the student has at a higher priority than the given topic.

$$\begin{array}{l} \\ \quad \wedge \\ \quad (\mathrm{ran}(1\,..\,i-1 \lhd studInterests\ s?) \cap \mathrm{ran}(lecInterests\ lec) = \{\}) \\ ) \end{array}$$

The allocation of the student to the supervisor is specified by

$$allocation' = allocation \cup \{s? \mapsto sup\}$$

Deallocating a student from a supervisor

The input to this operation is the student to be deallocated. The precondition is that the student is allocated to a supervisor.

DeAllocate

$\Delta\ ProjectAlloc$
$s? : PERSON$

$\exists\, sup : \mathrm{dom}\ lecInterests$
$\quad \bullet\ (s? \mapsto sup) \in allocation$
$\quad\quad \wedge\ allocation' = allocation \setminus \{s? \mapsto sup\})$
$studInterests' = studInterests$
$lecInterests' = lecInterests$

Allocation policies

We now have operations to add new people to the system and to allocate students to supervisors according to the preferences of both. However, we have said nothing about the temporal aspects of our allocation policy. The initial state is one with no people in the system. Clearly we must first use our 'add' operations to place some people into the system, before we can allocate any students to supervisors. In a real system, we might add all the lecturers who are designated for project supervision first, and then either add each student in turn in the order in which they show up for registration, immediately allocating each student to a supervisor (first-come, first-served policy), or add all the students, and then randomly allocate them to supervisors until there are no unallocated students left in the system. A more sophisticated policy could be specified in an effort to improve the overall 'contentedness' with the allocation; that is, a policy which pleases most of the people most of the time, rather than the first students to come along getting their highest preferences, and later

optimization of contentedness

students getting whatever is still available. In the interests of simplicity, we will leave such considerations out of the current specification, but you may wish to think about how such ideas could be represented in Z. Of course, real life is never straightforward, and it is possible that all supervisors will have used up their allocation while there are still students unallocated, so that extra supervisors have to be added, or that there are students whose chosen preferences for topics do not occur in the preference lists for any supervisors, in which case further compromises will be necessary.

Removing a topic from a lecturer's preference list

The lecturer must be in the system and the topic must be in the lecturer's preference list. If the lecturer is already allocated to supervise this topic, it's too bad – the lecturer has made an agreement! However, the lecturer will not have to take on any further students for this topic.

So more than one student can take a topic from a supervisor

RemoveLecsTopic

$\Delta\ ProjectAlloc$
$l? : PERSON$
$t? : TOPIC$

$l? \in \mathrm{dom}\ lecInterests$
$t? \in \mathrm{ran}(lecInterests\ l?)$
$lecInterests' = lecInterests \oplus \{l? \mapsto squash(lecInterests\ l? \rhd \{t?\})\}$
$maxPlaces' = maxPlaces$
$studInterests' = studInterests$
$allocation' = allocation$

function update covers this?

Range anti-restriction is used to remove the topic from the sequence, and *squash* is used to make the result into a sequence. An alternative would have been to use an existentially quantified expression as follows:

$$\begin{array}{l} \exists\, higher, lower : \mathrm{iseq}\ TOPIC \\ \quad \bullet\ (lecInterests\ l? = higher \frown \langle t? \rangle \frown lower \\ \quad\quad \wedge\ lecInterests'\ l? = higher \frown lower) \end{array}$$

The set of all lecturers available for supervision of a given topic

For a given lecturer to be a member of this set, the topic must be in that lecturer's list of preferences and the lecturer must still have supervision places available.

LecsAvailable

$$\Xi\ ProjectAlloc$$
$$t? : TOPIC$$
$$ps! : \mathbb{P}\ PERSON$$

$$ps! = \{p : \mathrm{dom}\ lecInterests \mid t? \in \mathrm{ran}(lecInterests\ p) \land maxPlaces\ p > \#\ (allocation \rhd \{p\})\}$$

Exercises 10.2

1. Write a schema for an operation to add a new topic to a lecturer's priority list at a given position. If the position is greater than the length of the list, the topic should be added to the back of the list.
2. Write a Z expression for the set of all students who are doing project topic t.
3. Write a schema for an operation to output the topic which a given student was allocated for his/her project.
4. Write a Z expression for the set of all unallocated students for whom none of their chosen topics are available for supervision.
5. Write a Z expression for the set of all students who were allocated their nth choice of topic in order of preference, where n is a natural number (excluding zero).

10.5 Error handling schemas

The following free type represents the set of output messages required to construct total versions of the above operations, and for reports from the query operations which are defined below:

$$MESSAGE ::= success \mid isStudent \mid isLecturer \mid notStudent \mid isAllocated \mid noLecAvailable \mid notAllocated \mid notLecturer \mid notListedTopic$$

The *success* message is used to indicate that an operation has been successfully completed, using the following schema:

$$SuccessMessage \mathrel{\widehat=} [outcome! : MESSAGE \mid outcome! = success]$$

The precondition for the *AddStudent* operation is

$$s? \notin \mathrm{dom}\ studInterests \cup \mathrm{dom}\ lecInterests$$

The exceptions to this operation occur if $s?$ is already a student or a lecturer.

In these cases, the state does not change and the appropriate message is produced. This is specified by the following schemas:

IsStudent

$$\Xi\ ProjectAlloc$$
$$s? : PERSON$$
$$outcome! : MESSAGE$$

$$s? \in \mathrm{dom}\ studInterests$$
$$outcome! = isStudent$$

IsLecturer

$$\Xi\ ProjectAlloc$$
$$s? : PERSON$$
$$outcome! : MESSAGE$$

$$s? \in \mathrm{dom}\ lecInterests$$
$$outcome! = isLecturer$$

For the *AddLecturer* operation, the exceptions are specified by the same schemas with appropriate renaming:

$$IsStudent\ [s? / l?]$$
$$IsLecturer\ [s? / l?]$$

The preconditions for the *Allocate* operation are that the student is in the system and is not already allocated:

$$s? \in \mathrm{dom}\ studInterests$$
$$s? \notin \mathrm{dom}\ allocation$$

and that there is at least one lecturer with places left who has at least one of the student's preferred topics on his/her preference list:

$$\exists\ sup : \mathrm{dom}\ lecInterests \bullet maxPlaces\ sup > \#\ (allocation \rhd \{sup\}) \land \mathrm{ran}(studInterests\ s?) \cap \mathrm{ran}(lecInterests\ sup) \neq \{\}$$

This leads to the following exception schemas:

NotStudent

Ξ *ProjectAlloc*
s? : *PERSON*
outcome! : *MESSAGE*

---

s? $\notin$ dom *studInterests*
outcome! = *notStudent*

IsAllocated

Ξ *ProjectAlloc*
s? : *PERSON*
outcome! : *MESSAGE*

---

s? $\in$ dom *allocation*
outcome! = *isAllocated*

NoLecAvailable

Ξ *ProjectAlloc*
s? : *PERSON*
outcome! : *MESSAGE*

$\neg \exists$ *sup* : dom *lecInterests* •
maxPlaces sup > # (*allocation* $\rhd$ {*sup*})
$\wedge$ ran(*studInterests s*?) $\cap$ ran(*lecInterests sup*) $\neq$ { }
outcome! = *noLecAvailable*

The precondition for the *DeAllocate* operation is that the student is allocated to a supervisor. This leads to the following exception schema:

NotAllocated

Ξ *ProjectAlloc*
s? : *PERSON*
outcome! : *MESSAGE*

---

s? $\notin$ dom *allocation*
outcome! = *notAllocated*

The preconditions for the *RemoveLecsTopic* schema are that the person is a lecturer in the system and that the topic is in the lecturer's preference list.

l? $\in$ dom *lecInterests*
t? $\in$ ran(*lecInterests l*?)

This leads to the following exception schemas:

NotLecturer

Ξ *ProjectAlloc*
l? : *PERSON*
outcome! : *MESSAGE*

---

l? $\notin$ dom *lecInterests*
outcome! = *notLecturer*

NotListedTopic

Ξ *ProjectAlloc*
l? : *PERSON*
t? : *TOPIC*
outcome! : *MESSAGE*

---

t? $\notin$ ran(*lecInterests l*?)
outcome! = *notListedTopic*

10.6 Total operations

We can now define the total versions of the operation schemas using the above exception handling schemas.

$$TotalAddStudent \mathrel{\widehat=} (AddStudent \wedge SuccessMessage) \vee IsStudent \vee IsLecturer$$

$$TotalAddLecturer \mathrel{\widehat=} (AddLecturer \wedge SuccessMessage) \vee IsStudent\ [s?/l?] \vee IsLecturer\ [s?/l?]$$

$$TotalAllocate \mathrel{\widehat=} (Allocate \wedge SuccessMessage) \vee NotStudent \vee IsAllocated \vee NoLecAvailable$$

$$TotalDeAllocate \mathrel{\widehat=} (DeAllocate \wedge SuccessMessage) \vee NotAllocated$$

$$TotalRemoveLecsTopic \mathrel{\widehat=} (RemoveLecsTopic \wedge SuccessMessage) \vee NotLecturer \vee NotListedTopic$$

The *LecsAvailable* operation has no precondition; it may be applied in any state.

10.7 A new concept: operation schema composition

The composition of two operation schemas A and B is a schema which specifies the effect of doing operation A followed by operation B. It is written $A \,;\, B$.

Let A and B be the following schemas:

$$\begin{array}{|l} A \\ \hline a, a', c! : \mathbb{Z} \\ \hline a' = a + 42 \\ c! = a' \\ \hline \end{array} \qquad \begin{array}{|l} B \\ \hline a, a', b? : \mathbb{Z} \\ \hline b? < 10 \\ a' = a + b? \\ \hline \end{array}$$

All variables with the same name (ignoring decorations) in the two schemas must also have the same type. The composition is formed as follows.

Firstly, we rename each variable which is both an 'after' variable (primed) in A and a 'before' variable (unprimed) in B to the same, new name. (See Chapter 4 if you need a reminder of what is meant by renaming.) In our example, we will rename a' from A and a from B to the new name a_c to give the schemas $NewA \mathrel{\widehat=} A[a_c / a']$ and $NewB \mathrel{\widehat=} B[a_c / a]$:

$$\begin{array}{|l} NewA \\ \hline a, a_c, c! : \mathbb{Z} \\ \hline a_c = a + 42 \\ c! = a_c \\ \hline \end{array} \qquad \begin{array}{|l} NewB \\ \hline a_c, a', b? : \mathbb{Z} \\ \hline b? < 10 \\ a' = a_c + b? \\ \hline \end{array}$$

Next, we conjoin the two schemas, and hide the renamed variables. (Again, see Chapter 4 if you need a reminder of what is meant by hiding.) In our example, this gives the schema

$$AcompB \mathrel{\widehat=} A \,;\, B$$

where

$$AcompB \mathrel{\widehat=} (A[a_c / a'] \wedge B[a_c / a]) \setminus (a_c)$$

$$\begin{array}{|l} AcompB \\ \hline a, a', b?, c! : \mathbb{Z} \\ \hline \exists a_c : \mathbb{Z} \\ \bullet (a_c = a + 42 \\ \quad \wedge c! = a_c \\ \quad \wedge b? < 10 \\ \quad \wedge a' = a_c + b?) \\ \hline \end{array}$$

10.8 Changing supervisor

We will now use the concept of schema composition to define an operation whereby a student, already assigned to a supervisor, changes to another supervisor. This operation may be specified as the composition of the *DeAllocate* and *Allocate* operations. However, this would allow the possibility that the new supervisor was the same as the old supervisor. We can disallow this by conjoining the result with a schema which specifies that the 'before' supervisor and the 'after' supervisor are different.

$$\begin{array}{|l} SupsDiffer \\ \hline \Delta\, ProjectAlloc \\ s? : PERSON \\ \hline \exists old, new : \mathrm{dom}\ lecInterests \\ \bullet (s? \mapsto old \in allocation \wedge \\ \quad s? \mapsto new \in allocation' \wedge \\ \quad old \neq new) \\ \hline \end{array}$$

The operation to change supervisor may now be specified.

$$ChangeSup \mathrel{\widehat=} (DeAllocate \,;\, Allocate) \wedge SupsDiffer$$

Note that the *Allocate* schema is non-deterministic in that more than one final state is possible (more than one lecturer may have places left and have the appropriate topic at the same priority on their list). Thus the *ChangeSup* operation is also non-deterministic. If we were to compose such a schema with a schema for which not all of those final states satisfied the precondition, then only the valid final states would count in the composition schema.

Exercise 10.3

How could we specify allocation which was on a first-come, first-served basis?

10.9 Abstraction

This specification is a little more abstract than the video shop specification of Chapter 8. In the latter, we used a *number* to represent the stock level of each video, comparing this number with the number of copies of the video out on rental when deciding whether any copies were available for rental. Modelling the concept of the whereabouts of copies using numbers is an example of

modelling at a *low* level of abstraction, akin to modelling an algorithm in a programming language. (See the answer to question 8 of Exercises 8.1 for a suggested alternative, more abstract, representation for the concept of a video copy for this specification.) This type of modelling may lead to a specification which is easier to implement in a programming language, but may also lead to a lack of clarity in the specification, and possibly less flexibility if it later becomes necessary to modify the specification to accommodate new requirements.

In contrast, the project allocation specification has a higher level of abstraction, with some use of non-determinism as already discussed. It may not be quite so obvious how to produce an implementation from such a specification, as it very much specifies *what* the system must do and not *how* it is to do it. However, the greater level of abstraction means that we can study the problem in its own right without having to be concerned about the paraphernalia of how we are eventually going to produce a program. So is it better to write specifications with a low level of abstraction, or a high level of abstraction? As always in the field of science, the answer is 'it depends'! The important thing is that the specification correctly represents the requirements, and is clear and understandable to all those who must read, discuss and use it. There are techniques in Z for writing specifications at a high level of abstraction, and then refining them to a lower level of abstraction, while proving that the properties of the original specification are preserved, thus getting the best of both worlds. However, these techniques are beyond the scope of this book.

CHAPTER 11

Two outline Z specifications

Aims

To apply the preceding material to some more demanding specifications, and to illustrate the use of generic constants.

Learning objectives

When you have completed this chapter, you should be able to:

- apply the Z notation to modelling the state of more complex systems using combinations of mathematical structures;
- specify more challenging operations;
- appreciate the need to validate your specifications to ensure they are a true description of the required system;
- recognise situations in which it is desirable to define a new operator, or to name subexpressions using **let**, to enhance the readability of a specification.

11.1 Introduction

In this chapter, we will outline the construction of two Z specifications. The specifications are incomplete and not very large, but they are a little more challenging than the examples we have encountered so far, and should illustrate the value of Z in expressing complex ideas clearly and precisely.

11.2 A timetabling system

A university requires a timetabling system to keep track of where and when its degree modules are scheduled, and which students are registered for each module. Sometimes a module can be scheduled in several rooms at the same