Z SCHEMAS

This page is intentionally blank

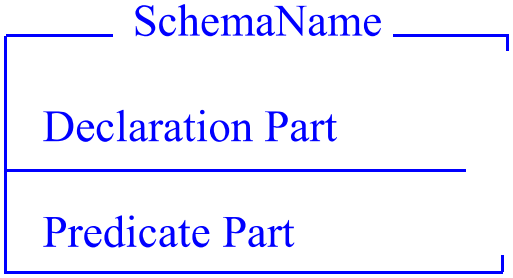
☺

INTRODUCTION

- A document written in the Z specification language comprises
  - *narrative text* in natural language, and
  - *formal symbolic descriptions* written in the Z notation

- To help distinguish the two forms Z employs a 'graphical' format called a *schema*

- A schema usually has a box-like framework containing mathematical text and comprises two parts:
  - a *declaration* part, and
  - a *predicate* (or *axiom*) part



VERTICAL FORMAT

- The 'box-frame' is often called the *vertical* form of a schema

- Given the base types [PASSWORD, USERID] the following schema might be part of the specification of a system module which:

  - monitors the passwords of users, and
  - identifies which users are logged-on:

TimeSharing

$$password : USERID \pfun PASSWORD$$
$$loggedOn : \mathbb{P}\ USERID$$

$$loggedOn \subseteq \mathbf{dom}\ password$$

- *TimeSharing* is the name of the schema and may be used to reference the schema from other parts of the specification document

- The declaration part contains two *components* (also known as *variables* or *observations*):

  - *password* and *loggedOn*

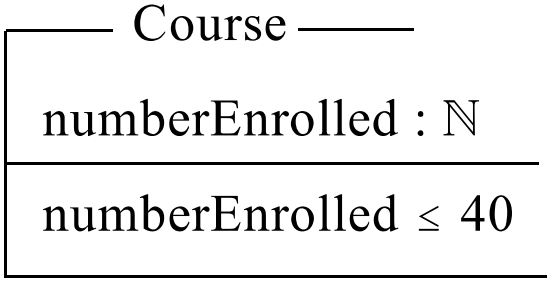
HORIZONTAL FORMAT

- If there are only a small number of components in the declaration part of a schema, then it is possible to adopt a *linear* or *horizontal* format:

  SchemaName ≙ [declaration(s) | predicate(s)]

  where ≙ is read as "*is defined as*"

- For example:



  could, instead, be represented as:

  Course ≙ [numberEnrolled : ℕ | numberEnrolled ≤ 40]

NAMING SCHEMAS IN Z

- It is possible to have an *anonymous* schema where the schema name is omitted but such a schema cannot then be referenced from other parts of a specification document

- Names of schemas and variables are frequently written in 'Smalltalk' format but other forms are also favoured:

  TimeSharing and *Time_Sharing* are both seen

- It is possible to have a schema with, apparently, no constraining predicate part

  For example:

  TimeHM

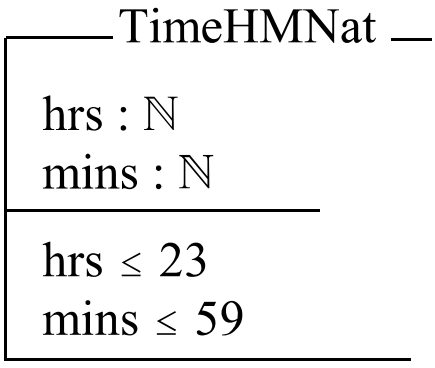
  hrs : 0 .. 23
  mins : 0 .. 59

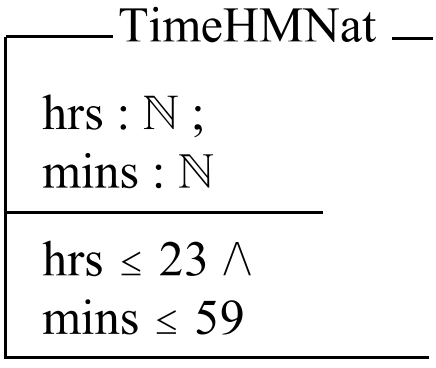
  might model the time components of a digital watch

IMPLICIT SYNTAX

- If a schema has several **declaration** components they are regarded as separated by *semi-colons* but these are, usually, explicit only in the linear (i.e. horizontal) form

- If the **predicate** part comprises a number of separate predicates then *conjunction* is implicit (i.e. the separate predicates are regarded as ANDed together). Thus:





are equivalent

AXIOMATIC DESCRIPTIONS

- *Global* variables (whose scope extends to all schemas in a specification) may be defined, possibly with a constraining predicate, in an *axiomatic description* (aka *axiomatic definition*)

- For example we can define something called *numberEnrolled* using:

$$\begin{array}{|l} \text{numberEnrolled} : \mathbb{N} \\ \hline \text{numberEnrolled} \leq 40 \\ \text{numberEnrolled} \geq 12 \end{array}$$

- We can then reference *numberEnrolled* in schemas without needing to declare it further

- Note that the box-frame has here lost its bounding upper and lower bars to emphasise the "global" nature of the declarations

AXIOMATIC DESCRIPTIONS

- The axiomatic definition:

$$\begin{array}{|l} size : \mathbb{Z} \\ \hline size \geq 0 \end{array}$$

may be, equally well, specified without any constraint using a form such as:

$$\begin{array}{|l} size : \mathbb{N} \end{array}$$

- If we wish to show that a function is to be used in *infix* form then *underscores* are used as "place-holders" in the axiomatic definition:

$$\begin{array}{|l} \_ \natural \_ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall x, y : \mathbb{N} \bullet x \natural y = x^2 + y^2 \end{array}$$

SCOPE

- It is important to appreciate that the declaration part of any schema merely introduces *variable* (or *component*) *names* and establishes their *types*

- Any variable declared in a schema is not 'known' in the declaration part - constraints governing such variables can be expressed only in the predicate part

- Any variable introduced in the declaration part of a schema is *local* to that schema and its scope extends only to the predicate part of the same schema

SCOPE

- Consider a schema called *Library*:

Library

onLoan, onShelves, books : ℙ BOOK
borrowers : ℙ PERSON
lentTo : BOOK ⇸ PERSON

..... predicates here .....

- Suppose we wish to assert the constraints:
  - the books associated with borrowers in *lentTo* are exactly the books in *onLoan*, and
  - the persons associated with books in *lentTo* must all be members of *borrowers*

- Since any declared variable is *known* only in the predicate part we **cannot** formalize the above assertions by declaring:

$$lentTo : onLoan \rightarrow borrowers$$

but must rather use predicates such as:

$$onLoan = \mathbf{dom}\ lentTo$$
$$\mathbf{ran}\ lentTo \subseteq borrowers$$

IMPLICIT PREDICATES

- Strictly, *lentTo* has *type* $\mathbb{P}$(BOOK×PERSON) but the declaration of *lentTo* uses the style:

$$\text{BOOK} \nrightarrow \text{PERSON}$$

  where we expect to find the **type** declaration

- The form of declaration used stresses the **functional** nature of the *lentTo* relation and, in effect, makes the following predicate *implicit* in the predicate part of the *Library* schema:

$$\forall\, b : \text{BOOK};\ p_1, p_2 : \text{PERSON}$$
$$\bullet\ (\mathit{lentTo}\ b = p_1 \land \mathit{lentTo}\ b = p_2) \Rightarrow p_1 = p_2$$

  which is equivalent to:

$$\mathit{lentTo} \in \text{BOOK} \nrightarrow \text{PERSON}$$

- If we use the predicate part of the *Library* schema to reason about the library system then such reasoning **must** take account of this implicit predicate

EXPLANATORY ANNOTATION

- When creating specifications it is important to include appropriate narrative comment to explain the symbolic forms (for the non-technical *end-user* if no-one else)

- One style which is popular is to include blocks of **plain English** text

  - **before each schema**
    explaining *declarations* in the schema, and
  - **after each schema**
    explaining any *constraining predicates*

- In these notes we shall include "comments" with the schemas using one of the styles outlined in the following examples

EXPLANATORY ANNOTATION

- In the following *Library* schema,

Library

$$onLoan, onShelves, books : \mathbb{P}\ BOOK$$
$$borrowers : \mathbb{P}\ PERSON$$
$$lentTo : BOOK \pfun PERSON$$

$$onLoan = \mathbf{dom}\ lentTo \quad (1)$$
$$\mathbf{ran}\ lentTo \subseteq borrowers \quad (2)$$
$$onLoan \cup onShelves = books \quad (3)$$
$$onLoan \cap onShelves = \{\} \quad (4)$$

the numbered predicates may be interpreted (in **plain English**) as:

(1) *the books on loan are exactly those for which we have a record of a borrower*
(2) *books can only be recorded as having been lent to known borrowers*
(3) *every book belonging to the library is either on loan or on the shelves*
(4) *no book can be on loan and on the shelves*

- Observe that although the *Library* schema contains several *redundant* components, such redundancy can often aid understanding

SIGNATURE AND PROPERTY

- The *signature* of a schema is the (unordered) collection of *names* introduced in the *declaration part*, together with their *types*

- The *property* of a schema is a *predicate* derived from
  - the *predicate part* of the schema, and
  - any *implicit predicates* from the declaration part

- The essential characteristics of any schema are defined by its
  - *signature*, and
  - *property*

SIGNATURE AND PROPERTY

- Consider again the schema:

Library

onLoan, onShelves, books : ℙ BOOK
borrowers : ℙ PERSON
lentTo : BOOK ⇸ PERSON

---

onLoan = **dom** lentTo
ran lentTo ⊆ borrowers
onLoan ∪ onShelves = books
onLoan ∩ onShelves = { }

- The *signature* of the *Library* schema would be:

onLoan	: ℙ BOOK
onShelves	: ℙ BOOK
books	: ℙ BOOK
borrowers	: ℙ PERSON
lentTo	: ℙ(BOOK × PERSON)

- The *property* of the *Library* schema would be:

onLoan ∈ ℙ BOOK ∧ *onShelves* ∈ ℙ BOOK ∧
books ∈ ℙ BOOK ∧ *borrowers* ∈ ℙ PERSON ∧
onLoan = **dom** *lento* ∧ *onLoan* ∩ *onShelves* = { } ∧
ran *lentTo* ⊆ *borrowers* ∧ *onLoan* ∪ *onShelves* = *books*
∧ *lentTo* ∈ BOOK ⇸ PERSON

SCHEMAS DESCRIBE STATES

- The *state* of a system is characterized by
  - a **set of values**, together with
  - **permitted inputs**
- Any system state can be modelled by a suitable "state schema"
- The *signature* of the schema specifies which data values are of interest in our description of the state
- The *property* of the schema imposes constraints on the variables described in the signature so that meaningless values are prohibited
- A state schema *property* is often said to specify the s*tate invariant*
- Together, the *signature* and *property* describe the set of all possible states (the *data space*) modelled by the state schema

SCHEMAS DESCRIBE STATES

- The *Library* schema, considered earlier, describes the state of the library at any time:

Library

$$\begin{array}{|l} onLoan, onShelves, books : \mathbb{P}\ BOOK \\ borrowers : \mathbb{P}\ PERSON \\ lentTo : BOOK \pfun PERSON \\ \hline onLoan = \mathbf{dom}\ lentTo \\ \mathbf{ran}\ lentTo \subseteq borrowers \\ onLoan \cup onShelves = books \\ onLoan \cap onShelves = \{\} \\ \hline \end{array}$$

- Three of the possible states of the library are shown as rows in the following table:

onLoan	onShelves	books	borrowers	lentTo
{ }	{ }	{ }	{ }	{ }
{ }	{bk1}	{bk1}	{ }	{ }
{bk1}	{bk2}	{bk1, bk2}	{pers1,pers2}	{bk1 ↦ pers2}

SCHEMAS DESCRIBE OPERATIONS

- Schemas may also be used to describe *operations* (which may cause changes of state)
- Since an operation *may* change values we use
  - *undashed*/*unprimed* names to denote the components which describe the state *before* an operation is performed, and
  - *dashed*/*primed* names to denote component values *after* the operation is performed
- Names ending with ? are used to denote *inputs* to an operation
- Names ending with ! are used to denote *outputs* from an operation
- ?, ! and ′ are often called *decoration* symbols

SCHEMAS DESCRIBE OPERATIONS

- The operation to add a book to the library stock might be modelled by the following schema:

AddBookToLibraryOK

$$
\begin{array}{l}
onLoan, onShelves, books : \mathbb{P}\ BOOK \\
borrowers : \mathbb{P}\ PERSON \\
lentTo : BOOK \pfun PERSON \\
onLoan', onShelves', books' : \mathbb{P}\ BOOK \\
borrowers' : \mathbb{P}\ PERSON \\
lentTo' : BOOK \pfun PERSON \\
b? : BOOK \\
r! : RESPONSE \\
\hline
onLoan = \mathbf{dom}\ lentTo \\
\mathbf{ran}\ lentTo \subseteq borrowers \\
onLoan \cup onShelves = books \\
onLoan \cap onShelves = \{\} \\
onLoan' = \mathbf{dom}\ lentTo' \\
\mathbf{ran}\ lentTo' \subseteq borrowers' \\
onLoan' \cup onShelves' = books' \\
onLoan' \cap onShelves' = \{\} \\
b? \notin books \\
onLoan = onLoan' \\
onShelves' = onShelves \cup \{ b? \} \\
borrowers' = borrowers \\
lento' = lento \\
r! = book\ added
\end{array}
$$

Where RESPONSE ::= *book added* | *on loan* | *already on loan* | etc

SCHEMAS DESCRIBE OPERATIONS

- The foregoing *AddBookToLibraryOK* schema is rather long and complex - later we shall introduce ways of simplifying it

- The first 8 predicates are just the invariants on the *before* state and the *after* state

- The remaining predicates may be interpreted as:
  - b? $\notin$ books — *b cannot be a book already owned by the library*
  - onLoan′ = onLoan — *there is no change to the set of books on loan*
  - onShelves′ = onShelves $\cup$ {b?} — *the new book is added to the set of shelved books*
  - borrowers′ = borrowers — *no change to borrowers*
  - lentTo′ = lentTo — *no change to relation between books on loan and those borrowing them*
  - r! = book added — *message to be displayed*

SCHEMAS DESCRIBE OPERATIONS

- The predicate $b? \notin books$ is concerned only with the input value and the "before" state - it is a *precondition* of the schema

- Other predicates are *postconditions*, concerned only with input, output and "after" state

- Note that the predicate: $books' = books \cup \{b?\}$ although clearly true is not documented explicitly in the schema

- It can, however, be easily deduced from the predicates which are included and is, therefore, redundant, but there is no harm in including it (and possibly something to gain)

- Note that the *AddBookToLibraryOK* schema says nothing about what happens if the precondition is violated

SCHEMA DECORATION

- The *AddBookToLibraryOK* schema is very cluttered

- *Schema decoration* offers a way of introducing the dashed names of the schema components together with the invariants on them

- For example, given a *Library* schema, we can declare *Library′* as a schema that has, for its declaration part, the declaration part of *Library* with a dash/prime on all names (global names are unaffected):

$$onLoan', onShelves', books' : \mathbb{P}\ BOOK$$
$$borrowers' : \mathbb{P}\ PERSON$$
$$lentTo' : BOOK \pfun PERSON$$

- The predicate part of *Library′* is the predicate part of *Library* with dashes/primes on the names from the declaration part:

$$onLoan' = \mathbf{dom}\ lentTo'$$
$$\mathbf{ran}\ lentTo' \subseteq borrowers'$$
$$onLoan' \cup onShelves' = books'$$
$$onLoan' \cap onShelves' = \{\}$$

SCHEMA INCLUSION

- By **including** the *Library* and *Library′* schemas we can dramatically simplify the schema *AddBookToLibraryOK*:

AddBookToLibraryOK

Library
Library′
b? : BOOK
r! : RESPONSE

---

b? ∉ books
onLoan = onLoan′
onShelves′ = onShelves ∪ { b? }
borrowers′ = borrowers
lento′ = lento
r! = book added

- The above process is called *schema inclusion* and results in the included schema(s) being textually imported:
  - **delarations** are *merged*, and
  - explicit **predicates** are *conjoined* (ANDed)

DELTA CONVENTION

- If both a schema S and the decorated (after) version S′ are included inside another schema then the abbreviation ΔS may be used instead

- For example, our *AddBookToLibraryOK* schema could be written:

AddBookToLibraryOK

Δ Library
b? : BOOK
r! : RESPONSE

---

b? $\notin$ books
onLoan = onLoan′
onShelves′ = onShelves $\cup$ { b? }
borrowers′ = borrowers
lento′ = lento
r! = book added

- The use of Δ helps the reader of a specification identify changes of state (updates)

- Δ should only be applied to a schema name if that schema describes a *state*

XI CONVENTION

- If an operation causes **no** change of state then Ξ can be used instead of Δ

- For example, if an attempt was made to add to the library stock a book that was already there, then the operation would fail and variables of the *Library* schema should remain unchanged

- This can be specified with a schema named, say, *BookAlreadyInLibrary*:

BookAlreadyInLibrary

Ξ Library
b? : BOOK
r! : RESPONSE

---

b? ∈ books
r! = book already in library

- The declaration part of Ξ *Library* is the same as that of Δ *Library* but the predicate of Δ *Library* is augmented by:

onLoan′ = onLoan
onShelves′ = onShelves
borrowers′ = borrowers
lentTo′ = lentTo

AIRCRAFT SEATING RE-VISITED

- We shall reconsider the simple system considered earlier which monitors the people who board an aircraft:

> *An aircraft has a fixed capacity and it is required to record the number of people aboard the aircraft at any time. The aircraft seats are not numbered and passengers enter the aircraft and choose seats on a first-come-first-served basis.*

- Our only basic type is [PERSON], the set of all possible persons who may, at some time, board the aircraft

- If the aircraft has a fixed capacity, we may

$$\mid \mathit{capacity} : \mathbb{N}$$

declare:
and assign a definite value to *capacity* at an appropriate time

SYSTEM STATE

- The state of the system is given by the set of people on board the aircraft. Obviously the number of people in this set must never exceed *capacity* (invariant property of system state)

- We may, therefore, use a schema *Aircraft* to specify the system state:

Aircraft

$$onboard : \mathbb{P}\ \text{PERSON}$$

$$\#\ onboard \leq capacity$$

- The states before and after any operation which causes a system update can be represented as $\Delta Aircraft$ where the Δ is used with the conventional Z meaning, and the expanded schema for $\Delta Aircraft$ appears on the next page

CHANGE OF STATE

- Schema for potential change of state:

$$\Delta Aircraft$$
$$onboard : \mathbb{P}\ PERSON$$
$$onboard' : \mathbb{P}\ PERSON$$

$$\#\ onboard \leq capacity$$
$$\#\ onboard' \leq capacity$$

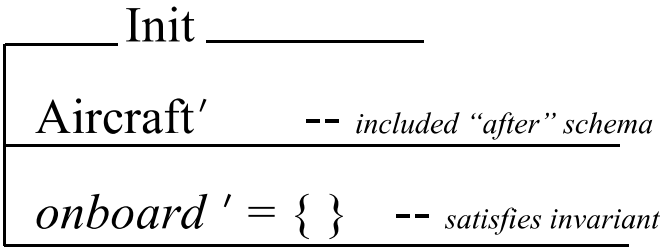
- Important note:
  - It is NOT usual to include schemas such as $\Delta Aircraft$ in a Z specification document because the interpretation of $\Delta Aircraft$ is implicit

- There is no provision in Z for explanatory annotation to be included inside a schema but, in what follows, we shall often include comments to 'explain' the symbolic forms
  - comments will be introduced by a double hyphen (- -) and terminated by end-of-line

INITIAL STATE

- The *initial* state of the system may be taken to be when the aircraft is empty

- Conventionally, the initial system state is shown in a schema called *Init*



- Initialisation is often regarded as a dummy operation with no "before" state and, in accordance with this view, it is the "after" versions of system sets which are cited in *Init*

SYSTEM CHANGING OPERATIONS

- If *BoardOK* is an initial version of the operation to allow a person to board the aircraft, then:

BoardOK

ΔAircraft -- include both *Aircraft* and *Aircraft'*
-- schemas which are potentially different

$p? : PERSON$ -- person to board aircraft

$p? \notin onboard$ -- precondition 1

$\#onboard < capacity$ -- precondition 2

$onboard' = onboard \cup \{p?\}$ -- postcondition

  - For the moment we defer consideration of precondition violations

- If *DisembarkOK* is a first version of the operation to allow a person to leave the aircraft:

DisembarkOK

ΔAircraft -- include both *Aircraft* and *Aircraft'*
-- schemas (potentially different)

$p? : PERSON$ -- person to disembark

$p? \in onboard$ -- precondition

$onboard' = onboard \setminus \{p?\}$ -- postcondition

ENQUIRY OPERATIONS

- Enquiry operations leave the system state unchanged and therefore should include the schema $\Xi Aircraft$ (like $\Delta Aircraft$ not normally **explicitly** included in a specification):

$$
\begin{array}{|l}
\hline
\Xi \text{Aircraft} \\
onboard : \mathbb{P}\ \text{PERSON} \\
onboard' : \mathbb{P}\ \text{PERSON} \\
\hline
\#\ onboard \leq capacity \\
\#\ onboard' \leq capacity \\
onboard' = onboard \\
\hline
\end{array}
$$

- One obvious enquiry is to determine the number of people currently on board the aircraft. If this operation is called *Number*, we can write:

$$
\begin{array}{|l}
\hline
\text{Number} \\
\Xi \text{Aircraft} \\
numOnBoard\,! : \mathbb{N} \\
\hline
numOnBoard\,! = \#onboard \\
\hline
\end{array}
$$

ENQUIRY OPERATIONS

- To ascertain whether a specific person is on-board we need to allow for a suitable reply, and to accommodate this we can define a free type YESORNO where: YESORNO ::= yes | no

- If the operation to determine whether a particular person (*p*?, say) is on board is called *OnBoard* we can write either:

OnBoard

ΞAircraft	-- there can be no change of state
p? : PERSON	-- person to check for
reply! : YESORNO	
$(p? \in onboard \land reply! = yes)$	-- found
$\lor$	-- or
$(p? \notin onboard \land reply! = no)$	-- not found

or:

OnBoard

ΞAircraft	-- there can be no change of state
p? : PERSON	-- person to check for
reply! : YESORNO	
$(p? \in onboard \Rightarrow reply! = yes)$	-- found
$(p? \notin onboard \Rightarrow reply! = no)$	-- not found

FAILURE OF PRECONDITIONS

- To accommodate situations where preconditions are **not** satisfied we can define a free-type RESPONSE where:

  RESPONSE ::= OK | already on board | full | not on board | two errors

  and a schema *OKMessage*:

  OKMessage ≙ [feedback!:RESPONSE | feedback! = OK]

- A schema, *BoardError*, to handle 'errors' in the operation which allows a passenger to board the aircraft might be:

BoardError

ΞAircraft	-- *there can be no change of state*
p? : PERSON	-- *person to board aircraft*
feedback! : RESPONSE	-- *feedback on operation*

(p?∈onboard ∧ #onboard=capacity ∧ feedback!=two errors)
∨
(p?∈onboard ∧ #onboard<capacity ∧ feedback!=already on board)
∨
(p?∉onboard ∧ #onboard=capacity ∧ feedback!=full)

FAILURE OF PRECONDITIONS

- A similar schema, *DisembarkError*, to cater for errors in the operation which allows passengers to disembark might be:

DisembarkError

ΞAircraft	-- *there can be no change of state*
p? : PERSON	-- *person to leave aircraft*
feedback! : RESPONSE	-- *feedback on operation*

p? ∉ onboard ∧ feedback! = not on board

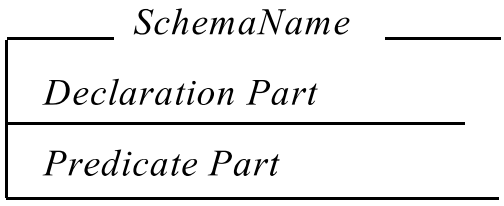
- As we shall see later, schemas may be combined with many of the standard logical operators and, assuming this is possible, we can specify "total" *Board* and *Disembark* operations, which cater for all possible outcomes (including violation of preconditions), as:

TotalBoard ≙ (BoardOK ∧ OK_Message) ∨ BoardError

TotalDisembark ≙
(DisembarkOK ∧ OK_Message) ∨ DisembarkError

SUMMARY OF SYMBOLS

- Vertical Format for a Schema:



- Horizontal (or, Linear) Format for a Schema:

  SchemaName ≙ [*declarations* | *predicate*]

?	*Decoration character (suffix) to denote* ***input***
!	*Decoration character (suffix) to denote* ***output***
′	*Decoration character (superscript) to denote* ***'after states'***
Δ	*Prefix character for names of* ***'before-after'*** *state schemas*
Ξ	*Prefix character for names of* ***'identical before-after'*** *state schemas*

EXERCISES

1. A simple game is based upon the following:

> *A pond may contain any number of fish up to and including a specified maximum number. Conceptually, an angler is fishing the pond with a rod and line. Any fish caught by the angler must be placed in a net which is suspended in the pond.*

Suppose the given set [FISH] represents all possible fish that may ever be in the pond and we declare a global:

$$| \; maxFish : \mathbb{N}$$

to represent the maximum number of fish that the pond can contain at any particular time. Assuming that we are only interested in the pond, the net and the relationship between them, create, based on two sets *pond* and *net*:

(a) a schema, *Fishing*, describing the *state* of the system;
(b) a schema for the operation whereby one fish is caught and placed in the net;
(c) a schema for the operation whereby one or more fish are removed from the net and returned to the pond;
(d) a schema for the operation whereby a number of new fish are added to the pond;
(e) a schema for the operation which reports the number of fish currently free in the pond.

2. A warehouse holds stocks of various items carried by an organization. A computer system records the *level* of all items carried. The computer system also records the *withdrawal* of items from stock and the *delivery* of stock. Occasionally a completely new item will be carried by the warehouse.

If [ITEM] is the set of all items that could possibly ever be carried by the warehouse then the *Warehouse* schema might describe the system state:

Warehouse

$$carried : \mathbb{P}\ ITEM$$
$$level : ITEM \pfun \mathbb{N}$$
$$\textbf{dom}\ level = carried$$

(a) Write down an *Initialize* schema which shows that, initially, there are no items carried by the warehouse.
(b) For a quantity of an item to be withdrawn, the item must be carried and there must be enough stock. Write down a *Withdraw* schema which withdraws a quantity *q*? of an item *i*? from the warehouse (ignore violation of preconditions).
(c) If only deliveries for carried items are accepted, write down a *Deliver* schema for the operation which delivers a quantity *q*? of an item *i*? to the warehouse (ignore violation of preconditions). There is no upper limit on stock held.
(d) A new item must not initially be carried and will initially have a stock level of zero. Write a schema *CarryNewItem* which adds a new item to the warehouse (ignore violation of preconditions).

3. A college provides a multi-user computer system for its students and staff. All staff and students must *register* with the college's IT Services department to be allocated a unique user-identification (*user-id*) before they are allowed access to the computer system. To use the system, each registered user must *log-in* using their *user-id* (no "password" needed). At any given time a registered user will either be *logged-in* or not *logged-in* (it is not possible for a user to be *logged-in* more than once concurrently).

Suppose [PERSON] denotes the set of people who may conceivably be registered for the system at any time, with

logged_in	denoting the people currently logged-in to the system, and
users	denoting those people registered with IT Services.

A global variable, *max_log_in* might denote the greatest number of people that may ever be logged-in concurrently:

$$max\_log\_in : \mathbb{N}$$

If we attempt to model the system using the suggested sets, then, using the declarations above, we might propose the state schema, *Computer*, for the system as specified:

Computer

$$users, logged\_in : \mathbb{P}PERSON$$

$$logged\_in \subseteq users$$
$$\#logged\_in \leq max\_log\_in$$

(a) Can you explain the symbolic expressions in *Computer* using plain English?

(b) Suppose now that this simple system is extended so that anyone logging on to the computer system must supply a password as well as a user-id. When a new user is registered they are allocated a "dummy" password to facilitate initial access to the system. Each user can then change the value of their password once they are logged-in.

To accommodate the extension we may assume the above specification is extended with:

[PASSWORD]	the set of all possible passwords
dummy : PASSWORD	the password for initial access

Use **schema inclusion**, as appropriate, with accompanying explanatory narrative in **plain English**, create schemas (with names as in the parentheses) for:

(i) the state of the extended system (*Secure_Computer*);
(ii) the operation to register a new user (*Secure_Add_User*);
(iii) the *log-in* operation with a correct password supplied (*Secure_Log_In*);
(iv) the operation where a user changes their password (*Change_Password*) - it is to be assumed that both "old" and "new" passwords must be supplied by the user.

In each of the above, you may assume that, for each schema, all preconditions are satisfied (i.e. there is no need to consider "error" situations).

4. Extend the previous question so that the operations in parts (ii), (iii) and (iv) of (b) are made "total" - i.e. consider violation of pre-conditions.

5. The COBOL programming language admits a special kind of data file called an *indexed-sequential* file. In principle, an indexed-sequential file is a sequence of records which may be accessed in any order by specifying the value of a special field (the *key*) in the required record. There is, at most, one record in the file corresponding to any particular value of the key. Operations are available to *read*, *update*, *insert* and *delete* records (using the COBOL instructions READ, REWRITE, WRITE and DELETE respectively). The first steps in an attempt to model the COBOL indexed-sequential filing system follow. At this stage, there is no requirement to include consideration of error conditions.

[KEY]	The set of all possible keys for the file
[DATA]	The remaining fields (apart from the key)

If the *file* is regarded as a function from a *key* to the rest of the *data* in the record, we have the state schema, *IS_File*:

IS_File

$file : KEY \pfun DATA$

Partial schemas for the COBOL file operations are:

Read

Ξ IS_File
k? : KEY
result! : DATA

::::::::::::::::::::::::::::::::

ReWrite

Δ IS_File
k? : KEY
new? : DATA

::::::::::::::::::::::::::::::::

Write

Δ IS_File
k? : KEY
new? : DATA

::::::::::::::::::::::::::::::::

Delete

Δ IS_File
k? : KEY

::::::::::::::::::::::::::::::::

(a) Why is *file* modelled as a **partial** function?

(b) Complete the predicate parts of the partial schemas for the four operations *Read, Rewrite, Write* and *Delete*. Your predicates should be expressed both symbolically and in **plain** English.

6. A college offers many courses. A number of different tutors are responsible for teaching on each course and each of these tutors will teach on two or more courses. A student, who is registered with the college, may be enrolled on only one course at any one time and each course has a limited capacity (i.e. for each course there is a maximum number of permitted student enrolments), which is determined at the time that the course is added to the college portfolio. Tutors are assigned to teach on a course and students can be enrolled for a course only after the course has been added to the portfolio. No course can be removed from the portfolio if it has tutors assigned to it, nor if students are enrolled for it. Suppose we declare the following:

[TUTOR]	is the set of all tutors that might ever be employed by the college;
[COURSE]	is the set of all courses that might ever be offered by the college;
[STUDENT]	is the set of all students that might ever register at the college;
taught_by	describes the connection between COURSE and TUTOR;
has_enrolled	describes the connection between COURSE and STUDENT;
course_limit	describes the connection between COURSE and the set of permitted maximum enrolments;

and suppose we define a state schema *College* to describe the system. Then an initial realization of *College* might be:

College

$$
\begin{array}{l}
known\_students : \mathbb{P}\ STUDENT \\
portfolio : \mathbb{P}\ COURSE \\
tutors : \mathbb{P}\ TUTOR \\
taught\_by : COURSE \leftrightarrow TUTOR \\
has\_enrolled : COURSE \pfun \mathbb{P}\ STUDENT \\
course\_limit : COURSE \pfun \mathbb{N}_1 \\
\hline
\mathbf{dom}\ taught\_by \subseteq portfolio \\
\mathbf{dom}\ has\_enrolled \subseteq portfolio \\
\mathbf{dom}\ course\_limit = portfolio \\
\mathbf{ran}\ taught\_by \subseteq tutors \\
\mathbf{ran}\ has\_enrolled \subseteq known\_students
\end{array}
$$

where
- *known_students* is the set of students currently registered with the college;
- *tutors* is the set of tutors employed to teach on the college courses;
- *portfolio* is the set of courses offered by the college.

(a) Explain, in **plain** English, what you think is the meaning of each of the five lines in the predicate part of the state schema, *College*.

(b) Why are *has_enrolled* and *course_limit* **partial** rather than total functions?

(c) Extend the *College* schema to provide a more complete system description.

(d) Devise a Z schema *Remove_Course* which specifies the operation which deletes a course, *c?*, from the college portfolio. You should include a set of reasonable preconditions; but there is no need to consider violation of any of these preconditions.

7. A university operates a modular degree scheme where students choose a selection of modules from a large menu. The choice is not entirely "free" and registrations for modules are subject to certain constraints. It is important that the administrators keep track of which students are registered for which modules.

[PERSON, MODULE] represent, respectively, the sets of *students* and *modules* that might ever be part of the system.

The ordered pair (p, m) where p is a student and m a module, represents the information that the student p is registered to take module m; so we can represent the entire modular degree scheme by the relation *taking*, where: $taking : PERSON \leftrightarrow MODULE$

The set of students currently enrolled at the university is *students* and the modules currently in the modular degree scheme are in the set *degModules*. Those who are doing modules must be enrolled as students and the modules they are doing must be *bona fide* degree modules at the university. The greatest number of students allowed to take any module is *maxNum* where:

$$maxNum : \mathbb{N}$$

(a) Write a Z expression for the set of students registered for module m.
(b) Write a Z expression for the set of all students who are registered for at least one module which student s is taking.
(c) Define a state schema, *ModuleReg*, for the modular degree system.
(d) Define schemas for the operations of:

(i) enrolling a new student at the university;
(ii) registering a student for a module;
(iii) cancelling a student's enrolment at the university;
(iv) cancelling a student's registration for a module;
(v) adding a new module to the degree scheme;
(vi) removing a module from the degree scheme.