Chapter 1

Introduction

A programming language is a formal notation that you use to tell a computer what to do. A list of instructions to the computer, drawn up to tell the computer to perform a specific task, is called a program. That is, when you write a computer program, you are drawing up a "program of action" for the computer.

A program typically reads some input data, computes with it, and then produces some output data. In simple programs, the input is often typed in on the keyboard by the person using the program (**the user**) and the results are displayed on the screen where it is viewed by the user.

More advanced programs typically obtain input from other sources as well and produce output not only on the screen. By now, you should be familiar with disk files, text editors and compilers. An editor obtains its input from the file you are editing, as well as from the keyboard. It displays its output on the screen and produces a new version of the edit file when you exit from it. Compilers obtain input from files and produce files as well as screen output.

Computer files are called **persistent** because their contents do not disappear when you switch off the computer. In contrast, a character typed in at the keyboard is a transient datum because it disappears from the input stream when it is read in by a program. Likewise, output on the screen disappears when it is overwritten by new output or scrolls off the screen. Also, variables used in computer programs lose their contents when the program finishes with execution. Input data, output data and data stored in program variables are thus called **transient**.

Without persistent data, computers will be of little more use than most pocket calculators. Imagine retyping your entire program every time you wish to compile it.

Different computer programs can also cooperate to perform larger tasks by communicating results via persistent data. For example, the compiler uses the persistent output of the editor as its input.

When different programs cooperate by using and producing persistent data, we call the collection of programs and data a system. For example, we can call program files, an editor and a compiler a program development system. When we wish to be less specific, we speak of a **software** system.

In the introductory programming course the chief concern was to learn the rules of using a programming language to write programs for a computer. The programs written in that course were meant to illustrate the rules rather than to perform truly useful functions.

In this course it is assumed that you know the rules of programming. The chief concern now is to learn how to combine programs into useful software systems. This means that we will be paying a lot of attention to persistent data. For our programming language we shall use Slim because this language makes it easier to instruct a computer than any other language available to us.

The major challenge of the course will be in learning how software systems can be constructed to form part of efficient administrative systems involving people, manual procedures, computers, persistent data and computer programs.

Other programming languages, particularly those that are used in industry, will be studied in Course 3.

Exercises

1.1 Go to the library and find an introductory book on computers. (Hint: use a computer to help you find it.) Scan through it, making lists of the following:

- input devices
- output devices
- storage devices
- software systems.

1.2 Make a list of all the software systems that you have had to deal with up to now.

1.3 Name at least one instance where your life would have been easier if a well constructed software system was in use.

Chapter 2

A review of Expressions

Before starting with a sporting match, one should warm up first. Likewise, before going on to study persistent data in more detail, we'll warm up by reviewing concepts covered in Course 1.

First we'll look at expressions. An expression (or formula) is something you know well from school days, for example $x^2 + 4x - 5$. As you can see from the example, an expression involves **variables** like x, **values** like 2 and **operators** like $+$.

If we agree to call a value or variable by itself an expression, we can generalize to define an expression to be either

- a variable, for example: x
- a value, for example: 2
- an expression of the form: *expression infix-operator expression*, for example: $x + 2$
- an expression of the form: prefix-operator expression, for example: -5.

We can write this definition more concisely in a formal language called EBNF:

expression =
  variable |
  value |
  expression infix-operator expression |
  prefix-operator expression

The major difference between the way you wrote expressions at school and the way you must write expressions in programs is that you must write $x * y$ instead of xy or $x \times y$, $x \,/\, y$ instead of $\frac{x}{y}$ or $x \div y$, and $x * *2$ instead of x^2.

Another thing to bear in mind is that when you calculate the value of an expression, you must do exponentiations before multiplications or divisions, and do the latter before additions and subtractions. That is, `2 + 2 * 2 ** 2` becomes `2 + 2 * 4`, which becomes `2 + 8` which becomes `10`. This is the way humans have agreed to proceed when calculating the value of an expression, and thus this is the way that expressions are calculated by computers when instructed to do so in a language like Slim.

If you need to break with this convention, you use another convention: expressions inside parentheses are evaluated first. Thus `(2 + 3) * 4` becomes `5 * 4` which becomes `20`. To make provision for parentheses, we update our EBNF definition:

expression =
    variable | value | '(' expression ')' |
    expression infix-operator expression |
    prefix-operator expression

2.1 Values

At school, the only values you dealt with were numbers. When programming you will also write expressions involving booleans, strings, lists of values, sets of values and abstractions. The latter kind of value is the main substance of this course and will be discussed at length in later chapters. The other kinds of values are reviewed below.

2.1.1 Numbers

There is not much to say about numbers, other than that you cannot type in a number such as $1,5 \times 10^{-57}$ from an ordinary keyboard exactly as you wrote it at school. Instead, you must type: `1.5e-57`. Also, numbers like 1 234 567 must be written without spaces: `1234567`.

2.1.2 Booleans

The values **true** and **false** are called the Boolean values after George Boole who developed an Algebra involving only these values. They are the result of expressions, such as $x > 1$, which can be either true or false.

2.1.3 Strings

The term **string** is short for *string of characters* and is meant to evoke the picture of a collection of characters strung up like beads on a necklace. Like a string of beads can be considered to be a single entity (a necklace), a string of characters can be considered to be a single entity (a string value).

Since words are also strings of characters and we use words like **or** and **not** to stand for operators, we need to indicate when a string of characters is to stand for a value. In Slim we do this by enclosing the character string between an open quote (`‘`) and a close quote (`’`) character. For example, `‘not’` is a character string consisting of the letters `n`, `o`, and `t`.

This raises the problem of how to write a string containing a close quote character. For example, `‘This string contains the ’ character’` can be mistaken by a simpleton (such as a compiler) for the *string* `‘This string contains the ’` followed by the *word* `character`, followed by a lonely close quote.

To avoid this, we precede every close quote by a backslash (\) inside strings. Similarly, if we wish to include a backslash in a string, we precede it too with a backslash. For example: `‘This string has a backslash (\\) and a close quote (\’)’`

2.1.4 Lists

A list of characters is much like a string of characters. The main difference is that we write the *list* of the letters `n`, `o`, and `t` the form `[‘n’, ‘o’, ‘t’]`, instead of the form `‘not’` used to write the *string* of those letters. While the latter form is more convenient to use when working with lists of characters, the former is more general and allows us to denote lists of other kinds of values. For example, the list of numbers `[1, 2, 3]` and the (shopping) list of strings `[‘sugar’, ‘tomatoes’, ‘milk’]`.

A further generalization offered by the list notation is that we can write lists of expressions. For example, `[i+j, i-j, i*j, i/j]`. Slim also offers shorthand notations such as `[1..5]` instead of `[1, 2, 3, 4, 5]` and `[1..9 by 2]` instead of `[1, 3, 5, 7, 9]`.

It is also possible to write list values that are selections of the values making up another list. For example, if `L` stands for the list `[1..10]` we can write `[all i in L | mod(i, 2) = 0]` to denote the list `[2, 4, 6, 8, 10]`. We can even write `[all i+10 for i in L | mod(i, 2) = 0]` to denote the list `[12, 14, 16, 18, 20]`. A good way to read the above expression aloud is to say "the list of all the values yielded by the expression $i + 10$ where i stands for successive values of L for which mod(i, 2) is equal to 0". The true power of this notation will become clear later on in the course.

The above can be summarized in the following BNF definition:

list = '[]' | '[' expression {',' expression} ']' |
'[' expression '..' expression ['by' expression] ']' |
'[' 'all' [expression 'for'] identifier 'in' expression ['|' condition] ']'

2.1.5 Sets

A set of values is a collection of values, much like a list of values, and is denoted with the same notation except that {} brackets are used instead of [] brackets. There are two important differences between sets and lists:

1. Sets are not ordered collections. It makes no sense to talk about the first, second or last element of a set. Thus, for example: $\{1, 2, 3\} = \{3, 1, 2\} = \{2, 3, 1\} = \{3, 2, 1\}$.

2. Given a set of values and a value, the only thing that matters about them is whether the value is a member of the set or not. It makes no sense to talk about a value belonging to a set twice or thrice. Thus, for example: $\{1\} = \{1, 1\} = \{1, 1, 1\}$.

2.2 Operators

Operators are denoted by symbols like + or words like **intersect**. Infix operators operate on two arguments and are written between the two, for example in the expression $5 + 6$, the infix operator '+' is written in between its two arguments 5 and 6. Prefix operators take a single argument and are written immediately preceding the argument. For example in the expression $-i$ the prefix operator '$-$' is written immediately preceding its argument i.

2.2.1 Numeric operators

$-i$	Yields the negative of i.
$i + j$	Yields the sum of i and j.
$i - j$	Yields the difference of i and j.
$i * j$	Yields the product of i and j.
$i \,/\, j$	Yields the quotient of i divided by j.
$i = j$	Yields **true** if i and j are equal, **false** otherwise.
$i \# j$	Yields **true** if i and j are not equal, **false** otherwise.
$i > j$	Yields **true** if i is greater than j, **false** otherwise.
$i >= j$	Yields **true** if i is greater than or equal to j, **false** otherwise.
$i < j$	Yields **true** if i is less than j, **false** otherwise.
$i <= j$	Yields **true** if i is less than or equal to j, **false** otherwise.

2.2.2 Boolean operators

not b	Yields **true** if b equals **false**, and **false** otherwise.
b **and** c	Yields **true** if b and c are both **true**, and **false** otherwise. If b is **false**, the expression c is not evaluated.
b **or** c	Yields **true** if b or c or both are **true**, and **false** otherwise. If b is **true**, the expression c is not evaluated.

2.2.3 String operators

$s(i)$	Yields the ith character of the string s. If i is greater than the length of the string, or less than 1, $s(i)$ yields the special value **Undefined**.
$s + t$	Yields a string made up of the characters in s followed by the characters in t.
$s = t$	Yields **true** if s and t are equal, **false** otherwise.
$s \# t$	Yields **true** if s and t are not equal, **false** otherwise.
$s > t$	Yields **true** if s is greater than t, **false** otherwise.
$s >= t$	Yields **true** if s is greater than or equal to t, **false** otherwise.
$s < t$	Yields **true** if s is less than t, **false** otherwise.
$s <= t$	Yields **true** if s is less than or equal to t, **false** otherwise.
s **in** t	Yields **true** if s is a substring of t, **false** otherwise.
s **not in** t	Yields **true** if s is not a substring of t, **false** otherwise.

2.2.4 List operators

$L(i)$	Yields the ith value of the list L. If i is greater than the length of the list, or less than 1, $L(i)$ yields the special value **Undefined**.
$L + M$	Yields a list made up of the values in L followed by the values in M.
$L = M$	Yields **true** if L and M are equal, **false** otherwise.
$L \# M$	Yields **true** if L and M are not equal, **false** otherwise.
v **in** M	Yields **true** if v is one of the values making up M, **false** otherwise.
v **not in** M	Yields **true** if v is not one of the values making up M, **false** otherwise.

2.2.5 Set operators

$S + T$	Yields a set made up of the values in S as well as the values in T.
$S - T$	Yields a set made up of only those values in S that are *not* also values in T.
S **intersect** T	Yields a set made up of only those values in S that are *also* values in T.
S **subset** T	Yields **true** if *all* the values in S are *also* values in T.
$S = T$	Yields **true** if S and T are equal, **false** otherwise.
$S \# T$	Yields **true** if S and T are not equal, **false** otherwise.
v **in** S	Yields **true** if v is a value in S, **false** otherwise.
v **not in** S	Yields **true** if v is *not* a value in S, **false** otherwise.

2.3 Variables

If someone asks you what value is yielded by the expression $i + 5$ you cannot answer with a particular number. You could say, for example "It varies, if i represents 1 then the answer is 6, if i represents 2 then the answer is 7, and so on." Or you could say "First tell me what value i represents".

Likewise, a computer instructed by a Slim program to evaluate the expression $i + 5$ needs to be told which value i represents. This can be done by using the assignment instruction:

assignment = identifier ':=' expression

For example, the instruction

```
i := 1
```

followed by the instruction

```
j := i + 5
```

have the effect of instructing the computer to associate the value 1 with i and the value 6 with j.

If, following the previous two instructions, the computer receives the instruction

```
i := 2
```

it will subsequently associate the value 2 with i instead of the value 1, that was used to calculate the value of $i+5$ when carrying out the previous instruction. The value associated with i can thus change (vary) as time progresses and we thus call i a variable to emphasize this. Note that the value associated with j does *not* change when the value associated with i changes. (In other words, j is associated with the value 6, *not* with the expression $i + 5$.)

The variability of the value the computer associates with i has some interesting consequences. For example, if after carrying out the above instructions, the computer is instructed as follows:

```
i := i + 1
```

what value is associated with i after this instruction has been carried out? The answer is 3. To see why, consider the fact that the computer cannot change the value it associates with i until it knows what the new value will be. Thus, the computer *first* calculates the value of $i+1$, getting 3 because prior to this instruction it associated the value 2 with i. The computer then associates the new value with i and will use the value 3 for i when evaluating any subsequent expressions involving i (until it is instructed to change the value associated with i again).

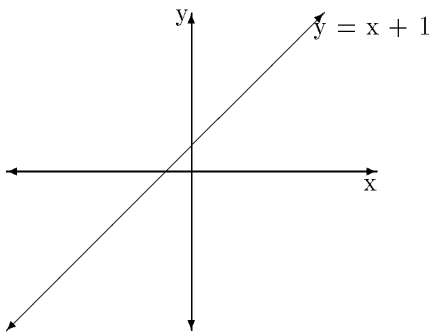
As a convenience for the programmer, Slim allows expressions of the form `i := i + expr` to be written as `i +:= expr` The same applies to `-`, `*` and `/`.

Chapter 3

A review of Functions

3.1 Computed Functions

In high school, functions are usually synonymous with graphs. You may well recall graphs like the following:



In the figure, the line labeled $y = x + 1$ represents the function. The line allows you to determine the value of y for any value of x, by drawing a vertical line through the corresponding position on the x-axis and then drawing a horizontal line through the point where the vertical line crosses the function. The position where the horizontal line crosses the y-axis represents the value of y.

Of course, given the formula $y = x + 1$, you may as well calculate the value of y directly. In fact, you can draw the graph of the function by calculating y for many values of x and then plotting the (x, y) coordinates.

You may also recall seeing the notation $f(x) = x + 1$ used instead of $y = x + 1$. The use of the letter f emphasizes the fact that the expression $x + 1$ is a function and the (x) following the f emphasizes the fact that the function calculates a y value, given an x value. It is usual to refer to x as the **parameter** of f.

You should also have seen the expression $f(1)$ used to refer to the y value of f when $x = 1$. Given that $f(x) = x + 1$, calculating $f(1)$ is easy: Replace all occurrences of x in $x + 1$ with 1 and evaluate the resulting expression. Likewise, given that $g(x, y, z) = x^3 + 5xy^2 - 6yz$, you can calculate $g(1, 2, 3)$ by replacing all occurrences of x in $g(x, y, z) = x^3 + 5xy^2 - 6yz$ with 1, all occurrences of y with 2 and all occurrences of x with 3 and then evaluating the resulting expression.

In Slim, we use the following notation to define functions:

identifier '(' identifier {',' identifier} ')' '=' expression

The following functions are predefined in Slim:

abs(n)	Yields $-n$ if $n < 0$, n otherwise.
sign(i)	Yields -1 if i is negative, 0 when i is 0 and 1 when i is positive.
ceil(i)	Yields the smallest integer greater than or equal to i.
floor(i)	Yields the largest integer smaller than or equal to i.
int(i)	Yields the integer part of i, for example $int(-2.6) = -2$.
round(f)	Yields the integer (whole) number closest to f, for example $\mathrm{round}(-2.4) = -2$.
div(i, j)	Yields the integral number of times j divides into i.
mod(i, j)	Yields $i - j \times \mathrm{floor}(i/j)$.
even(i)	Yields **true** when i is an integer multiple of two, **false** otherwise.
odd(i)	Yields **false** when i is an integer multiple of two, **true** otherwise.
length(L)	Yields the length (number of elements) of list L. Can also be applied to strings.
size(S)	Yields the number of elements in set S.
max(C)	Yields the maximal (largest) element of collection C .
min(C)	Yields the minimal (smallest) element of collection C.
some(C)	Yields a randomly selected element of collection C.
locase(s)	Yields string s with all uppercase letters converted to lowercase.
slice(s, [i..j])	Yields a string consisting of the ith to jth characters of string s.
upcase(s)	Yields string s with all lowercase letters converted to uppercase.
mknum(s)	Yields the number corresponding the string s.
mkstr(x)	Yields a string that corresponds to value x.
fmap(f, C)	Yields a [all f(x) for x in C] or {all f(x) for x in C}, depending on whether C is a list or a set.
reduce(f, C)	Distributes binary operator f between elements of collection C. For example, `reduce('+', [1 2 3]) = 1 + 2 + 3`.
concat(C)	Equivalent to *reduce('+', C)*.
sort(C)	Yields a list consisting of the elements of C, stored in ascending (increasing) order.
sort(C,'>=')	Yields a list consisting of the elements of C, stored in descending order.

Exercises

3.1 Give the Slim equivalent of the following function $f(x, y) = x^2 + 3xy - y^2$. Calculate the values of f(1,2), f(2,1) and f(2,2) without the help of a computer, then use Slim to check your answers.

3.2 Calculate the values of the following expressions yourself, then check your answers using Slim:

```
1. min([9,abs(-8)])
2. max([ceil(3.14), floor(2.71)])
3. min({ceil(0.5), floor(2.9)})
4. int(-3.7) + abs(floor(-3.7))
5. max({abs(-7), 6})
6. div(-7, 3)
7. sign(-3) + sign(4)
8. [all i in [1..10] | odd(i)]
```

3.2 Enumerated functions

Given a set of $(x, f(x))$ pairs, all with distinct x values, you can "calculate" the value of f(x) for a given x simply by finding the $(x, f(x))$ pair with the corresponding x value. This is much like calculating $f(x)$ from a graph by means of drawing vertical and horizontal lines. The major difference is that, in principle, a graph allows you to find $f(x)$ for *any* value of x, whereas a set of $(x, f(x))$ pairs only allows you to find $f(x)$ for a fixed set of x values. If your x value is not in some pair in that set, you cannot find a corresponding value for $f(x)$.

Functions that are defined only for a fixed set of x values are called finite and discontinuous. These functions are too simple to be of much use to mathematicians, but, as will become clear in later chapters, they are very useful to computer scientists.

In Slim, any set of lists of $(x, f(x))$ pairs can be made into a function, using the predefined function *mkmap*. Thus, if

```
f := mkmap({[1,11], [2, 12]})
```

then

```
f(1) = 11 and f(2) = 12.
```

Note that enumerated functions need not involve numbers. For example, if:

```
day_after := mkmap({['Mon', 'Tue'], ['Tue', 'Wed'], ['Wed', 'Thu'],
              ['Thu', 'Fri'], ['Fri', 'Sat'], ['Sat', 'Sun'],
              ['Sun', 'Mon']})
```

then

```
day_after('Mon') = 'Tue' and day_after('Sun') = 'Mon'
```

Note further that the predefined function *domain* will return the set of x values occurring in a set of $(x, f(x))$ pairs and that the expression *fmap(f, domain(f))* will return the set of $f(x)$ values. For example:

```
domain(day_after) = fmap(day_after, domain(day_after)) =
{'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'}
```

Exercises

3.3 Assuming that the following assignment statement has already established a mapping, evaluate the given expressions:

```
M := mkmap({['England', 'London'], ['USA', 'Washington'], ['Japan', 'Tokyo']})
```

1. domain(M);
2. size(domain(M))
3. size(fmap(M, domain(M)))
4. M('USA')
5. M('Tokyo')

Chapter 4

A review of instructions

4.1 Programs

A computer program is a "program of action" for the computer, consisting of one or more instructions to be carried out. These instructions are not carried out randomly, but in a very specific order and the consequences of an instruction usually depends on cumulative consequences of all the instructions carried out prior to it. For example, a program consisting of only

```
put x
```

will display `Undefined` on the screen, whereas

```
x := 'Hello'
put x
```

will display `hello` on the screen.

In general, variables represent 'unknown' values which are either computed from the values of other variables, or have been read in as input data. For example:

```
put 'Enter your name, please:'
get name
put 'Hello' name'!'
```

4.2 Conditional instructions

It is rare for one set of instructions to be applicable in all circumstances. For example, if you are instructed to go to the shop and buy a loaf of white bread, but get there and find that the white bread is all sold out, you are unable to carry out the instruction. Whoever gave you the instruction can probably expect you to adapt to the situation by buying a loaf of brown bread or by going to another shop or by waiting for the bakery to make another delivery.

Computers, however, have no intelligence and cannot adapt automatically to unexpected situations. It is thus necessary to anticipate all eventualities and to give the computer explicit instructions for each of them. The computer is instructed to choose the correct course of action by means of the if instruction. For example:

```
put 'This program computes x/y, given x and y.'
put 'Please enter x:' & get x
put 'Please enter y:' & get y
if y = 0
```

```
  put 'Sorry, 'x'/0 does not make sense!'
else
  put x'/'y '=' x/y
end
```

Note that the if instruction provides the computer with two alternative instructions and a means for deciding which of these to follow. More alternatives can be provided by means of elsif clauses. The format of the if instruction is as follows:

'if' condition ['then']
  {instruction}
{'elsif' condition ['then']
  {instruction}}
['else'
  {instruction}]
'end'

4.3 Repetitive instructions

Computers make up for their lack of intelligence and flexibility by being able to carry out their simple instructions faithfully and very quickly. Computers are thus well suited for carrying out simple tasks again and again, each time for a slightly different purpose.

For example, it would be extremely tedious and time-consuming for a human clerk to write a letter to each student at this university, informing them of their results. However, once we have instructed a computer how to print such a letter for one student, it is a relatively simple matter to get the computer to repeat these instructions, each time printing the letter for another student.

We can instruct the computer to repeat itself by using the for instruction:

'for' identifier 'in' expression ['|' expression] ['do']
  {instructions}
'loop'

For example:

```
for i in [1..3]
  put i
loop
```

will print

```
1
2
3
```

Note that the repeated instruction, `put i`, does something different each time that it is followed because it contains a variable and the variable represents a different value every time the repeated instruction is followed. The optional condition that follows the vertical bar, shown in the syntax, plays the same role as the one appearing in set and list constructors.

In addition to for instructions, Slim also offers while instructions:

'while' condition ['do']
  {instruction}
'loop'

and repeat instructions:

'repeat'
  {instruction}
'until' condition

These instructions cause their nested instructions to be followed repeatedly, checking the condition before/after each repetition and stopping when the condition becomes false/true, respectively.

4.4 Procedures

We've seen that a group of instructions can be followed repeatedly, doing something different each time, because different values are assigned to some of the variables referred to by the instructions.

Quite often, it is useful to give each group of instructions a name and to treat the name as a new kind of instruction. Every time the name of the group of instructions is used where an instruction is expected, the computer carries out all of the instructions in the named group. We call such a group of instructions a **procedure**.

The variables that are referred to in a procedure and which could have different values assigned to them each time the procedure is followed, are similar to the parameters of functions and hence the same notation is used to specify their values when the procedure is followed.

Here is an example of a procedure:

```
proc rectangle(w, h)
  for i in [1..h]
    for j in [1..w]
      put'*'&
    loop
    put ''
  loop
endp
```

Given the instruction

```
rectangle(4, 3)
```

the instructions nested in the procedure body are followed with the value 4 assigned to `w` and the value 3 to `h`, with the result that

```
****
****
****
```

is displayed.

The syntax for procedure definitions are as follows:

'proc' identifier ['(' identifier {',' identifier} ')']
  {instruction}
'endp'

4.5 Procedural functions

Sometimes it is not convenient to calculate the value of a function as a single expression. Slim therefore allows a form of function definition similar to procedures:

> 'func' identifier '(' identifier {',' identifier} ')' '−>' expression
>   {instruction}
> 'endf'

When the value of a procedural function must be calculated for some parameter value, the instructions in the function definition are followed, exactly as for a procedure. The value of the function is whatever value has been assigned to the special variable **result** after the last instruction has been followed. (The expression following the −> specifies the type of this variable.)

For example, the following function converts numbers from the South African string format to the American string format:

```
func RSA_to_USA(num) -> String
  result := ‘’
  for ch in num
    if ch = ‘,’
      result +:= ‘.’
    elsif ch = ‘ ’
      result +:= ‘,’
    else
      result +:= ch
    end
  loop
endf
```

Chapter 5

Data

Computers are often called Data Processing Machines because the programs running on computers essentially consist of operations on data. Before writing a program, it is important to first get a good idea of the data the program must process and what kind of operations can be performed on the data.

The meaning of the word **data** is somewhat vague. In the Pocket Oxford Dictionary it is defined as "facts of any kind". The word **fact**, in turn, is defined as "something that is known to be true". Computers thus perform operations on "things known to be true"!

It seems that an example is called for, so let's consider the persistent data that are processed by the programs making up a computerized university administration system. The facts making up this data, clearly include your name, an address to which your results must be sent, your date of birth, the degree you are registered for and so on.

5.1 Extending Slim

We have seen in the previous chapters that Slim provides operators to manipulate things such as numbers, strings, lists and sets. However, the designers of Slim obviously could not anticipate all the purposes for which we would wish to use Slim to instruct the computer. Not surprisingly, Slim offers no operators to manipulate things such as names, addresses, dates and so on.

Suppose, however, that we are designing a new version of Slim and that we wish to include operators to manipulate names, what kind of operators make sense? Well, names are of little use unless we can display them. However, given that a name has been assigned to variable n, what would you expect to see in response to the instruction `put n`? For instance, if `n` contained the author's name, should you see

`Prof. B. H. Venter`, or
`B H Venter`, or
`Barend Hermanus Venter`, or
`Herman Venter`, or
`Herman`, or
some other variation?

The correct variation obviously depends on what we wish to do with the printed name. Most likely, we would wish to use different formats at different times. It simply does not make sense to choose a particular format at the expense of the others. We will therefore choose to make names 'unprintable' and instead provide a number of operators with which we can construct printable names. For example:

title n	Yields the title of n as a string.
formal_names n	Yields the formal names of n as a list of strings.
informal_name n	Yields the informal name of n as a string.
surname n	Yields the surname of n as a string.

With these operators and the string operators provided by Slim, all of the above print formats for names can be constructed with ease.

Fortunately, there is no need to go to the trouble of extending Slim with the above operators. Slim already allows us to define new functions, so we may as well stick to standard Slim and include function definitions in our programs so that:

title(n)	Yields the title of n as a string.
formal_names(n)	Yields the formal names of n as a list of strings.
informal_name(n)	Yields the informal name of n as a string.
surname(n)	Yields the surname of n as a string.

Exercises

5.1 Construct a suitable expression for each of the above formats. Test your answers by writing a program. Start your Slim program with:

```
import ex51
```

5.2 Suggest some additional functions that will allow formats not listed above.

5.3 Think of some other functions on names that might be useful.

5.2 Abstract Data

It is likely that someone who is very interested in names will wish to use many functions on names that have not been listed above or thought of by yourself. This does not really matter. If we are developing a university administration system, all we care about is to have those functions that are needed in programs making up our system.

In other words, we are deliberately concentrating on only some details about names and are suppressing all the others. The process of suppressing unnecessary details is called **abstraction**. We can also think of the 'name' assigned to `n` as a **model** of the real name. As you know, model cars, model houses and model students are not real cars, houses and students. A model which includes *all* details is not a model but a duplicate. In general, a model includes only those details that are useful for the purpose for which the model was built.

You can thus think about the data recorded about yourself in the university administration system as representing a model of yourself. Because models are abstractions, the data (facts) making up a model are usually referred to as **abstract data**.

5.3 Types

Although the word **value** has not yet been used in this chapter, it should be obvious that the 'name' assigned to `n` is a new type of value. In other words, in addition to numbers, strings, lists and sets, we now also have names.

It is usual to use the term **type** to refer to a set consisting of all the values that are regarded as being of the same type. It is also usual to give a descriptive proper name to a type. Below, the proper names *Number, String, List, Set* and *Name* will be used to refer to the set of all numbers, the set of all strings, the set of all lists, the set of all sets and the set of all names.

One of the interesting things to note about the above types is that they are all disjoint. That is, no element of *Number* is an element of any of the other sets, and likewise for the other sets. There is, however, no reason why types should not intersect. For instance, the set $\{-\infty, \ldots, -1, 0, 1, \ldots, \infty\}$ consists of *all* the whole numbers (integers) and can thus be labelled the type *Integer*. This type clearly intersects with *Number*. In fact, *Integer* is a proper subset of *Number*. A type that is a subset of another type is usually called a **subtype** of the other type.

Two types can also intersect without one being a subtype of the other. For example, in a university administration system, the type *Student* will often intersect the type *Staff*. In other words, the value that represents the computer model of a staff member who is also studying at the university will be a member of both the set of all student values and the set of all staff values.

5.4 Type definitions

In Slim it is the convention to group the functions associated with a type together in a file bearing the same name as the type and to import this file into any program needing to use values of this type. For example, to provide for type *Name*, we have a file named `name.slm` containing the following:

```
export Name

Name : set of Entity := {}
title : map from Name to String
formal_names : map from Name to list of String
informal_name : map from Name to String
surname : map from Name to String
```

5.5 Constructors

Two things should bother you about the example in the previous section: Firstly, why assign the empty set to `Name` when it is supposed to be the set of all names? Secondly, what do the words following the colons signify?

To answer the first question: It is unthinkable to write an assignment statement that assigns all possible names to `Name`. We thus include only those names that are of interest to us in `Name` — initially none. As new names become of interest, they are added to `Name`. To do this, we need to be able to obtain new values of type *Name*. This is done for us by the expression `new(Name)` which yields a new value of type *Name* and includes this new value in the set of values assigned to variable `Name`. For example, after:

```
Name : set of Entity := {}
n : Name := new(Name)
```

we have

```
Name = {n}
```

To anwer the second question: In Slim, the expression following the colon serves to associate a type (a set of permissable values) with a variable. Thus

```
n : Name
```

declares that the variable `n` can only be assigned values which are members of the set of values represented by type *Name*. In other words, not just the names currently of interest (those in the set currently assigned to variable `Name`), but also all names that might be constructed in future.

5.6 Data Functions

The instruction

```
title : map from Name to String
```

declares that the value assigned to variable `title` must be a function which results in a string when applied to a name. Thus, if `n` is a variable to which a name has been assigned, the expression `title(n)` will yield either a string or **Undefined**.

Although it declares that the value of `title` must be a particular kind of function, the above instruction does not **define** the function. In other words, we know that `title(n)` will be a string value, but we do not know which string value corresponds to which name value.

The reason why function *title* (the value of variable `title`) is left undefined, is that we cannot possibly define, once and for all, the title corresponding to each and every name that ever was and will be. Rather, we define the function only for those names that are of actual interest – which happen to be the set of values assigned to variable `Name`. Since the latter is a finite set, function *title* is best defined as a set of $(x, f(x))$ pairs, i.e. an enumerated function. For example:

```
Name : set of Entity := {}
title : map from Name to String
n : Name := new(Name)
title := mkmap({[n, 'Prof.']})
```

Each time a new name value is added to the set assigned to `Name`, the function assigned to `title` must be redefined accordingly. For example, if we add

```
m : Name := new(Name)
```

to the above, we must also add:

```
title := mkmap({[n, 'Prof.'], [m, 'Dr.']})
```

An easier way to redefine function *title* to accomodate the new name, is to use a special form of the assignment statement:

```
title(m) := 'Dr.'
```

5.7 Constructor Functions

To make matters more convenient, we usually package such assignments together in a function. For example:

```
func new_Name(t, f, i, s) -> Name
  result := new(Name)
  title(result) := t
  formal_names(result) := f
  informal_name(result) := i
  surname(result) := s
endf
```

We can then use this function to construct a new *Name* value as follows:

```
n := new_Name('Prof.' ['Barend' 'Hermanus'] 'Herman' 'Venter')
```

Not surprisingly, we call functions such as *new_Date* **constructor functions**.

Exercises

5.4 Devise suitable operators for values of type *Date* and write Slim statements to define this type.

Chapter 6

Modelling

In the previous chapter we saw that the data processed by computer systems can be thought of as a collection of models (abstractions) of entities (things) existing in the real world. For example, a university administration system manipulates models of real students, lecturers, courses, venues, degrees and so on.

The techniques needed to build these models have been covered in the previous chapter and are quite simple. Unfortunately, the models themselves are seldom simple. Furthermore, model building is a creative process that can only be learnt by trial and error. In this chapter, a number of models that might be used in a university administration system are introduced and discussed.

You are urged to spend most of your study time devising your own models, rather than memorizing these examples.

6.1 Students

It is tempting to start off with:

```
Student : set of Entity := {}
name : map from Student to Name
address : map from Student to Address
date_of_birth : map from Student to Date
```

However, students are not likely to be the only type of value for which the function *name* makes sense in a university administration system. It is better to make *name* applicable to any value of type *Person* and to declare *Student* to be a subtype of *Person*. Thus

```
Student : set of Person := {}
address : map from Student to Address
date_of_birth : map from Student to Date
```

Of course, you should also be asking yourself "Are the *address* and *date_of_birth* operators peculiar to students, or should they be applicable to all persons?". This question is not trivial to answer and will be discussed further in the next section. The answer preferred by the author is that these operators should be applicable to all persons. The above model for *Student* is thus reduced to

```
Student : set of Person := {}
```

In searching for further operators, we should be asking ourselves: "What operators make sense for students only?"

By definition, anyone registered for a degree is a student. So we can add:

```
degree_registered_for : map from Student to Degree
```

However, a number of questions immediately present themselves:

1. Is it necessary to introduce *Degree* as a new type?

2. What about students who are not studying toward degrees?

3. What about previous registrations?

The usual strategy for asking whether we need a new type of value is to ask whether such values model things that are of interest to us in their own right. In other words, are there functions, specific to values of type *Degree*, that we would be interested in applying to such values? We shall assume the answer is yes.

To answer the second question we need to know more about the underlying reality which we are trying to model. For example, are the students that are not studying towards degrees so few that we can accept a result of **Undefined** (or perhaps a special "No degree" degree value) whenever *degree_registered_for* is applied to one of these students? Alternatively, do we need to distinguish among several different subtypes of *Student*, such as *Degree_students* and *Diploma_students*?

Another solution, which is the one we shall adopt, is to change the model so that the question becomes irrelevant. We can do this by generalizing the concept of *Degree* into *Qualification* and by requiring every student to study towards some sort of qualification. We thus replace the above with:

```
qualification_registered_for : map from Student to Qualification
```

As for the last question, previous registrations are of interest. However, these can be recorded as part of an academic record.

It should be clear by now, that virtually every aspect of the model of a student raises a number of questions for which there are no "right" answers. This is why, despite the simplicity of model descriptions, modelling is such a difficult task. Below, a "complete" model of a student is given without further discussion. You are urged to devise a series of questions about this model and to raise them in class.

```
Student : set of Person := {}
student_number : map from Student to Integer
qualification_registered_for : map from Student to Qualification
courses_registered_for : map from Student to set of Course
academic_record : map from Student to Academic_record
fees_due : map from Student to Number
fees_received : map from Student to Number
address_for_account : map from Student to Address
```

6.2 People

A university administration system has to keep track of quite a number of different types of people (students, academics, administrators, service employees, parents, donors, alumni and so on). Obviously, the type of details kept about one person may differ substantially from the type of details kept about another person.

In the previous section the question arose whether a detail like the date of birth should be regarded as part of the standard model of a person, or whether it should be included only in the models of subtypes like *Student*. Clearly, the university may not be able to obtain a date of birth for every type of person it keeps track off. For instance, some donors might object to being asked to supply their dates of birth to the university. On the other hand, any donor would probably appreciate receiving a birthday card! So, if the date of birth is known for some or other reason, it may as well be recorded.

In general, many of the details that present themselves for inclusion in a model are **optional**. For maximum flexibility, the model should make provision for these details. However, doing so means accepting that the corresponding operators can sometimes result in **Undefined**.

Using a value like **Undefined** to signify the absence of a detail is not without problems. For instance, any expression involving the subexpression *date_of_birth(p)* would have to provide for the possibility that the result of *date_of_birth(p)* could be **Undefined**. Failure to provide for **Undefined** can be expected to be a frequent source of programming errors. Consequently, many people prefer to sacrifice flexibility in order to make their programs smaller and less error prone.

One way to retain flexibility while reducing the possibility of errors is to identify subtypes for which details that are optional in the supertype, will always be required (**mandatory**). For instance, if *date_of_birth* is mandatory for *Student* then *date_of_birth(s)* is known not to result in **Undefined** for any student *s*.

Our notation for describing models does not make specific provision for distinguishing among optional and mandatory details. Rather, a generalized method for describing constraints can be used to identify mandatory details. By default, details are optional. For example, in the model given below:

```
Person : set of Entity := {}
name : map from Person to Name
sex : map from Person to {'Male', 'Female'}
marital_status : map from Person to {'Single', 'Married', 'Divorced'}
next_of_kin : map from Person to Person
date_of_birth : map from Person to Date
address : map from Person to Address
work_telephone : map from Person to String
home_telephone : map from Person to String
further_information : map from Person to list of String

inv name(p) # Undefined forall p in Person
```

the condition following `inv` must invariably be true. This has the effect of ensuring that a model of type *Person* always includes a name.

Likewise, we can ensure that *date_of_birth* is mandatory for all students, by including

```
inv date_of_birth(s) # Undefined forall s in Student
```

in the description of the model of a student.

6.3 Addresses

Addresses are surprisingly complex to model. For example, we could model addresses as follows:

```
Address : set of Entity := {}
Room_number : map from Address to String
Floor_number : map from Address to String
Building_name : map from Address to String
Street_number : map from Address to String
Street : map from Address to String
Area : map from Address to String
```

```
City : map from Address to String
Region : map from Address to String
Country : map from Address to String
Postal_code : map from Address to String
```

This model includes all the details the author has ever encountered in a residential address and also allows for “numbers” like LV24. However, the model is clearly not suitable for addresses like the following:

Department of Computer Science
University of Fort Hare
Private Bag X1314
Alice, Ciskei

Furthermore, if you look at the address appearing on a typical company letterhead, you could also encounter other forms of addresses like telex address, telegram address, phone number, fax number and electronic mail address.

This poses us with the question of whether we should restrict our notion of an address to something that can be acted upon by the postal service when it appears on the front of an envelope, or whether we should generalize the definition of address to something like the following "data that allows one to locate or communicate with an entity".

If we adopt the latter definition, we need to change our definition of *Person* by deleting *address* and adding:

```
addresses : map from Person to set of Address
```

since a typical person will have several different addresses.

In practice, this approach is too general. Most designers opt for the simpler (but less flexible) approach taken in the previous section (providing separately for postal/residential addresses and telephone numbers).

The problem of providing for addresses that do not fit the model presented at the start of this section is usually handled by defining *Address* as folllows:

```
Address = list of String
```

Because this definition imposes very little structure on an address, any kind of address can be accommodated. On the other hand, because there is so little structure, not much can be done with an address other than printing or displaying it. We shall assume that these two operations are the only really important ones for our university administration system.

Exercises

6.1 Design a model for postal addresses which preserves as much structure as possible, while providing as much flexibility as possible. (Hint: You have to identify elements which are common to *all* postal addresses.)

6.4 Academic record

The academic record of a model second year computer science student would look something like this:

```
1992 Matric (Highbrow High -- JMB)
  First Language    A
  Second Language   A
  Third Language    A
  Computer Studies  A
  Mathematics       A
  Science           A
  Biology           A

1993 B.Sc. 1 (UFH)
  Computer Science 1    A
```

```
  Mathematics 1         A
  Numerical Mathematics A
  Physics 1             A
```

This can be modelled as follows

```
Academic_record = set of Ac_rec_for_one_year
Ac_rec_for_one_year : set of Entity := {}
year : map from Ac_rec_for_one_year to Integer
qualification : map from Ac_rec_for_one_year to Qualification
year_of_study : map from Ac_rec_for_one_year to Integer
institution : map from Ac_rec_for_one_year to String
results : map from Ac_rec_for_one_year to set of Result
Result : set of Entity := {}
course : map from Result to Course
symbol : map from Result to {'A'..'F'}
```

6.5 Other models

The next three models are given without further comment, save to say that they are oversimplified in order to keep the overall complexity of this chapter manageable. You are urged to analyze them with a view to criticizing them in class.

```
Qualification : set of Entity := {}
abbreviation : map from Qualification to String
full_name : map from Qualification to String
type : map from Qualification to {'Degree', 'Diploma', 'Certificate'}
usual_duration : map from Qualification to Integer
awarded_by : map from Qualification to String

Course : set of Entity := {}
code : map from Course to String
title : map from Course to String
lecture_group : map from Course to {'A'..'G'}
lecturer : map from Course to Person
attendees : map from Course to set of Student
venue : map from Course to Venue

Venue : set of Entity := {}
building : map from Venue to String
room : map from Venue to String
capacity : map from Venue to Integer
administrator : map from Venue to Person
facilities : map from Venue to list of String
```

Chapter 7

Retrieval

The major reason for storing data in a computer is to use the computer as an aid in retrieving the data. To appreciate the extent of the aid a computer can provide, imagine registering students without the use of computers and then writing out class lists for all courses, given only a stack of registration forms.

It is not possible to do such a task efficiently unless the need for it has been anticipated and the registration process has been designed accordingly. Furthermore, more work will have to be done during registration than is otherwise required. This extra work is not only costly, if performed by humans, but boring and error-prone.

In the examples below, it is assumed that the necessary Slim statements have been executed to set up data according to the models discussed in the previous chapter.

7.1 Selection

Most of the time, we need to retrieve details of all entities x such that some condition $C(x)$ holds. For instance, we may wish to retrieve details about all the students that were first registered in 1990. The set of such students can be obtained as follows in Slim:

```
students_90 :=
  {all s in Student | div(student_number(s), 100000) = 90}
```

Given this set, we can print out the necessary details by using a for loop. For example, we can print out the student numbers as follows:

```
for s in students_90
  put student_number(s)
loop
```

Occasionally, we are interested only in a particular value. Such values can be retrieved by constructing a list with that value as only member, and then selecting its first and only element. For example:

```
CS2 := first([all c in Course | code(c) = 'COSC 2'])
```

When you use such statements, it is a good idea to check if the set contains exactly one element and to generate some error report if it does not.

Exercises

7.1 Write an expression that results in the set of Computer Science 2 students. You can assume that `CS2` has been defined as above.

7.2 Assign the member of *Qualification* that models the B.Sc. degree to the variable `BSc`.

7.2 Computed details

In the previous section, if we had the following:

```
for s in students_90
  put name(s)
loop
```

The data printed out will not be very useful since `name(s)` returns an "unprintable" value. Obviously, it would be more convenient if `name(s)` returned a string value such as `'Mr. S.T. Udent'`. On the other hand, storing a string instead of an abstract value is less flexible.

Fortunately, it is possible to have the best of both worlds by introducing computed details. For example, if we add

```
sname(p) = title(name(p))+' '+initials(name(p))+' '+surname(name(p))
```

to the Slim statements defining type *Person*, and

```
initials(n) = concat([all fn(1)+"." for fn in formal_names(n)])
```

to the Slim statements defining type *Name*, then we can use `sname(s)` when we need the name of s as a string in "standard" format and `name(s)` when we need the name of s as an abstract value.

Because `sname(s)` is computed from `name(s)`, there is no need to supply its value as a parameter when constructing a new person value. Likewise, there is no need to supply the initials when constructing a new name.

Exercises

7.3 Define a computed detail `s_ac_rec` that returns the academic record of student s as a list of strings.

7.4 Examine the models presented in the previous chapter. Are there any details listed there that could be computed rather than data?

7.3 Sorting

One of the properties of a set is that its members are not ordered in any particular way. Thus

```
for s in students_90
  put sname(s)
loop
```

will produce a list of names in no particular order.

Long lists of names are of little use if the names do not appear in alphabetical order (to see why, consider how useful a telephone directory would be if its entries do not appear in any particular order).

The process of arranging a collection of values into a particular order is called **sorting**. Sorting is accomplished by calling the function *sort* with a collection of values as its first parameter and the desired ordering as its second parameter. Since it is impossible to draw up a list of all the orders a programmer might ever desire, the latter parameter must be supplied in the form of a function of two parameters that returns **true** when the value of the first actual parameter precedes the value of the second in the desired ordering and **false** otherwise. For example

```
by_name(x,y) =
  surname(name(x))+initials(name(x)) <= surname(name(y))+initials(name(y))

for s in sort(students_90, by_name)
  put sname(s)
loop
```

Exercises

7.5 Rewrite the above example so that students appear in reverse alphabetical order.

7.4 Quantifiers

In Slim, the expression `C(x) forsome x in S` corresponds to the conventional mathematical notation, $\exists x \in S$ **such that** $C(x)$. One may also think of this expression as a convenient shorthand for `{all x in S | C(x)} # {}`. The symbol $\exists$ is usually referred to as the **existential quantifier**, hence the title of this section.

The existential quantifier is particularly useful in retrieval problems that contain the word "some", or can be reworded to contain it. For example, if we are to retrieve the set of all courses that are taken by B.Sc. students, we can reword the request to the following: "find the courses for which it is true that *some* of the attendees are registered for B.Sc." The retrieval can then be expressed as a set constructor:

```
{all c in Course |
  qualification_registered_for(s) = BSc forsome s in attendees(c)}
```

If we were to change the above retrieval request to "find the courses for which it is true that *all* of the attendees are registered for B.Sc." we need to use the **universal quantifier**. The mathematical symbol is $\forall$ and the Slim keyword is `forall`. The above expression thus becomes:

```
{c : c in Course |
  qualification_registered_for(s) = BSc forall s in attendees(c)}
```

and should result in a subset of the set constructed with `forsome`.

Exercises

7.6 Write an expression that will result in the set of those students whose academic records include at least one A symbol.

7.7 Repeat the above, but restrict it to those students whose records include *only* A symbols.

7.5 Indices

You should already be familiar with one sort of index: the alphabetical list of words and phrases at the end of textbooks, giving the page numbers where these words and phrases appear in the text. Such a table of (key, result) pairs is, of course, nothing other than an enumerated function.

In general, any enumerated function that allows one to find a particular item (such as a page number) from a large collection, given a value associated with that item (such as a word appearing on the page with that number), is called an index. In other words, if we had an enumerated function

```
course_for_code : map from String to Course
```

such that

```
code(course_for_code('COSC 2')) = 'COSC 2'
```

and likewise for all the other codes, then we would call *course_for_code* an index. Indices make retrieval requests more convenient to program, for example

```
CS2 := course_for_code('COSC 2')
```

instead of

```
CS2 := first([all c in Course | code(c) = 'COSC 2'])
```

Another important advantage is that indices require less computation to be done. In the above example, the first instruction can be carried out by looking the result up in a table, whereas the second instruction requires *every* element of *Course* to be examined (unless the compiler is clever enough to use an index anyway).

Exercises

7.8 Describe an appropriate index for *Qualification*.

An index need not always return a single entity as its result. For example, `courses_for_lecture_group('F')` should result in a set of courses (including `course_for_code('COSC 2')`) and the index should be described as follows:

```
courses_for_lecture_group : map from String to set of Course
```

Nor is it necessary for an index to map from values that can be obtained directly from the instances returned. For example we are likely to define index

```
persons_with_surname : map from String to set of Person
```

even though the surname of a person cannot be obtained directly by an expression such as `surname(p)` where $p \in Person$.

If no single value will suffice to identify a unique instance of an entity, we can define indices that map a combination of details onto instances. For example

```
venues_for_building : map from String to set of Venue
```

but

```
venue_for_building_and_number : map from String, String to Venue
```

In other words

```
venue_for_building('Science Lecture Block')
```

will result in the set of all the lecture venues in this block, while

```
venue_for_building_and_number('Science Lecture Block', 'NE2')
```

will result in the venue representing this lecture room only.

Exercises

7.9 Describe all the indices for *Person* that should be useful in typical retrieval requests.

7.6 Privacy

The data stored in a typical computer system is often of such a nature that it should only be retrieved under controlled circumstances. For example, given a choice, many people will avoid having accounts at a bank that allows anyone to see how much money they have on deposit. Likewise, many people object to their salaries being public knowledge. Closer to home, many students object to their academic records being public knowledge.

One way to control the access people have to data, is to restrict the people that can run the programs that retrieve the data. For example, one could set up the operating system so that only users from the exams office can run the program that retrieves academic records. This is approximately the current policy at Fort Hare.

Such simple controls can be overly restrictive, however. For example, it may be helpful for lecturers to access the academic records of students currently enrolled in their courses. There also do not seem to be any particularly good reasons why students should not be able to inspect their own academic records. In general, therefore, retrieval programs must be able to provide users with access to selected portions of the database.

To do this, a program must be able to establish the identity of the person using it. If possible, this should be done by querying the operating system. If not, the program should enter a dialogue to establish and authenticate the identity of its user. The login procedure required by the Novell operating system is an example of how this can be done.

In general, the rules for determining who gets to retrieve what, can be quite complicated and provision must be made for them in the data model. For example, in order to restrict a student to retrieving only his own academic record, it must be possible to find out which is his academic record. To do this, the program must be able to obtain a value of type *Person*, given the authenticated user identifier of the student. This requires the model for *Person* to be extended with a detail recording the user identifier this person will use when running programs.

Exercises

7.10 Why not add the user identifier detail to the model for *Student*?

7.11 Is this detail optional or mandatory?

7.12 Is the model outlined in chapter 6 sufficient to allow programs to distinguish among lecturers and staff working in the exams office?

7.7 Statistics

It is often useful to summarize data rather than merely retrieving selected parts of it. Summarized data can help one to gain a better understanding of events and can help one to predict the likely consequences of changes.

Perhaps the most basic statistical technique is **counting**. For example, university administrators usually like to know how many students are registered at the university. Other useful counts are the number of staff members, number of courses, number of faculties, number of qualifications awarded, and so on. To retrieve a count, we need only construct a list of the values modelling the things we wish to count and then apply the *length* function to the list.

Exercises

7.13 List 10 types of things, other than those listed above, that you could count, given the data models described in chapter 6.

7.14 Write an expression that results in the number of lecturers currently lecturing courses.

7.15 Write an expression that results in the number of venues that offer slide projectors.

Counts are often more meaningful when combined with other counts in the form of **ratios**. For example, dividing the total number of students with the total number of lecturers gives one a number which can be used to compare the staffing level of this university with other universities. Ratios between 0 and 1 often arise when we divide the count of a subset by the count of the full set, for example dividing the number of venues with slide projectors, by the number of venues. Such ratios are usually multiplied by 100 and called **percentages**.

Exercises

7.16 Write a function *perc(m,n)* that returns the ratio of m and n as a percentage. Use rounding to return an integer rather than a real number.

7.17 Calculate the percentage of venues offering overhead projectors that *also* offer slide projectors. (Hint: divide the number of venues offering both, by the number of venues offering slide projectors.)

It can also be useful to add up lists of numbers. For example,

```
sum([all capacity(v) : v in Venue])
```

gives the maximum number of students that can be seated in lecturing halls at any one time. We also combine such **totals** with other totals and counts to form ratios and **averages**. For example, the total amount of fees received divided by the total amount of fees due currently gives a ratio distressingly smaller than 1, whereas the total amount of fees due, divided by the number of students gives the average amount owed by each student.

Exercises

7.18 Write an expression to compute the total number of attendees of all courses in group 'F'.

7.19 Compute the ratio of the above total and the total number of lecture venue seats.

7.20 Write a program to print a list of lecture groups and their corresponding attendees/capacity ratios.

7.21 Compute the average of the above list of ratios.

There are many other statistical techniques that can be applied to data, but it is not the purpose of this course to give you a grounding in statistics.

Chapter 8

Changes

There are a number of reasons why it is necessary to make changes to the data stored in a computer. The most important is that the data represent models of real entities like students. When the real entity changes in a way that is of interest, the model must be changed accordingly. For example, if a student drops a course, the definition of function *courses_registered_for* must be changed so that it no longer returns that course when applied to that student.

8.1 Enumerated versus computed functions

Only enumerated functions store data. Computed functions merely return the data stored in enumerated functions, in more convenient forms. Enumerated functions can be updated by means of assignment statements. For example, when student s drops course c, the definition of *courses_registered_for* can be changed as follows:

```
courses_registered_for(s) := courses_registered_for(s) - {c}
```

However, *courses_registered_for* need not be an enumerated function. It could be computed as follows:

```
courses_registered_for(s) = {all c in Course | s in attendees(c)}
```

In the later case, *courses_registered_for* will automatically return the right answer once function *attendees* has been changed.

Of course, it is also possible that both *courses_registered_for* and *attendees* will be enumerated functions, rather than computed functions. Storing both functions uses more storage space in the computer, but evaluating an enumerated function is usually quicker than evaluating a computed function. In this case, it is an error to change only *courses_registered_for* or only *attendees*.

Obviously, writing programs would be simple if we could change models in a uniform manner, whether or not the model details are computed or enumerated functions. It is thus advisable to augment models with procedures that will change their details in the appropriate manner. For example, we could add a procedure `drop_course` to the model for a student and define it in one of the following three ways:

```
proc drop_course(s, c)
  courses_registered_for(s) -:= {c}
endp
```

or

```
proc drop_course(s, c)
  attendees(c) -:= {s}
endp
```

or

```
proc drop_course(s, c)
  courses_registered_for(s) -:= {c}
  attendees(c) -:= {s}
endp
```

depending on whether *courses_registered_for*, or *attendees*, or both are enumerated.

Exercises

8.1 Write three versions of procedure **drop_student** that changes the definition of *attendees*.

8.2 Make a list of procedures suitable for changing the model of a student. (Hint: your list is not complete unless every detail can be changed.)

8.2 Changing indices

Indices are enumerated functions that store data that can already be obtained from other functions. In principle, any index can thus be implemented as a computed function. For example,

```
course_for_code(co) = first([all c in Course | code(c) = co])
```

Obviously, enumerated indices must always give the same results as their computed equivalents. It is thus necessary to change indices when changing indexed details. For example, the procedure for changing the code of a course should be defined as follows:

```
proc change_code(c, new_code)
  course_for_code(code(c)) := Undefined
  code(c) := new_code
  course_for_code(code(c)) := c
endp
```

In other words, the index must not only be changed to return c when applied to the *new* code, but must also be changed to no longer return c when applied to the *old* code.

The above procedure will keep the enumerated version of *change_code* consistent with its computed version, only if no two courses ever share the same code. This requirement should be made explicit by adding the following invariant constraint to the model:

```
inv code(c) # code(d) forall d in Course-{c} forall c in Course
```

Exercises

8.3 Write a procedure that will allow us to change the lecture group of a course, while keeping the enumerated version of the index *courses_for_lecture_group* consistent with its computed version.

8.4 Write a procedure that will allow us to change the name of a student, while keeping all indices correct.

8.5 List all the reasons you can think of why we would need to change the name of a student.

8.3 Security

It is of little use to store data in a computer if it cannot be trusted when retrieved. For example, if any student is able to run a program that can change his or her academic record, it is inevitable that some students would wish to change their records to contain better symbols. It is equally inevitable that people will start distrusting the academic records retrieved from the computer.

It is thus necessary to take measures that will retain the credibility of stored data. The most important and effective of these measures is to make it difficult for anyone to run a program that can change stored data, unless specifically authorized to do so. The techniques used to implement this measure are the same as those discussed in the section on privacy in the previous chapter.

However, even authorized persons can make invalid changes to stored data. This can happen when an unauthorized person succeeds in posing as an authorized person or when an authorized person becomes corrupt. Of course, it can also happen because the authorized simply makes a mistake. The latter case is hopefully the more frequent.

What cannot be prevented, must be detected. It is thus wise to keep a log of all changes made to stored data, recording the identity of the person running the program, the date and time the change is made, the justification for the change, the details of the change and enough information to undo the change if necessary. From time to time, a skilled person of high integrity, called an auditor, will use this log to look for invalid changes to the stored data. The log is thus often called an audit trail. Naturally, making changes to the audit trail should be as difficult as possible and as detectable as possible.

Exercises

8.6 Make a list of all the unauthorized changes that people might be tempted to make if anyone could change any of the data stored in a computer running a university administration system.

8.4 Transactions

Quite often, several changes go together. For example, if an autobank dispenses some cash to a customer, two things must change in one step: the balance of the customer and the amount of cash in the autobank. If the customer's balance is reduced, but no cash is dispensed, the customer will be most unhappy. And if cash is dispensed while the customer's balance is left unchanged, the bank will be unhappy.

A group of changes to data that must *all* be made, if *any* are made, is called a transaction. A transaction can be left incomplete if the program carrying out the changes stops executing for some reason, such as a power failure or hardware failure.

When the failure is corrected and operation of the system is resumed, any changes made by incomplete transactions must be undone. The details of how this can be achieved will be considered at a later stage. For now, it is sufficient to note that it is important to identify the groups of changes that constitute transactions.

Exercises

8.7 In the previous sections, some of the procedures contain groups of changes. Are they transactions?