Chapter 46 - Declarative programming

46.1 Declarative languages

As with other paradigms, you will not fully understand the fundamentals of declarative programming unless you get some experience using this type of language. You will need to put aside an afternoon. Download a PROLOG compiler from the 'Download' section at: **http://www.swi-prolog.org/** When you have downloaded the file, simply double-click on it to install the program. You can then do the work in this chapter or search online for a beginner's tutorial.

A declarative language is very different from a procedural one. It is written in a completely different way. With procedural languages, you have to tell the computer exactly what to do. With declarative languages, you state a list of facts and a list of rules that tell the computer how the facts are related. When you want to run a program, you simply state a goal, what you want to find out. The program then starts at the beginning and searches through its facts using the rules in the program. It will follow a route through the facts until it finds the answer to the query. As the program threads through the facts in the program, however, it may come to a dead-end. If this happens, the program simply **backtracks** and tries to find another route that will lead to the answer. It doesn't actually matter in what order the facts in the program are stated or indeed the rules. The program will start at the beginning and find the best route, backtracking as necessary. We have said that languages like PROLOG are known as 'fifth generation languages'. They are ideally suited to the production of 'expert systems'. These systems have a knowledge base made up of facts and the rules that relate those facts. The knowledge base can then be queried. The user would state goals and input data, and the facts in the knowledge base are then searched using the relationships defined by the rules. The facts are gradually 'filtered' until just a few facts remain. These can then be displayed to the user.

46.2 An example of a PROLOG program

You will write programs in PROLOG using a text editor and then run them in PROLOG. Open Notepad or another text editor. Type in the following facts about cats about the relationships between them.

```
/*These are the facts*/
parent(patch,spot).
parent(patch,fred).
parent(patch,tommy).
parent(fuddy,spot).
parent(fuddy,janet).
parent(fang,tommy).
parent(tommy,sammy).
parent(mandy,sammy).
parent(max,fang).
parent(jill,patch).
male(patch).
male(fred).
male(tommy).
male(max).
female(spot).
female(fang).
female(fuddy).
female(sammy).
female(mandy).
female(janet).
female(jill).

/*These are the rules*/
mother(A,B):-
        parent(A,B),
        female(A).

grandmother(A,B):-
        parent(C,B),
        mother(A,C).
```

The mother rule above can be read as:

A is the mother of B if:

A is the parent of B and
A is female.

The grandmother can be read as:

A is the grandmother of B if:

C is the parent of B and
A is the mother of C.

If you installed PROLOG in a directory called '**prolog**', then you will have a subdirectory called '**bin'**. Save this file in **bin** and call it **cat.pl** Also in bin, you will see a file called **plwin.exe** – double-click on it to open PROLOG. At the prompt, type in:

? – [cat].

(Don't forget the full stop and use square brackets.) This will compile the program. If you have stored your program in another directory, use FILE - CONSULT to browse to it. If your program has compiled correctly, you should see this on the screen:

% cat compiled 0.00 sec, 2,704 bytes
Yes
?-

Now you can set yourself some **'goals'**. For example, to find all of the female cats type:

female(A). (Don't forget the full stop!) Press the **RETURN** key on your keyboard and you should get:
A= spot

If you then press the **semi-colon** key on the keyboard, you should get the next female cat, and the next and finally the word NO to signal that there aren't any more in the database. It should end up looking like this:

?- [cat].
% cat compiled 0.00 sec, 2,704 bytes
Yes
?- female(A).
A = spot ;
A = fang ;
A = fuddy ;
A = sammy ;
A = mandy ;
A = janet ;
A = jill ;
No
?-

If you want to find Tommy's parents, then you would type: **parent(A,tommy).** Your screen should look like this if you have done everything correctly:

?- parent(A,tommy).
A = patch ;
A = fang ;
No
?-

If you want to find Spot's grandmother, then you would type: **grandmother(A,spot).** If you have done this correctly, your screen should display:

?- grandmother(A,spot).
A = jill ;
No
?-

46.3 Instantiation

If you search for male(A) in the cats program, then the goal for PROLOG is to find all instances of male. PROLOG searches its facts and finds 'patch'. We say that A is **instantiated** to patch. PROLOG outputs: **A = patch**. PROLOG then continues. It finds that A is also instantiated to fred, tommy and max. The total output is:

?- male(A).
A = patch ;
A = fred ;
A = tommy ;
A = max ;
No

46.4 Goals, predicates, arguments and backtracking

Consider the goal **mother(A,tommy)**. 'Mother' is known as a **'predicate'**, as are parent, male, female, father, grandmother, grandfather and so on. The values inside the brackets are known as 'arguments'.

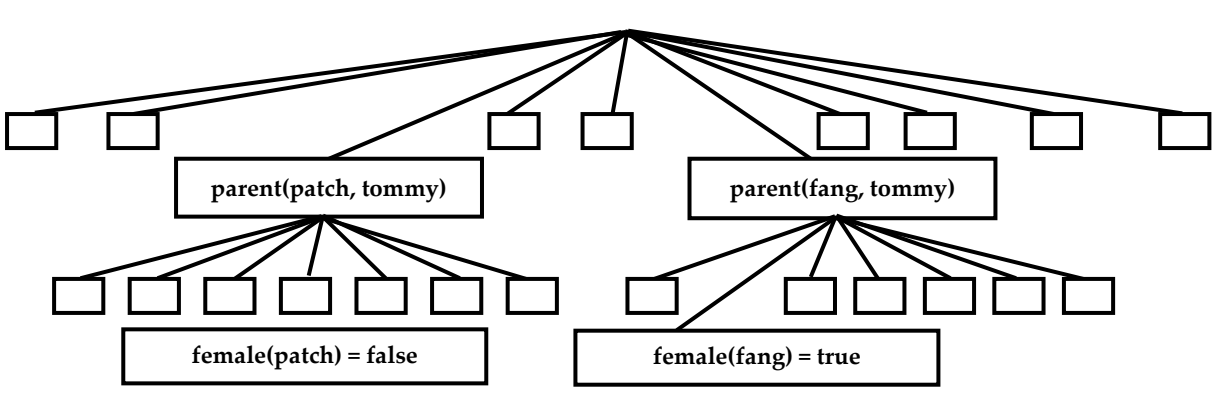
- The **goal** is mother(A,tommy).
- The **predicate** mother has a rule: mother(A,B) :- parent(A,B), female(A).
- tommy and A are the **arguments** of the goal mother(A,tommy).
- PROLOG searches the database for the first part of the rule: parent(A, tommy).
- It finds A is **instantiated** to patch.
- PROLOG checks patch to see if it meets the second rule. It checks female(patch). This fails however!
- PROLOG now **backtracks** to where it found parent(patch,tommy).
- It continues its search from that point and finds that A is **instantiated** to fang.
- PROLOG checks fang to see if female(fang) is true.
- Female(fang) is true so it displays it.
- It continues searching for parent (A,tommy) but finds no other match so ends.

46.5 An alternative way of describing backtracking

An alternative way of describing backtracking is to draw a proof tree. Remember:

mother(A,B) :-

parent(A,B),
female(A). So for this goal, we can draw the following:



A proof tree.

This is what is happening in the proof tree:

- PROLOG starts by trying to solve parent(A, tommy).
- It starts at the top of the list of facts and works its way down the list.
- PROLOG finds the first match with the third parent it checks, with parent(patch, tommy).
- A is instantiated to patch.
- Because it has a match, PROLOG tests the second part of the mother definition using patch i.e. female(patch).
- There are seven female facts to check. Female(patch) is returned as false, however, because patch is not defined as female.
- After this attempt, PROLOG retraces its steps right back up to the top of the proof tree to look for more instances of parent(A, tommy).
- It finds that A is instantiated to fang with the sixth parent it checks, parent(fang, tommy).
- PROLOG checks to see if female(fang) is true and finds that it is with the second female fact.
- It displays that A is instantiated to fang i.e. fang is the mother of tommy.
- It backtracks up the tree to parent(fang, tommy).
- Then PROLOG checks for female(fang) amongst any of the remaining five females, just in case! It doesn't find any more matches.
- It backtracks again right up to the top of the tree this time.
- PROLOG checks the remaining four parent facts for parent(A, tommy) but finds no other matches, so ends.

46.6 Another type of goal we could set

So far, we have set goals of the form **mother(A,tommy)**. This means, '**Who is the mother of tommy?**'

Another type of goal we could set is **mother(fang,tommy)**. This means '**Is fang the mother of tommy?**'

There are two possible outcomes to this type of goal, TRUE and FALSE. Sometimes YES / NO are returned by the program - it depends upon which PROLOG application you are using.

Type in the goal **mother(fang,tommy).** (Don't forget the full stop!) This should return **TRUE** because:

- parent(fang,tommy) returns TRUE and
- female(fang) returns TRUE.

Type in the goal **mother(patch,tommy).** (Don't forget the full stop!) This should return **FALSE** because:

- parent(patch,tommy) returns TRUE but
- female(patch) returns FALSE.

Another goal might be **male(mandy)**. This would return FALSE. Can you work out why?
Another goal might be **male(max)**. This would return TRUE. Can you work out why?

Q1. What are the characteristics of a declarative language?
Q2. Define a 'fifth generation language'.
Q3. What is an 'expert system'?
Q4. What are the four components of an expert system?
Q5. Describe how a mechanic would use a car diagnostic expert system to find an engine fault.
Q6. Why should expert systems be regularly updated?
Q7. What is a 'predicate'?
Q8. Describe with an example what instantiation means.
Q9. Describe what is meant by backtracking.
Q10. Draw a proof tree to demonstrate your understanding of backtracking.

Chapter 47 - Low-level languages

47.1 Low-level Languages

Some programming tasks require a programmer to program in a way that reflects the CPU design. The programmer may need, for example, to directly manipulate memory addresses or the CPU's registers. This is often true when a device driver for a piece of hardware is being written or when a translator is being written for a high level language. It would be appropriate in these cases to use a language that has instructions designed to easily carry out these tasks. High-level languages generally would not be the first choice, because their focus is more on solving application-based problems. The first choice would be assembly languages. These types of languages have instruction sets that reflect the way the CPU carries out its instructions, including instructions that allow you to manipulate a CPU's registers. Their instructions are very close to machine code itself. These languages use **mnemonics**, which are reasonably easily remembered codes for instructions. Examples include:

- DIV 18 ---- Divide the contents of the accumulator by 18 and store the result in the accumulator.
- LDA (3000) ---- Get the contents of address 3000 and move them to the accumulator.
- SUB (5000) ---- Get the contents of address 5000 and subtract them from the accumulator. Store the result in the accumulator.
- MOV B @100 ---- Get an address held in address 100. Go to that address and get the contents. Move them into accumulator B.

Note that decimal numbers are used in my examples throughout this section for simplification. Using hex might be an easier option! Can you think why? Note that in the above examples, each instruction has an **instruction part** and an **address part** to it. For example, in the instruction LDA (3000), the LDA is the instruction part of the instruction and the 3000 is the address part of the instruction. The address part refers to an address where the CPU needs to look in RAM to find some data. This is a very common format for assembler instructions (although a few instructions differ from this format).

- The instruction part of an assembler instruction is usually known as the operation code, or **opcode** for short.
- The address part of the instruction is usually known as the **operand**.

We can represent this general format of most assembler instructions like this: **<OPCODE><OPERAND>** Now you might think that to refer to a piece of data in RAM, you simply give the address in RAM where the data can be found. Unfortunately, it is not quite as simple as that! There are a number of different ways that an assembler language can refer to any particular RAM address. There are different ways because each method has some particular strengths in a particular set of circumstances. The four methods we will discuss are immediate addressing, direct addressing, indirect addressing and indexed addressing.

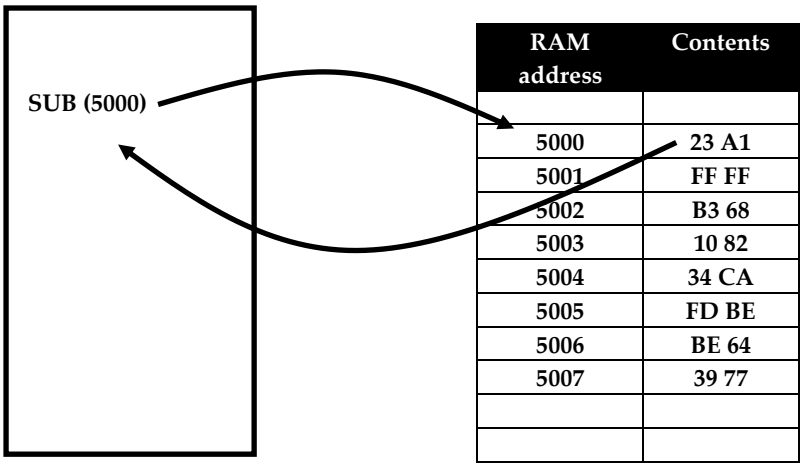
47.2 Immediate addressing

We have said that an assembler instruction usually follows the format <OPCODE><OPERAND>, with the operand containing an address. In immediate addressing, the operand part of the instruction doesn't contain an address! It contains a piece of actual data instead. The CPU doesn't need to waste time by going to a RAM address and fetching a piece of data because it is already there, as part of the instruction! This is a very useful form of addressing if you need to use a **constant**. For example, if you needed to divide a number by the number of months in a year, then you could get and use the number 12 using immediate addressing, because the number of months in a year will never change. Consider the mnemonic : **DIV 18.** We know that this is an assembler instruction that uses immediate addressing because of the form it takes. It has an opcode followed simply by the decimal number 18 (use the bit pattern for the decimal number 18) and nothing else. If I had written the instruction as DIV #18, I would need to use the bit pattern for the hex number 18.

47.3 Direct addressing (also known as absolute addressing)

This is so called because the instruction being worked on, for example, ADD, SUB and so on refers **directly** to the memory location where the data for the instruction can be found! It sounds a mouthful, but you have already seen some examples of direct addressing.

We saw the instruction SUB (5000) earlier. The instruction SUB contains a reference to the RAM memory location 5000 where the data it needs for the instruction can be found. Note the difference between SUB 5000 and SUB (5000). SUB 5000 is an example of **immediate addressing** whereas SUB (5000) is an example of **direct addressing**. The brackets are important!



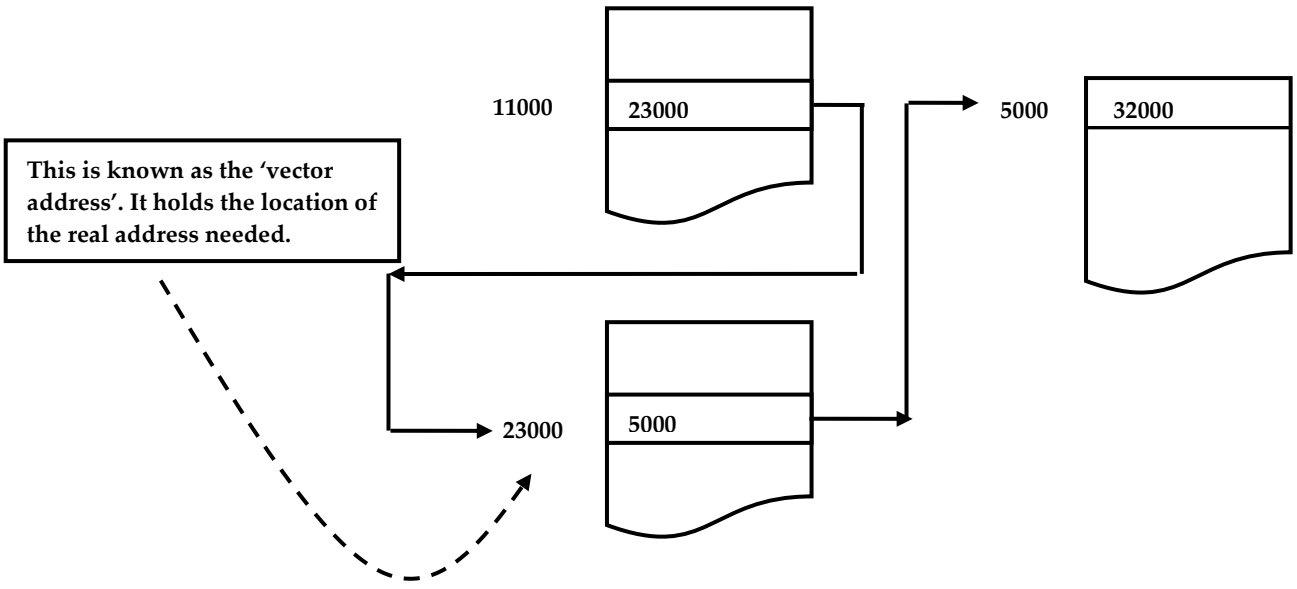
Direct addressing.

Using direct addressing, you can write programs that have instructions that refer to data found in specific memory locations. In the instruction, you have actually said where the data is to be found. The instruction SUB (5000) tells you to go to memory location 5000 to find the data to use in the SUB instruction. Direct addressing is simple to code compared to some other methods such as indirect addressing and can be useful technique. For example, you may need to access a constantly changing running total. Direct addressing would allow you to do this.

47.4 Indirect addressing

Direct addressing is fine as long as you know what memory locations contain the data you need. But suppose you need to use a library program that's in memory. How can you access that program, if you don't know where in RAM the loader has put it? Indirect addressing is one way to solve this kind of problem. To see how indirect addressing works, follow this example. Imagine the instruction: **MOV A, @11000** (The @ symbol is being used simply to show we are using indirect addressing).

Also, assume that RAM location 11000 contains the value 23000, and RAM location 23000 contains the value 5000. Memory location 5000 holds 32000.



When the CPU carries out this instruction, it:

- goes to memory location 11000
- and then gets the value contained in that location, i.e. 23000
- and then goes to memory location 23000
- and gets the value contained in that memory location, i.e. 5000
- goes to memory location 5000
- and in memory location 5000, it finds the value to use, 32000. This is loaded into the accumulator.

47.4.1 Why indirect addressing is very important

Consider this problem. A programmer wants to use a pre-written library procedure. They want to refer to it in their own program. The problem is, they don't know where in memory the loader will actually put the procedure. For example, a programmer is writing a program that needs to call a library routine called PrintDocument. When the library program is compiled, however, the loader, which is responsible for putting the library program in RAM, could put it anywhere! And if our programmer doesn't know where it is, how can she jump to it in her program? What she does know, however, is that there is a special area of RAM that holds memory locations (called vectors) that point to where library routines will be. She has the Specification of these vectors and on this Specification will be the vector for the PrintDocument program. So in her program, she can now use indirect addressing to get access to the PrintDocument program simply by using the vector address. When the library program is compiled, it will be the job of the 'loader' to keep the addresses held in the vectors up-to-date, so that programs that jump to the vectors can be re-directed to the correct place in RAM.

47.4.2 An analogy of indirect addressing

Each day when you arrive for school, you must go to a room for your first lesson. The problem is that your timetable is not fixed! The school puts you in a different room every single day. Assume that it does this because the teachers only decide what exciting activities you will be doing (and what facilities you need) the night before, after you have gone home. Now if the teachers don't decide this until the previous evening, and you never know from one day to the next which room you'll be in, how will you find out where to go when you arrive in the morning? It is very unlikely that you will see the teacher before lessons start! What you and your teacher can do is to use a notice board. Each evening, once the decision has been made about which classroom you will be in, the teacher puts a notice on the notice board telling which room to go to. In the morning, although you don't know which room to go to straightaway, you know a place where you can go to find out i.e. the notice board. This is a wonderful system because by using the notice board, the teacher has complete and total flexibility about which room to use. You, of course, will never miss a lesson because, even though you don't ever know where your first lesson is when you arrive at school, you know where to go to find out. The notice board is like a vector address. If you go to the 'vector address', you will find some information about where your first lesson is.

47.5 Indexed addressing

Indexed addressing is very often used to access numbers held in arrays. The programmer needs to set up an 'index'. This is simply a variable that they can manipulate. To use this addressing method, you state a **base address**. You then add to it the value held in the index. This gives you a new address. You then go to this address to get the data! For example, suppose you have a simple array that starts at address 1001. You can set the base address to 1000. Then if you want the data from the sixth location, for example, you add 6 (the index) to the base address to get 1006 and then jump to that location to get the data. If you want the data from the ninth location in the array, then you add 9 to the base address, and jump to location 1009. What is the point of this addressing system? It is a relatively quick and easy job to alter the index register. You can therefore access large chunks of data easily. For example, if you had to get all the values held in an array, you would get the base address, set up an index and then get the data from the address pointed to by (base address + index). Of course, you would need to set up a loop and increment the index in each loop until the end of the file is reached, like this:

```
GET BASE address
MAKE INDEX=1
WHILE the index does not point to the end of the array
BEGIN
        ADD BASE to INDEX
        READ data
        INCREMENT INDEX
END.
```

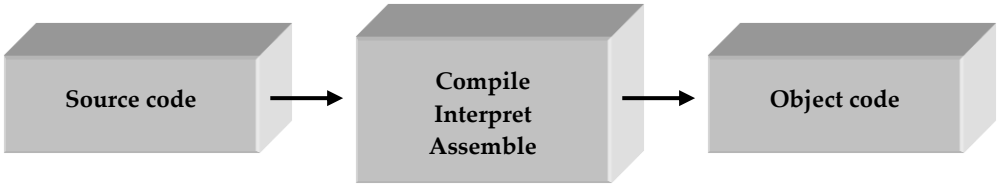
This method is also very convenient if you want to store your array in different places in RAM. The only thing that you would actually need to change is the base address. For example, if you now want to run the program with the data starting in 2001 and not 1001, you just need to change the base address in the program. All the other addresses in the array are worked out relative to the base address. The advantage of this is that you don't have to make lots of changes to your program - only the base address needs to be changed.

Q1. What is meant by a low-level language?
Q2. How does immediate addressing differ from direct addressing?
Q3. Using a diagram, explain how indirect addressing works.
Q4. Using an example, explain why indirect addressing is useful.
Q5. Write some pseudo-code to illustrate how you can read a block of data using indexed addressing.

Chapter 48 - BNF and syntax diagrams

48.1 Backus Naur Form

When you write a program, each instruction you use must be constructed according to strict rules. You must pay great attention to getting the syntax of each instruction perfect or you will not be able to translate your program! Once you have written the program in the programming language of your choice, you need to translate it into code that the computer understands. The process can be represented by the following diagram:



Source code must be translated before it can be used.

48.2 Compilation

We have seen that to run a program, you must turn the source code into object code. We have already seen in an earlier chapter that translation can be split up into three sections:

48.2.1 The lexical analysis stage

1) The source code is looked at and any unnecessary parts of it are removed e.g. comments and spaces.
2) The code is then grouped into tokens. A token can be:

- a keyword like FOR, PRINT and so on
- a symbol that has got a fixed meaning, for example, +, *
- numerical constants
- user-defined variable names.

48.2.2 The syntax analysis stage

The syntax of the string of tokens created in the lexical analysis stage is checked to see if they obey the syntax rules for the language. The entire stream of tokens is split up (or 'parsed') into expressions and then these are checked against a database of syntax rules. There are a number of different ways to represent syntax definitions, such as using Backus Naur Form (BNF) or syntax diagrams. We are about to see these ways in action!

48.2.3 Code generation and optimisation

Code suitable for running on a CPU is generated and then checked for optimisation in this stage.

48.3 Backus Naur Form (BNF)

Once the source code has been turned into a string of tokens in the lexical analysis stage, how does the computer check the tokens, to ensure that the rules for the language have been followed? This is where BNF comes in!

48.3.1 How are syntax rules represented, so that syntax analysis can take place during compilation?

There are three basic symbols you need to know:

::= This means 'is defined by'	**| This means 'Or'**	**< > This means 'The meta variable'.**

- So, a ::= b means 'a is defined by b'.
- A ::= b | c means 'a is defined by b or c'
- < num > ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 means 'meta variable num is defined by 1 or 2 or 3 etc up to 0'.

This last definition simply means that the variable num can either be a 1 symbol or a 2 symbol or 3 and so on. The 1, 2, 3 etc in the last example have a special name. They are called '**terminal symbols**' because they cannot be broken down any further.

48.3.2 An example using BNF

How can you define a positive integer? A positive integer could be a number of any length. E.g. the following numbers are positive integers: 2, 243, 7199, 23244, 12222346 and so on. It can be done as follows:

< integer > ::= < digit >
< digit > ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

Write out these definitions in English!

But this will only allow you to represent **single-digit** numbers. How can you represent integers of **any** length? It can be done using a technique that has its roots in **recursion**. Consider the number 1524.

This is a digit (1) followed by an integer (524).
But (524) is a digit (5) followed by an integer (24).
But (24) is a digit (2) followed by an integer (4).
But (4) is a single digit.

< integer > ::= < digit >< integer >

< digit >followed by< digit >< integer >

< digit >< digit >followed by< digit >< integer >

< digit >< digit >< digit >followed by< digit >

An integer is defined as a digit followed by an integer. The recursion will stop when the final integer is a single digit. This is why we need the OR symbol in BNF, to enable us to drop out of the recursion. The final definition of an integer is:

< integer > ::= < digit > | < digit >< integer >
< digit > ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

Write out these definitions in English.

This definition will allow us to define any ***unsigned*** integer. Perhaps a better name for the meta variable would be **unsigned_integer** so we now have:

< unsigned_integer > ::= < digit > | < digit >< unsigned_integer >
< digit > ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

Suppose we wanted to represent **signed** integers, i.e. an integer with either a plus or a minus sign in front of it. How could we do this? There are a number of ways but one way is as follows:

< signed_integer > ::= < sign><digit > | < digit >< signed_integer >
< digit > ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
< sign > ::= + | -

We simply defined a new meta variable called 'sign'. Then we included it in our definition of signed_integer.

48.3.3 Another example

Consider this definition of a membership ID:

< code > ::= # | &
< digit > ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
< letter > ::= A | B | C | D | E
< number > ::= < digit > | < digit >< number >
< group > ::= < letter > | < letter >< group >
< membership_ID> ::= < code >< group >< number >

We started building up this definition of membership_number by defining the meta variables that didn't use recursion because they are the simplest ones to deal with. Then we defined the more complex meta variables, the ones that did use recursion. Finally, we created a meta variable for the membership ID from the meta variables we just created.

- **&BAEE427** is a valid membership ID.
- **#D56177354** is a valid membership ID.
- **&EEE8** is a valid ID.
- **#AAEF52** is not a valid ID because F is not a valid letter.
- **#7233** is not a valid ID because you must have a letter from A to E after the # or &.
- **&aC3655** is not valid because a small 'a' is not allowed according to the definition of letter.
- **&BBE593433** is not valid because we are not allowed the digit 9.
- **CE67** is not valid because it doesn't begin with # or &.

Now test each other. Create membership IDs and see if a friend can tell you correctly whether they are valid or not.

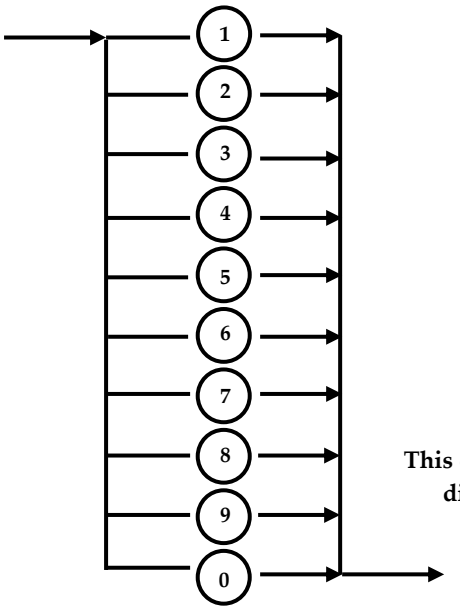
48.4 Syntax diagrams

Another way of representing the syntax of keywords, instructions and the elements of a programming language is to use syntax diagrams. These are possibly more straightforward to understand than BNF notation. They are very useful to the programmer because they can be used to ensure that the correct syntax of instructions and declarations, for example, are used.

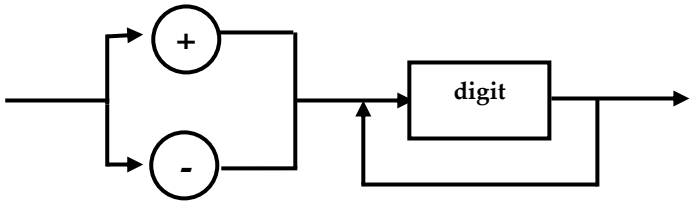
Consider the BNF definition of a signed integer. This is an unsigned integer with either a plus or a minus in front of it. **There are different ways of solving this** but one way is as follows:

<unsigned integer> ::= <digit> | <digit><unsigned integer>
<digit> ::= 1|2|3|4|5|6|7|8|9|0
<signed integer> ::= +<unsigned integer> | -<unsigned integer>

A signed integer could also be represented using a syntax diagram. Again, there are different ways of representing the same logic. Here is one way, using 2 diagrams.



This diagram is the syntax diagram for a digit.



This diagram is the syntax diagram for a signed integer.

48.4.1 How do you read a syntax diagram?

Always start reading the diagram from the left and follow the flow to the right! You cannot go back on yourself unless there is an arrow that lets you! The first diagram defines a digit. Starting from the left, you can flow to the right through only one of a number of paths, and then out again. So, for example, you may flow through the digit 7 and then out again. You cannot backtrack because there isn't an arrow to take you back. Thus, a digit is made up of only one number.

The second diagram again must be read from left to right. You have a choice how a signed integer can begin, either with a plus or a minus sign. Then it needs a digit. From the definition in the first diagram, a digit is made up of one symbol. After one symbol has been selected, the signed integer can finish, or it can loop back on itself (because there is an arrow that lets it do this) and collect another digit, and so on, until the signed integer is finished.

48.4.2 Terminal symbols and rectangular boxes

Some further points to note. Some symbols are in circles, like 0, 1, 2, + and so on. These are **terminal symbols** and cannot be broken down further by reference to other syntax diagrams. Other symbols such as digit are in a rectangular box. That means that you need to refer to another syntax diagram to get the full definition. This ensures that complicated syntax diagrams can be produced which aren't too cluttered.

48.5 IF – THEN – ELSE example

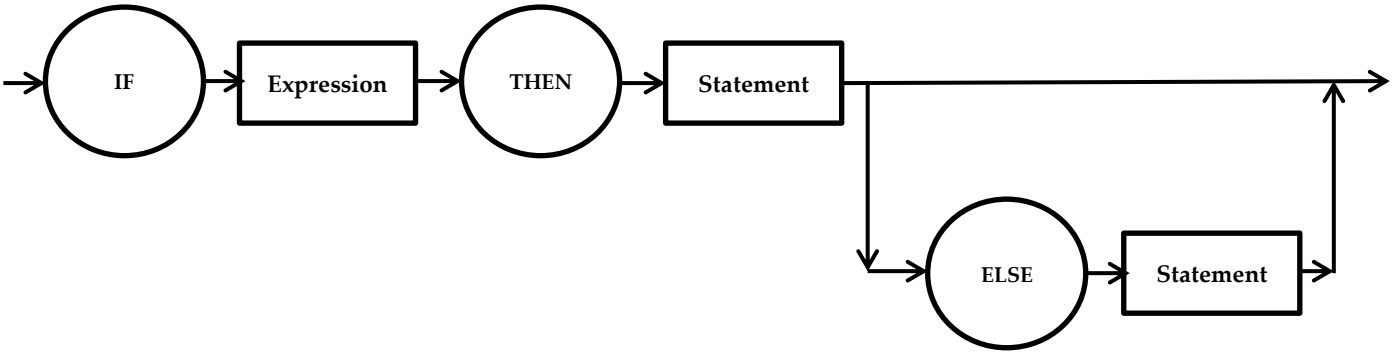
In the programming language Pascal, you could use an IF construction in the following way:

```
IF score >= 80 then
        writeln('Pass with honours')
```

or you might use an ELSE in the code, like this:

```
IF score >= 70 then
        writeln('Pass with honours')
ELSE
        IF score >= 60 then
                writeln('Pass')
        ELSE
                writeln('Fail');
        {ENDIF}
{ENDIF}
```

We can show the syntax diagram for an IF statement or IF – THEN – ELSE statement like this:



'Expression' and 'Statement' are in rectangular boxes so they have their own definition, which we would need to refer to. IF, THEN and ELSE cannot be broken down any further so they are shown in circles.

Q1. What is the British Computer Society's definition of syntax?
Q2. Define 'source code' and 'object code'.
Q3. Describe briefly the differences between compiled code and interpreted code.
Q4. What are the stages involved in translating source code into object code?
Q5. Define using BNF notation the meta variable 'Small_letter' (as in the letter of the alphabet).
Q6. Define using BNF notation the meta variable 'HexSymbol', where HexSymbol is a single hex digit.
Q7. Define using BNF notation the meta variable 'HexNumber', where HexNumber is a hex number made up of any number of hex symbols.
Q8. Draw the syntax diagram for a hex symbol.
Q9. Draw the syntax diagram for a hex number.
Q10. Use the Internet. Find a source of syntax diagrams for whatever language you have been using.

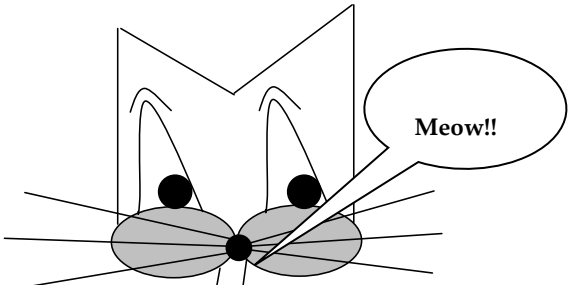
Chapter 49 - An introduction to database design

49.1 Introduction - What is a relational database and why bother?

The NoKats dog club, formed in 1973, currently keeps a record of its members and the dogs they own on little index cards (an example of which is shown below). The cards are kept together by the secretary and can be used for reference, e.g. to inform members of a show. This was an excellent system for the first few decades of the dog club when membership was around 100. In the last few years, however, membership has grown to over 5000. This is proving a real headache for the secretary.

Customer Record for the NoKats Dog Club

Record number: _____
Initial: _____
Surname: _____
Title: _____
Sex: _____
Postcode: _____
Telephone no: _____



Dog details:

Name	Sex	DofB	Breed	Origin of breed	Breed life expectancy

Index card for the NoKats Dog Club.

49.2 Problems for the secretary

1) It takes a long time to find an individual member's record because there are now over 5000 of them. This is especially true if you search using multiple search criteria.
2) It takes even longer to find a dog's record, especially if a member's details aren't known for some reason.
3) Finding all dogs who meet certain criteria, e.g. all poodles born after a certain date takes a long time manually.
4) Occasionally, all of the records need to be sorted into a different order. This can take a long time.
5) When records are filled in, some details are recorded over and over again. For example, a spaniel, its origins and its life expectancy are recorded in a number of different records. This is clearly a waste of the secretary's time. It would be far better if the details for each breed could be stored just once.
6) Details for personalised letters and address labels have to be manually transferred over to the letters and labels from the index cards. This is another time-consuming job.
7) When new records are entered, spelling mistakes or incorrect details are sometimes entered onto the cards.

49.3 Why a computerised database would help the secretary

1) When new records are entered into the database using a data input form, validation rules could trap some errors.

2) All records can be easily sorted by different fields and multiple fields. A 'field' is an attribute of a thing or person, a piece of information held in a record. In this case, fields include Record number, Initial, Surname, Breed, Origin of breed and so on.
3) The secretary can easily and quickly search the entire database for records that meet certain criteria.
4) The results of sorting and searching can be presented in well-designed reports.
5) Details held in the database can be merged into a word processor to produce personalised letters and labels.

49.4 What the dog club needs to think about when setting up a database

- Buying computer equipment is initially expensive. Savings to the dog club may not materialise for years.
- Computer equipment can break down. Who will repair it? How will the dog club function without a computer?
- The secretary may need to be trained and retrained.
- The secretary may find using computers a stressful experience if he or she does not have an IT background.
- How will software problems be dealt with? Does anyone in the dog club have the necessary skills?
- Computerised equipment is attractive to thieves. Measures will need to be taken to protect the hardware and software.
- The data in the system will be very valuable. A backup procedure will need to be put in place.
- The Data Protection Act will need to be complied with.

A simple flat file database has been created. A 'flat file' means that all the records are held in one table. Each row in the table corresponds to one record. Each column in the table corresponds to a different field. Tables are two-dimensional structures. They are therefore also referred to as 'flat', hence the name 'flat file'.

ID	Init	Surname	Title	Postcode	Tel	Dog Name	Gen	DofB	Breed	Origin	L/E Years
1	A	Fish	Mrs	CV35QW	02476111111	Bongo	M	21/08/99	Poodle	China	5
1	A	Fish	Mrs	CV35QW	02476111111	Hiccup	F	08/08/98	Poodle	China	5
1	A	Fish	Mrs	CV35QW	02476111111	Rizla	F	09/09/00	Poodle	China	5
1	A	Fish	Mrs	CV35QW	02476111111	Gov	F	11/01/01	Alsatian	Germany	10
2	C	Here	Mrs	CV27RF	01788222222						
3	D	Lapidated	Mr	CV14RR	02476333333	Manic	M	11/01/01	Poodle	China	5
3	D	Lapidated	Mr	CV14RR	02476333333	Blip	F	02/02/01	Spaniel	France	7
4	X	Ray	Ms	CV12YY	02476444444						
5	Y	Nott	Mr	CV24TT	01788555555	Ruff	M	08/08/00	Poodle	China	5
5	Y	Nott	Mr	CV24TT	01788555555	Addi	M	10/02/00	Poodle	China	5
5	Y	Nott	Mr	CV24TT	01788555555	Catnip	F	10/03/99	Poodle	China	5
5	Y	Nott	Mr	CV24TT	01788555555	Emmi	F	11/03/01	Poodle	China	5
5	Y	Nott	Mr	CV24TT	01788555555	Gov	F	11/01/01	Alsatian	Germany	10

Sample records in a flat file for the NoKats Dog Club.

49.5 The good points about flat files

1) Flat files are relatively quick and easy to set up and use.
2) They are ideal for smaller databases.
3) They provide many of the sorting and searching tools commonly needed by users of the database.

49.6 The problems with flat files

1) You can see that in the flat file, every member has an ID number. This should be unique for each member and is known as the Primary Key. The problem is that some records (some rows) have the *same* ID number. This means that you cannot pick out one and only one record if you searched for a member by their membership number.
2) Dogs do not have their own unique identity number. It is therefore impossible to find an individual dog. Consider Gov in the above flat file. There are different two dogs, both called 'Gov'. Their personal details just happen to be the same! You cannot tell them apart from the details held in the flat file.
3) Because dogs do not have their own ID number, you cannot enter in a 'new' breed until a member owns a dog of that breed.
4) Lots of details are held over and over again, for example, the origins and life expectancy of poodles, and the details about the member known as 'A fish'. This is known as '**data redundancy**' and is a waste of (hard disk)

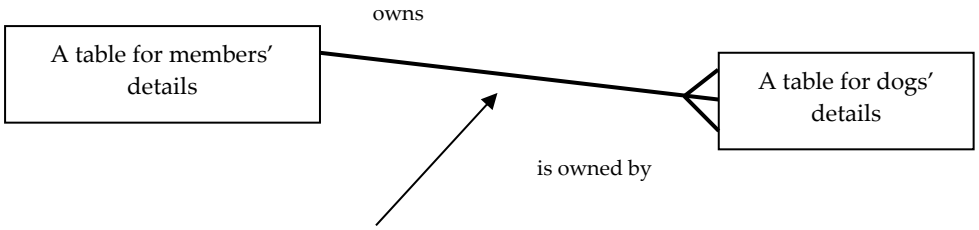
space. Data redundancy also contributes to larger files, which means longer search times. In addition, if you enter the *same* data over and over again in different places in the flat file, you are more likely to make a mistake when entering in one of the entries. This results in some of your data being **inconsistent**.

5) If 'A Fish' got married and changed her name to 'A Haddock', then the secretary would have to make four changes to the database rather than just one. This is clearly a waste of time. The secretary could also introduce some data inconsistencies if he or she changed the name in one of A Haddock's records to A Hadock (with one 'd'). This is referred to as an 'amendment anomaly'.
6) If one member such as 'Y Nott' left the club, the secretary would have to delete five records (not just one). This is sometimes referred to as a 'deletion anomaly'.
7) If one new member joined with more than one dog, you would have to store more than one record (each one holding details about the member) - even though only one member has joined! We will call this an 'adding anomaly'.

To summarise, flat flies are easy and quick to set up. They are easy to use and are ideal for small databases. They have a series of problems. These relate to the repetition of data stored in the database (known as '**data redundancy**') as well as problems to do with adding and removing records or changing the data in records. These are referred to as 'adding, deleting and amendment anomalies' and can result in the data in the records becoming **inconsistent**.

49.7 Improving the flat file design

Clearly, there are problems with flat file databases. However, there is something that can be done about it! Having two smaller tables is no different to having one big table, as long as we link them (or 'relate' them - hence the term 'relational database'). Consider one record in a table. We can have one big record in one table, or split the big record into two parts and link them together (using a special link called a 'foreign key'). One way to split the table is to have a table for members and a table for dogs. We can draw a diagram of this (called an E-R diagram, or Entity-Relationship diagram). 'Entity' is just a name for a table, so this diagram shows the relationship between entities, or tables. In the diagram below, we have a 'one-to-many relationship'.



There are different ways that records in two tables can be related. The most common way is that they are related in a 'one-to-many' relationship. The line that joins the two boxes in this example is the line used to describe this type of relationship. To correctly describe it, you must read it in both directions, like this:

- Each member can own many dogs.
- Each dog is owned by one member.

We say that these tables are linked in a 'one-to-many relationship'. The members table is on the 'one' side of the relationship whereas the dogs table is on the 'many' side.

There is a lot more advice about how to produce, read and interpret E-R diagrams in a later chapter.

It is a very good idea to put some real or made-up records into the tables you have identified as part of your E-R analysis. It really helps you to 'visualise' the problem. Note that when you split up a table into two or more smaller tables, the first thing you have to do is to check that each of the tables you have got has a primary key, a field whose value for every single record (i.e. every single row) will be unique - no other record will have the same value. The members' table has such a field (ID) but the dogs' table doesn't have a suitable one. One simple solution (but not the only one) is to add an ID attribute to the dogs' table and make this the primary key. We will call this new added field 'DogID'.

The second thing we need to do is to check that each record in one table is related to a record in the other table. We can do this by using a 'foreign key'; we copy the primary key from the one side (members) and put it in the many side (dogs). In other words, we copy the relevant ID from the member's table and put in the relevant record for each dog.

We now can view our improved database design:

ID	Init	Surname	Title	Postcode	Tel
1	A	Fish	Mrs	CV35QW	02476111111
2	C	Here	Mrs	CV27RF	01788222222
3	D	Lapidated	Mr	CV14RR	02476333333
4	X	Ray	Ms	CV12YY	02476444444
5	Y	Nott	Mr	CV24TT	01788555555

DogID	Dog Name	Gen	DofB	Breed	Origin	L/E Years	ID
1	Bongo	M	21/08/99	Poodle	China	5	1
2	Hiccup	F	08/08/98	Poodle	China	5	1
3	Rizla	F	09/09/00	Poodle	China	5	1
4	Gov	F	11/01/01	Alsatian	Germany	10	1
5	Manic	M	11/01/01	Poodle	China	5	3
6	Blip	F	02/02/01	Spaniel	France	7	3
7	Ruff	M	08/08/00	Poodle	China	5	5
8	Addi	M	10/02/00	Poodle	China	5	5
9	Catnip	F	10/03/99	Poodle	China	5	5
10	Emmi	F	11/03/01	Poodle	China	5	5
11	Gov	F	11/01/01	Alsatian	Germany	10	5

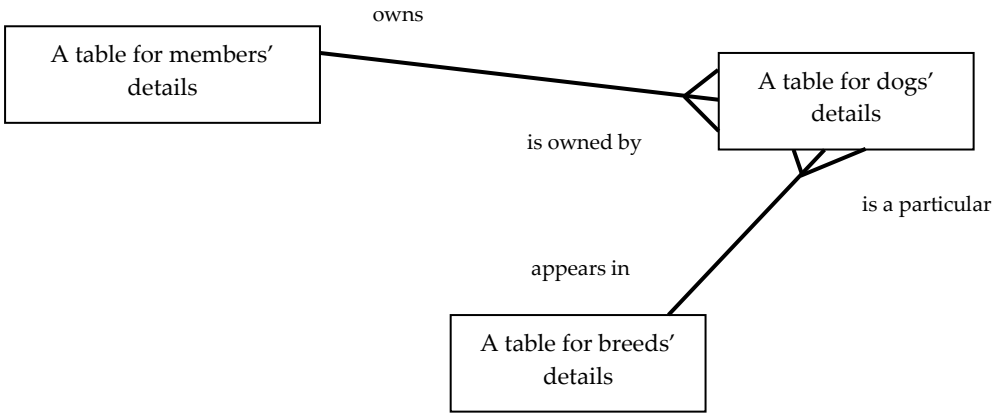
Don't worry if you are still a little confused! E-R diagrams can be confusing at first. There is a much more detailed explanation of E-R diagrams given in Section 2.

49.8 How is the relational database an improvement over the flat file design?

1) Each member's details are now only stored once.
2) Each dog is now identified by their own unique identity number.
3) A new member now has their details entered into only one record, not multiple records.
4) Changes to records need only be made in one place.
5) Details that are to be deleted only need to be deleted from one record.

49.9 Can we improve the design even further?

We have solved some problems by splitting the original flat file into two related tables. We haven't solved all the problems, however. We still have to record breed details over and over again and we can't record a new breed in our database unless there is a real dog of that breed actually owned by someone! We could try splitting up the dogs' table into two, like this:



Reading all of the relationships, we now have:

- Each member can own many dogs.
- Each dog can be owned by only one owner.
- Each dog can be only one particular breed. (This is a dog club for pedigree dogs only!)
- Each breed can appear in many different dog records.

As before, we can add some records to the tables to make 'seeing' what we have designed a little easier.

ID	Init	Surname	Title	Postcode	Tel
1	A	Fish	Mrs	CV35QW	02476111111
2	C	Here	Mrs	CV27RF	01788222222
3	D	Lapidated	Mr	CV14RR	02476333333
4	X	Ray	Ms	CV12YY	02476444444
5	Y	Nott	Mr	CV24TT	01788555555

DogID	Dog Name	Gen	DofB	ID	BreedID
1	Bongo	M	21/08/99	1	1
2	Hiccup	F	08/08/98	1	1
3	Rizla	F	09/09/00	1	1
4	Gov	F	11/01/01	1	2
5	Manic	M	11/01/01	3	1
6	Blip	F	02/02/01	3	3
7	Ruff	M	08/08/00	5	1
8	Addi	M	10/02/00	5	1
9	Catnip	F	10/03/99	5	1
10	Emmi	F	11/03/01	5	1
11	Gov	F	11/01/01	5	2

BreedID	Breed	Origin	L/E Years
1	Poodle	China	5
2	Alsatian	Germany	10
3	Spaniel	France	7

49.10 How is this relational database an even better improvement over the flat file design?

Not only can we now do the following:

1) Each member's details are now only stored once.
2) Each dog is now identified by their own unique identity number.
3) A new member now has their details entered into only one record, not multiple records.
4) Changes to records need only be made in one place.
5) Details that are to be deleted only need to be deleted from one record.

But also:

6) We only need to store the details of each breed once.
7) We can add a new breed to our database, even if a member has not acquired a dog of this breed yet.

49.11 Summary

- What we have seen is that flat files have their uses for small databases.
- As the number of records in a flat file increases, however, jobs that used to be simple and quick start taking a lot longer.
- There comes a time when it is much better to consider breaking up one big file to produce smaller tables that are related.
- These types of databases are known as 'relational databases'.
- A useful technique to help the designer achieve this is Entity-Relationship modelling.
- We talk about producing an E-R diagram for a particular database.
- E-R diagrams show a database designer what tables they need in a particular database and how those tables are related to each other.
- E-R diagrams do *not* in themselves show the designer what fields need to go in each table.
- (The designer will produce another document - a Data Dictionary - to record this information.)
- As with any new computerised system, an organisation needs to think carefully before it goes ahead and installs a computerised database.

Q1. What is meant by a 'flat file'?
Q2. When is it appropriate to use a flat file database rather than a relational one?
Q3. What is meant by 'data redundancy'?
Q4. What is meant by 'data consistency'?
Q5. What is the difference between a Primary Key and a Foreign Key?
Q6. A company has to think very carefully before changing from a paper-based database system to a computerised one. List some of the things they need to think about and plan for.
Q7. What does the E and the R stand for in E-R Diagram?
Q8. Define 'Entity'.
Q9. What exactly does an E-R diagram show you?
Q10. The manager of a shop keeps records of her customers, what they bought and who the sales person was for each purchase. She uses a flat file to store these records. Some of the records in the flat file are shown below. Suggest some of the problems that have occurred as a result of storing the records in this kind of structure.

Name	Address	Phone Number	CD_ID	CDName	Price	Staff_ID	StaffName
Mr Smith	21 Tree Rd	723144	34	Sam's Hits	£12.99	4	John Jones
Mr Jones	7 Lorry St	435554	34	Sam's Hits album	£12.99	6	Ali Patel
Miss Ng	43 Poppy St	344441	83	Cushion	£10.49	2	Peter Song
Miss Ng	43 Poppy St	344441	45	India	£15.99	2	Pete Song
Mr Smith	21 Tree Rd	723644	34	Sam's Hits	£12.99	4	John Jones

When you look at a problem like this, you need to consider the problems that have been highlighted in the NoKats example we looked at in detail earlier in this section. Ask yourself if any of the data in the records in the above table have been stored more than once. If you can find examples of this then you have found examples of data redundancy. Also ask yourself if data that is supposed to be the same, has been entered into the flat file identically in all places. If you can find examples where data is supposed to be the same, but has not been entered into the flat file identically then you will have found examples of adding anomalies. You will have found examples where the data has become inconsistent.

Chapter 50 - Normalisation

50.1 Introduction - Using normalisation to produce a relational database design

Normalisation is an analytical technique used in database design. It aims to create a database that has two key characteristics.

1) Redundant data is minimised.
2) The chance of making data in the database inconsistent is minimised.

To 'normalise' a database, you take a flat file version of it and apply three rules, one after the other. After each rule has been applied you may see that a table gets broken into two or even more tables, although the records in them will all still be related. This will help overcome all kinds of problems. Approached in the right way, normalisation is a very straightforward, very mechanical process. Approached in the wrong way and you will end up confused!

50.2 An example of normalisation

'DVD Now' is a DVD club. A typical member will have a membership card with their ID on, their name, address, phone number and join date on. When they want to take a DVD out, they present their membership card along with the empty DVD box from the shelf in the shop. The assistant gets out their Member's Record and the following details are written onto it: the ID of the DVD, the name of the DVD, the date it is due back on, the certification of the DVD and what the certification means. Up to 3 DVDs can be borrowed. A typical record looks like this:

Member's record
ID number: 241
Surname: Smith
Address1: 23 Jones Rd
Address2: Walforth
Phone: 123456
Joined: 12/01/99

DVD NOW!

DVD-ID	DVDName	DateDueBack	Cert	CertDescription
323	Roboteacher	11/08/04	X	18 and over only
6512	Harrytron	12/09/04	PG	Parental guidance
441	The Brainbox	30/09/04	PG	Parental guidance

A typical record.

You have been asked to analyse this database using normalisation. Where do you start? We will start with the setting-up step, which we will call STEP 0.

50.3 Set up the Analysis Table

Before we begin, UNF stands for 'Un-normalised form'. In other words, it is your database before you've normalised it. You may sometimes see 0NF instead. It means the same thing. There are a number of parts to this stage:

1) Get a piece of A4 paper and turn it around into landscape. Divide the paper into 5 columns and put the following 5 headings at the top of each of the 5 columns: UNF, 1NF, 2NF, 3NF and Name.
2) Then list all the attributes in the first column, UNF. You may want to shorten each attribute name to reduce how much you have to write!
3) Now for the first tricky bit! You need to identify any 'repeating groups' in the UNF column. If you look back at the example of a typical member's record, you can see that you would only enter the member's details **once**, but for each DVD, you would enter DVD-ID, DVDName, Due, Cert and CertDes. In other words, you may need to enter in these details **many** times - you will need to keep entering in these details for each DVD taken out. Groups of attributes that you have to keep entering in over and over again are known as a 'repeating group'. To show this, you should put those attributes inside brackets.
4) Finally, you need to identify a 'primary key' for both the repeating group and the group of attributes that are only entered in once. Imagine any table full of records (and remember that each record is in a row in the table). The primary key is the attribute that is *different*, and will always be different, for every single record (every single row). Very often, the primary key is an ID number or a reference number. In the repeating group, the one attribute that will always be different in every record is the DVD-ID attribute. To show this is the primary key for that group of attributes, you underline it. You also need to underline the MemID attribute because that one is the primary key for the group of attributes that are only entered into the database once.

If you have done everything correctly, your Analysis Table should look like this:

UNF	1NF	2NF	3NF	Name
<u>MemID</u> Surname Add1 Add2 Phone Join (<u>DVD-ID</u> DVDName Due Cert CertDes)				

The database is now ready for normalisation.

Now you have made yourself a table and listed all the attributes in the database and identified the primary keys in all the groups, you are ready to begin normalisation. The first step is put your database into 'first normal form', or 1NF.

50.4 Put the UNF data into first normal form, otherwise known as 1NF

A table is in 1NF if it contains no 'repeating groups'. So what do you do? You have already identified a 'repeating group' of attributes and the primary key for that repeating group.

1) Now, copy over the repeating group into the 1NF column. (You don't need the brackets any more but keep the primary key underlined.)
2) Once you have done this, copy across the attribute that acts as the primary key from the non-repeating group in the UNF column and add it to the repeating group in the 1NF column. Make sure that this attribute is also underlined. The repeating group in 1NF now has a 'compound' primary key, a primary key made up of more than one attribute (in this case, the primary key is made up of two attributes but they can be made up of three or even more attributes).
3) Finally, copy across the data items that remain in UNF to 1NF, into its own group.

It sounds a lot to do, but it is very mechanical. You do exactly the above and nothing different every time you want to convert a database from UNF to 1NF. If you have done everything correctly, your Analysis Table should now look like this:

UNF	1NF	2NF	3NF	Name
MemID Surname Add1 Add2 Phone Join (DVD-ID DVDName Due Cert CertDes)	MemID Surname Add1 Add2 Phone Join MemID DVD-ID DVDName Due Cert CertDes			

The database is now in 1NF.

Some points to note.

- You now have two groups of attributes, not one.
- Each group of attributes corresponds to a table of records, so we now have two tables of records.
- One table holds just the members' details.
- The second table holds all the other attributes.
- The records in each table can at any time be recombined into one record, however, because they are linked using the attribute MemID.
- The second table in the 1NF column has a ***compound*** primary key, not a ***simple*** primary key. This means that the key is made up of more than one attribute. If you want to retrieve a particular record from a table with a compound key, you must state all attribute values in the key; in this example you must state both the MemID and the DVD-ID to get any particular record back from that table.

You have now removed all repeating groups from UNF. That means by definition, your table is in 1NF. The next step is to put your database into the 'second normal form', or 2NF.

50.5 Take the database in 1NF and put it into second normal form (2NF)

A database is in 2NF if it is in 1NF and all the non-key attributes depend ENTIRELY upon the primary key! An alternative way of viewing this is to ask yourself which of the non-key attributes need *both* key attributes specified to get back a value for the non-key attribute. If a non-key attribute needs *both* key attributes, then that non-key attribute stays with the table with the compound primary key. If the non-key attribute does not need both key attributes, then it is moved into another table. Now this is a mouthful again, but the good news is that, like riding a bike, it is a lot easier to see and do than explain!

1) If any table in 1NF has a simple primary key (made up of only one attribute) then it is automatically in 2NF and can be copied across to the 2NF column.

UNF	1NF	2NF	3NF	Name
MemID Surname Add1 Add2 Phone Join (DVD-ID DVDName Due Cert CertDes)	MemID Surname Add1 Add2 Phone Join MemID DVD-ID DVDName Due Cert CertDes	MemID Surname Add1 Add2 Phone Join		

The first part of putting the database into 2NF.

2) The next step is to identify those non-key attributes (i.e. attributes that are NOT part of a compound primary key) which are related to only PART of the compound key.

   - In our example, the non-key attribute DVDName is dependent upon the DVD-ID attribute, which is part of the compound primary key. If I give you a DVD-ID, you should be able to get a DVDName from it. However, if I give you a Member's ID, that in itself will not let you get back the name of a DVD!
   - Again, if you are given a DVD's ID, you should be able to determine what certificate it is and indirectly what that certificate means. You can also think of it like this: what has a DVD's certificate and the certificate's description got to do with a member's ID? The answer is of course nothing at all. Both of those non-key attributes therefore need to be moved out of that grouping and into their own table because those two non-key attributes do not depend upon both parts of the compound primary key in that table.
   - The only attribute that requires some thought is 'Due'. A DVD is only due back if a member has it out. If you are given a DVD-ID, it might not be due back unless it has been taken out by a member! To be able to specify 'Due' you will need, therefore, to give the Member's ID and the DVD's ID.
   - What we are saying is that DVDName, Cert and CertDes are all dependent upon the actual DVD, but not dependent at all upon the member's ID. Due is dependent upon the Member's ID **and** the DVD's ID.
   - So now what do you do? You copy those non-key attributes that depend only on the DVD-ID attribute from the 1NF column to the 2NF column, putting them into their own table.

3) You then copy the part of the primary key that those non-key attributes depend upon from 1NF into the new table you just created in 2NF. In this example, you copy the DVD-ID attribute across. Now underline this attribute to show that it is the primary key of this new group.
4) You finally copy across any attributes left over from the old group from 1NF into 2NF, into their own table.

If you have done everything correctly, your Analysis Table should now look like this:

UNF	1NF	2NF	3NF	Name
MemID Surname Add1 Add2 Phone Join (DVD-ID DVDName Due Cert CertDes)	MemID Surname Add1 Add2 Phone Join MemID DVD-ID DVDName Due Cert CertDes	MemID Surname Add1 Add2 Phone Join MemID DVD-ID Due DVD-ID DVDName Cert CertDes		

The database is now in 2NF.

Some points to note.

- You now have three, related tables.
- Do not be concerned if one table is only made up of a compound primary key - that is quite common (although it does not apply in this example).
- At any time, the original record from UNF can be retrieved from the three tables in 2NF, because they are all linked!
- Your database is now in 2NF because by definition it was in 1NF and you have no tables where the non-key attributes only depend on part of the primary key. (You just removed them into their own table!).
- Going from 1NF to 2NF is usually the hardest part of normalisation.

Take your time and ensure you understand each of the steps in this part of the normalisation process. Once you have memorised the steps, they are the same for every design! The next step is to put the database into 3NF.

50.6 Take the database in 2NF and put it into third normal form (3NF)

A database is in 3NF if it is in 2NF and each table has no non-key attributes that depend upon other non-key attributes. Again, this is a mouthful but again it is easier to do than talk about!

1) Any table that is in 2NF and has *zero* or *one* non-key attributes, are automatically in 3NF. They can be moved across without further thought. This applies to the second group of attributes in the 2NF table.

UNF	1NF	2NF	3NF	Name
MemID Surname Add1 Add2 Phone Join (DVD-ID DVDName Due Cert CertDes)	MemID Surname Add1 Add2 Phone Join MemID DVD-ID DVDName Due Cert CertDes	MemID Surname Add1 Add2 Phone Join MemID DVD-ID Due DVD-ID DVDName Cert CertDes	MemID DVD-ID Due	

Tables with zero or one non-key attributes are moved automatically into 3NF.

2) Now, you do not even need to look at any attributes that are part of a key. Going from 2NF to 3NF concerns only non-key attributes. You are looking for any non-key attribute that is directly related and dependent upon another non-key attribute. The only example where this occurs is in the third table. The description of any certificate is clearly dependent upon the actual certificate's ID. If you are given a certificate's ID, then you should be able to get its description from it. Certificate description is dependent upon the certificate ID. In this situation, you must ***move*** CertDes but ***copy*** across Cert into their own table in 3NF. Notice the subtle difference between 'copy' and 'move'. Also note that Cert in the DVD table is now a foreign key (and it is a primary key in the CERTIFICATE table). It is used to link the table DVD with the CERTIFICATE table. Cert has a star put next to it in the DVD table to indicate that it is a foreign key in that table.
3) Now Identify which attribute is the primary key in this new table you have created. It is the attribute that the other attributes are depending upon. In this case, it is Cert. You should underline Cert to show it is the primary key.
4) Move across all the other tables and data, as they have no non-key dependencies.
5) Give each table a suitable name in the Name column.

If you have done everything correctly, your Analysis Table should now look like this:

UNF	1NF	2NF	3NF	Name
MemID Surname Add1 Add2 Phone Join	MemID Surname Add1 Add2 Phone Join	MemID Surname Add1 Add2 Phone Join	MemID Surname Add1 Add2 Phone Join	MEMBER
(DVD-ID DVDName Due Cert CertDes)	MemID DVD-ID DVDName Due Cert CertDes	MemID DVD-ID Due	MemID DVD-ID Due	MEMBER_DVD
		DVD-ID DVDName Cert CertDes	DVD-ID DVDName Cert*	DVD
			Cert CertDes	CERTIFICATE

The database is now in 3NF.

Your database is now in 3NF because by definition, it was in second normal form and now has no non-key dependencies.

Some points to note:

- Your database design now has four tables in it, all related.
- The database is designed so that redundant data is minimised. Also, the chances of introducing data inconsistencies have been reduced because you have removed the need to add, delete and amend data more than once.
- The table with the compound primary key has a name derived from the two tables that made up its key. However, you could more usefully have called this table LOANS!

Take your time and understand exactly what you have to do at each step - it will be the same for every database design!

50.7 An alternative notation

It is quite common in textbooks to see tables laid out not in a table form but using some 'notation'. For example, the Member table in the 'DVD Now' example at the beginning of this chapter had a primary key called MemID, and then the attributes Surname, Add1, Add2, Phone and Join. This would be represented using this notation as:

MEMBER (MemID, Surname, Add1, Add2, Phone, Join)

Also in the 'DVD Now' example, we had these tables:

MEMBER_DVD (MemID, DVD-ID, Due)
DVD (DVD-ID, DVDName, Cert*)
CERTIFICATE (Cert, CertDes)

Note the following things with this notation.

1) The table name is in capital letters.
2) The primary key is underlined. If a table has a compound primary key (one that is made up of more than one attribute) then all the attributes used to make up that key are underlined (as in the MEMBER_DVD table).
3) The attributes in each table are listed after the primary key.
4) If an attribute is a foreign key (used to link two tables together) then it has a star put next to it in the table where it is a foreign key. An example of this can be seen in the DVD table where Cert has a star next to it. In that table, it is a foreign key. In the CERTIFICATE table, however, Cert is a primary key.

Some students find this notation easy to work with after a little practice. Others find it easier to work with the Analysis Table because it presents a visual representation of the table design. Whichever method you prefer, it is straightforward converting from one method to the other.

Q1. What does normalising a database achieve?
Q2. What is meant by UNF or 0NF?
Q3. Write down exactly what you have to do to go from UNF to 1NF.
Q4. What is actually achieved when you put a database in 1NF?
Q5. Write down exactly what you have to do to go from 1NF to 2NF.
Q6. What is actually achieved when you put a database in 2NF?
Q7. Write down exactly what you have to do to go from 2NF to 3NF.
Q8. What is actually achieved when you put a database in 3NF?
Q9. What is the difference between a simple primary key and a compound primary key?
Q10. Showing the normalisation steps, normalise a student-course database. A typical record is shown below.

STUDENT RECORD

Student ID: 1232223
Surname: Patel
Initial: A
DofB: 09/06/1996

CourseNo	CourseName	LecturerID	LecturerSurname	LecturerInitial
435	Fractal Maths	23	Cooper	A
200	Mechanics	18	Smith	F
191	Chaos Theory	23	Cooper	A
31	Random Theory	11	Mistry	D