2. The Language Instinct (Programming Languages)

The communication of the dead is tongued with fire
beyond the language of the living.
T.S. Eliot, Four Quartets 'Little Gidding', (1940)

Ambiguity

Algorithms are just instructions used to describe actions. In this booklet all the algorithms are described in English. However this can lead to problems. English (as is any other Human Language) is a very difficult language to be precise in, and the whole point of an algorithm is that it tells you precisely what to do. One of the problems is that human "natural" languages are **ambiguous**. By this we mean that a given sentence can mean several different things. My favourite is one pointed out by horror writer Stephen King. He quoted a passage from a thriller he had read:

"His eyes slid down the front of her dress".

As he pointed out, if he had written that in one of his horror books, it would have been describing something totally different to the lecherous thing that the thriller writer was trying to suggest. There are many other examples where a sentence can mean several different things. Sometimes the differences are subtle. For example
"He picked up the red pen"
can mean something slightly different depending on the context and where the emphasis is placed.
"*He* picked up the red pen"
might be emphasising it was him who did it not her.
"He *picked up* the red pen"
might be trying to suggest that he did not leave it alone.
"He picked up the *red* pen"
might be pointing out that he picked up the red pen, leaving the blue and black ones.
"He picked up the red *pen*"
might be pointing out he picked up the pen leaving the pencil behind. The point of these examples is that it is easy to misunderstand precisely what is being said in English.

This fact has been known and exploited by lawyers for centuries: it is how many earn their living drafting laws and contracts to try to remove ambiguity. What do they do to minimise the problems? They resort to using very stylised language. They give definitions of the meaning words at the start. Essentially what they are doing is restricting the language to get rid of the ambiguity.

Despite all the trouble the first set of lawyers go to, to remove ambiguity, a whole other set of lawyers earn their money by arguing over the meaning of those same laws and contracts. Take insurance contracts for example. Frequently people discover they are not covered for something they thought they were. When I was a student I lived in a student residence and took out an insurance policy in case my possessions were stolen. When I took it out I asked the salesman if my records would be covered and was assured they would be. On reading the small print I noticed it seemed to suggest "collections" were not covered – and that there was an additional policy for DJs to cover such record collections. So was my record "collection" covered? That depends on what they meant by “collection”. I had about a hundred LPs at the time, but they were not a “collection” in the sense that there were no rarities amongst them. In the end the insurance broker decided they would not be covered, so I cancelled the policy and found another one that did not mention “collections”.

Whilst at University I also caught Glandular Fever. Just over a year later, whilst in Canada, I ended up in Hospital connected to drips, again with Glandular Fever. Every morning I had an accountant standing at the bottom of my bed asking how I was going to pay the thousand pound hospital bill. Luckily I had travel insurance but it contained a clause stating that I was “not covered for any illness I had suffered within a year of the claim”. Was I covered? That depends on when the year ran from – from when I was first diagnosed as having Glandular Fever or from when I recovered? The accountant was not convinced the insurance company would pay so wanted me to pay before I was treated.

Both the above problems arise from the ambiguity in the meaning of statements. When claims go to court lawyers from one side will be arguing that the contract means one thing. Lawyers on the other side will be arguing that it means the opposite.

Many jokes rely totally on there being different meanings to the same words or phrase (especially the really cringe-worthy ones children love to tell).

> *Why did Cinderella not get selected for the Soccer team?*
> *Because she ran away from the Ball.*
>
> *A man enters a restaurant and says “do you serve Oysters here?”*
> *“Why certainly sir, we always serve Oysters first” replied the waiter.*
> *“There you see”, said the man, pulling an Oyster out of his pocket and putting it on the chair, “You sit there and they will serve you first”.*

Why are they supposed to be funny? The first relies on two meanings of the word “ball” and the second on subtly different uses of the word “serve”. Telling a joke of this kind in a language designed for writing algorithms such as a programming language ought to be very difficult as such languages are designed so as not to have any ambiguity in meaning.

The rules of games can also suffer from being ambiguous. There are several games that when we play them we have to decide whose rules we are playing: my family read the rules as meaning one thing. My wife’s family grew up believing they meant something else. Before we can play we must remove the ambiguity, as otherwise it will end in argument. We must decide which rules we are going to play: the “Sheffield rules” or the “London rules”.

Recipes are similar to algorithms. My hope is that by following a recipe I will end up with something that looks like the picture in the recipe book. That rarely happens. This is in part due to my inability to follow instructions, but is also due to the vagueness of the instructions. “Stir until the sauce thickens”: how thick? Is this thickened or not? “Fry until golden brown”: was that golden brown or yellow? Oh it seems to be black now. “Add a tablespoon of sugar”. Would that be a level tablespoon or a heaped one, and if heaped, how heaped? Again the ambiguity of English is a problem in giving precise instructions that will be interpreted by everyone in exactly the same way.

Programming Languages
When giving computers instructions they need us to be precise. We cannot use a language for which there is any ambiguity. Computers need to be told exactly what to do at every point. That is why programming languages were devised. There are many different programming languages in existence, just like the fact that there are many different human languages. As with human languages some are similar, as they have evolved from the same original language. Many European languages have evolved from Latin – so having learnt Italian, Spanish is relatively easy to learn. The computer languages, Java and C++ similarly have much in common, so having learnt one the other is relatively simple to learn. Others like Russian have a completely different history, so are much harder to learn. Similarly, having learnt Java would only be of limited help in learning the Prolog programming language, as it is from a completely different family.

Languages can exist as several dialects. The English and Americans both speak "English" but there are differences between the two. For example, "I went out wearing only my pants" would mean something different to an American and to a Brit. In England "pants" are underwear, whereas in America the word just means "trousers". Similarly there are programming languages that though essentially being the same have minor differences: with some things meaning slightly different things, or with extra words. For example, there are several variations of the Java language.

To avoid the problems this causes many languages are standardised. A committee declares exactly what is allowed in the language and precisely what each construct means. Such languages can change over time, but only in ways permitted by the committee. The French try and do the same thing with the French language. The *Academie Francaise* rules on additions, with the aim of keeping out words like "le Hotdog" and ensure there is only one French language. As I write a law has been passed in Poland making it illegal to use non-Polish words and phrases such as "sex shop" and "supermarket" to protect the language (Connolly 2000). Because a language has been standardised does not mean it is set in stone. It just means it only changes gradually in ways that have been vetted as sensible and in a way that can ensure everyone changes at the same time.

Most (European at least) languages are based on alphabets. For example, English uses an alphabet of 26 letters (A-Z). All English words are made up of letters from that alphabet. There is no reason that a language has to use that particular alphabet though. Indeed many languages do not. Russian for example uses a totally different set of letters (33 in all): a different alphabet. Some of the letters like A are familiar to English speakers. Others like Я and Ю are completely different. Greek also uses a different alphabet with letters like α, β, γ, δ. Computer languages generally use similar alphabets to English, but allow numbers and other symbols like '*' to be used as extra letters.

There are several different kinds of computer languages. We will consider two: **machine code** and **high-level languages**. Computers need instructions to be converted into machine-code languages if they are to be followed, as machine code is all computers really understand. These are very simple languages that use the alphabet that computer hardware works with. They have only two letters: 0 and 1. All words in machine code languages are therefore made from sequences of 0s and 1s. For example, 00000100 might be one word, whereas 10101010 might be another meaning something different. Of course not all sequences of 0s and 1s might be a word that means anything. This is just like the fact that *quelle* is not a word in English – though it is a word in French. Similarly the same word might mean different things in different languages. In human languages these are called *false friends.* A famous example of someone making this kind of mistake is when the then US President John F. Kennedy announced in a speech in West Berlin in 1963 "*Ich bin ein Berliner*" intending to say "*I am a Berliner*". Unfortunately in German *ein Berliner* is a doughnut not a person from Berlin, so he had actually said "*I am a doughnut*". The applause that resulted was not for the reason he thought! Different computer languages also often use the same words with very slightly different meanings. For example the word "for" in the language Pascal means something slightly different in some situations to the same word in the language Java.

Compilers

Humans find machine-code languages very hard to read and write. High-level languages aim to overcome this problem. They use more familiar alphabets and have words similar to those in English to help humans understand them. This however leaves us with a problem. If a computer can only follow instructions if they are written in a machine-code language of 0s and 1s, what use is a high level language to it. This is where compilers come in.

A programming language **compiler** is just a form of translator. Just like a human translator it takes a document written in one language and writes out a copy of it in another language. Different compilers can translate between different languages just as human translators often can only translate between two languages. A difference is that compilers are not capable of translating documents back again. It is as though someone could translate from French to English, but if given an English document would not be able to translate it to French. I am a little like that. I know enough French that I can get the general idea of what a French book is saying, but I do not remember French well enough to write French. Compilers take that to an extreme. They can translate perfectly from one language to the other, but not at all in the other direction. In particular, a compiler translates from a high-level language to the machine-code language of the computer that is to follow the instructions. The compiler reads the high level program and writes out a new version of it in 0's and 1's as a separate document. The computer then reads and follows the instructions in this document rather than the high level version originally written. Executing a newly written program is thus done in two stages. First it is compiled (ie translated). This need only be done once as you can then just keep using the translated version rather than needing to retranslate it every time you wish to use it. **Compile-time** refers to the point when this translation and error checking is done. Once compiled, the translated instructions can be followed: that point is referred to as **run-time**.

In the Discworld books by Terry Pratchet, the non-magical technology works by Imp power (Pratchett, 1983). For example, cameras are a box with an Imp with a paint box and easel. The Imp paints whatever it sees out of the camera when told to – the result is a Discworld photograph. Other technology works in a similar way. The only computer works in a different way, however – using ants and possibly due to being developed by Wizards has the whiff of magic about it. A Discworld computer that works using Imps would be possible and no doubt will eventually be invented on the Discworld. All it needs is a large number of imps each capable of doing specific simple tasks – following instructions given. Of course, humans could be used instead of Imps but then the box containing it would be too big to fit on a desk top. In fact a whole building would be needed. In essence this was the first meaning of the word computer as we dicussed in the introduction – people doing computation.

How would an Imp computer work in practice? A program written in some appropriate programming language would first be compiled. For an Imp computer, this would involve checking that the instructions made sense, translating them into Imp language and giving one instruction to each Imp. A special set of bilingual Imps could perhaps do this. Some mechanism would also be needed to ensure that the Imps knew when it was their turn to follow their instruction. Otherwise, the actions of the algorithm might be performed in the wrong order. Storage space (such as boxes) would be needed for them to store results of calculations. We will return to the design of this Imp computer in subsequent chapters as we discuss these issues in detail.

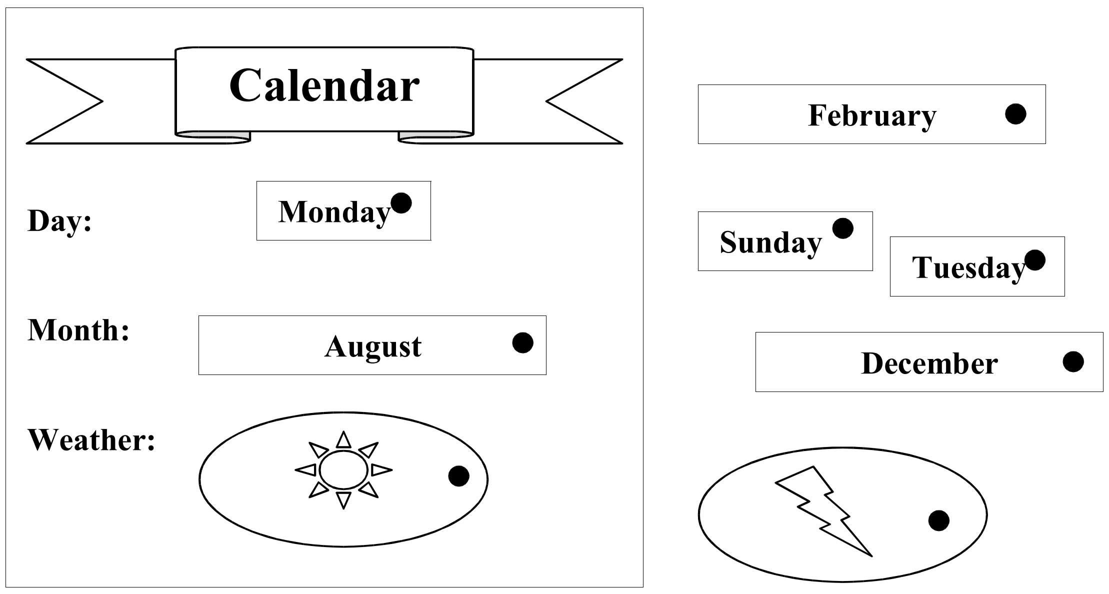
Compilers take a whole program and translate it completely before executing it. Suppose I had a set of directions written in Arabic which told me how to get to the market. As I cannot (yet) read Arabic I cannot follow the instructions unless I have them translated. With a compiler-approach I would give the instructions to someone who could read Arabic. They would give me back a copy of the instructions written in English. Armed with this new document I would be able to follow the instructions and get to the market. Some programming languages use **interpreters** rather than compilers. An interpreter effectively translates the program as it is executed: instructions are translated and immediately followed before moving to the next instruction. It is as though I took the person who could read Arabic with me to the market. They would translate the instructions one at a time. At each junction we came to they would translate the instruction and tell me which way to turn. Unlike with a compiler, with an interpreter I never receive a copy of the whole set of instructions in English. If on the following day I wanted to go to the market again using the same instructions I would need to take the translator with me again who would go through the same process all over again. With the compilation approach, I only do the translation once. I then have a copy of the instructions in a language I understand that I can use over and over again. Such a translated set of instructions is in computing terms **object code**. Similarly, once a program has been compiled, the object code can repeatedly be executed without any further need for the original program to be compiled again.

Syntax and Compile-time Errors

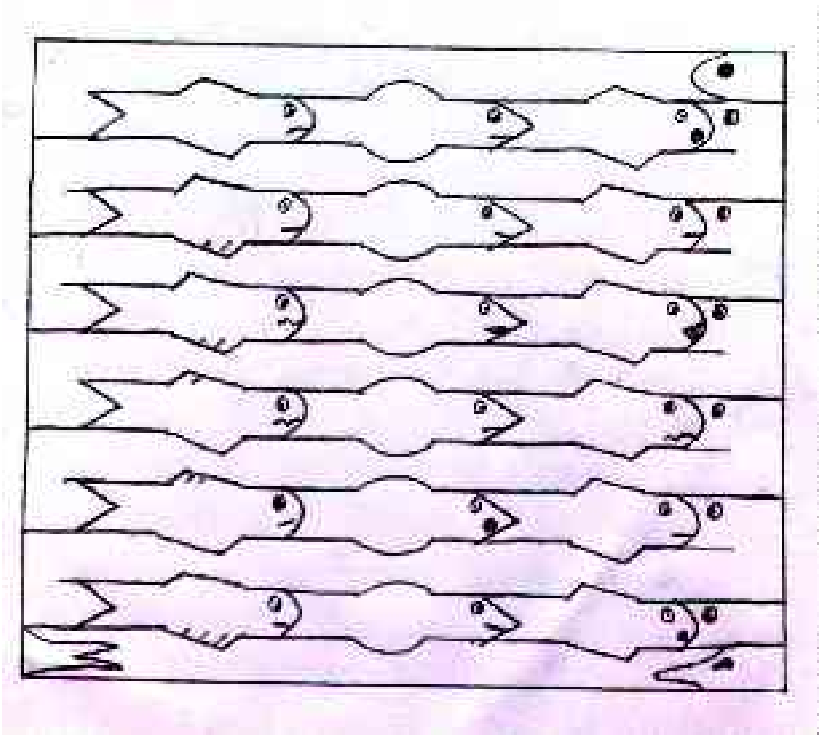
A computer language is defined by two things: its **syntax** and its **semantics**. By **syntax** we mean which combinations of letters are allowed as words (i.e. how to spell in the language) and what order those words can be put in to make meaningful sentences (i.e. how to write grammatically correct). By **semantics**, we mean what those words and sentences actually mean. As computer languages are languages for giving instructions in, the semantics of a computer language tells you what each word tells you to do. As we discussed above, for computers, it is important that it is always possible to tell exactly what each thing means with no ambiguity.

This leads to a second thing a compiler does. It looks out for mistakes. If I gave a human translator the “sentence”: “The cat sat on the matrs” to translate into French they would tell me that “matrs” is not a word in English so they cannot translate it. I made a mistake in the English that I need to correct before the sentence is translated. Similarly they would probably point out if I asked them to translate “The cats sat on there mats” that using “there” was grammatically incorrect in English. They are pointing out spelling and grammar errors. These are **syntax errors**: errors that mean what I have written is not English so cannot be translated until it has been corrected in the English version. Compilers will also point out syntax errors: spelling and grammar errors in programming languages and refuse to translate the program until it has been corrected. Students often complain that the compiler ought to correct such errors and carry on the translation rather than just giving up. Similarly one might expect the human translator above to be able to work out what I meant and do the translation. However, that could be dangerous. What if they mistakenly assumed I head meant the first sentence to be “The cat sat on the mat” and translated that. In fact I meant to write “The cat sat on the mattress”. Compilers cannot read the minds of the person who wrote the program. It is therefore safer for them to just point out the error rather than trying to correct it. Syntax errors are **compile-time errors**. They are found during the translation process.

The way the compiler treats a program is a little like a special kind of jigsaw puzzle. The program is split into what are known as tokens – the separate pieces of the puzzle. Some pieces have identical shapes and can be interchanged. Some simple toddler jigsaws are like that – for example each piece is the carriage of a train containing a different letter of the alphabet. They fit together in any order, except the engine has to be on the front as it only fits to a piece behind. A more complicated example is a “Jigsaw Calendar”. Imagine one of the wooden jigsaws for young babies – the sort that have pieces with handles that fit into slots on a wooden board. A Jigsaw calendar is similar except that several different pieces fit into each slot. One day holds the day of the week, another the month and another perhaps for the weather – where pictures of the sun might fit. However the pieces are such that only day pieces fit into the day slot, only month pieces will fit in the month slot and only weather pieces in the weather slot. The pieces are the tokens. Many different versions of the calender can be created using the different pieces, but they all are syntactically correct. The shapes physically enforce a syntax. Checking for syntax errors involves checking that the pieces really do fit.



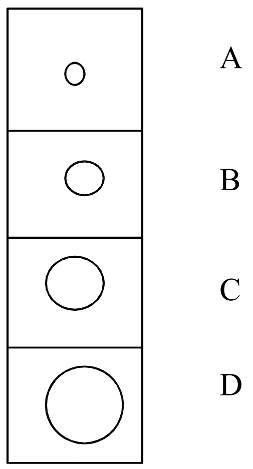
Here is another more complex jigsaw puzzle made of fishy shapes. Each of the pieces is a fish or an eel. Most of the fish pieces are the same shape. The middle fish each have round fins and are identical to each other in shape. The pointed-fin fish are also identical to each other in shape. Each fish has a slightly different expression. There are thus many ways to create a jigsaw that fits. However, only one combination is the one below.



Such jigsaws are such that as long as the pieces fit, you get a sensible picture (though not necessarily the one on the box, or one that corresponds to today's date in the case of the calendar). Syntax errors are like the situation when someone tries to force a piece into a position it does not fit. Even if you have used all the pieces or filled all the gaps, if any of the pieces do not properly fit then you have solved the jigsaw puzzle. If you have written a program and the tokens have been put in places they do not fit, then you do not have a real program. The situation when all the pieces do fit, but you have a different picture to that on the box is like a run time error. You cannot detect it by looking at the way the pieces and whether they fit. It is after all a perfectly good picture. It is only wrong with respect to the picture you were trying to create. Unless you know what that is you cannot say the jigsaw is wrong. Similarly with a program, if the tokens do fit, then it is a program that does something. However, the compiler cannot tell whether it does what it is supposed to do because it does not know what that is.

What are the tokens for a language (human or computer)? They are the different kinds of words. In human languages like English, the tokens are things like nouns (dog, rock, hat, etc) and verbs (eat, run, etc). In English nouns and verbs fit into different positions in a sentence – they have different "shapes". Get the words in places where they fit and you get a sentence, though one that might be nonsense. In computer languages, the tokens are not nouns and verbs but "identifiers", "keywords" etc. We will look at the different kinds of tokens in later chapters. For now we will use a puzzle to give you the idea.

Consider the following watery puzzle (variations of which we will return to in a later chapter). You have four counters representing bubbles of different sizes placed on a board in the order shown with the smallest bubble at the top and the largest at the bottom.



The aim of the puzzle is to get the counters so that the one with the biggest bubble is at the top, that with the next biggest bubble comes next and so on. This must be done in as few steps as possible. The bigger the bubble, the nearer to the top (the surface) that it bubbles. However this can only be done be swapping pairs of adjacent bubbles at a time (which counts as a turn). For example, on the first turn you might swap, the counters on squares A and B. The second turn might be to swap C and D and so on. The hard (well not that hard!) part of the puzzle is that you only have 6 turns.

Now lets add a twist to the puzzle. Not only must you solve the puzzle, but you must write down the set of instructions that will be followed – you must write down an algorithm that always solves the puzzle in 6 turns. Moves are written in the form

____ is swapped with ____ .

In the blanks you must write letters A, B, C or D. For example, the first two moves of the algorithm might be:

A is swapped with B.
B is swapped with C.

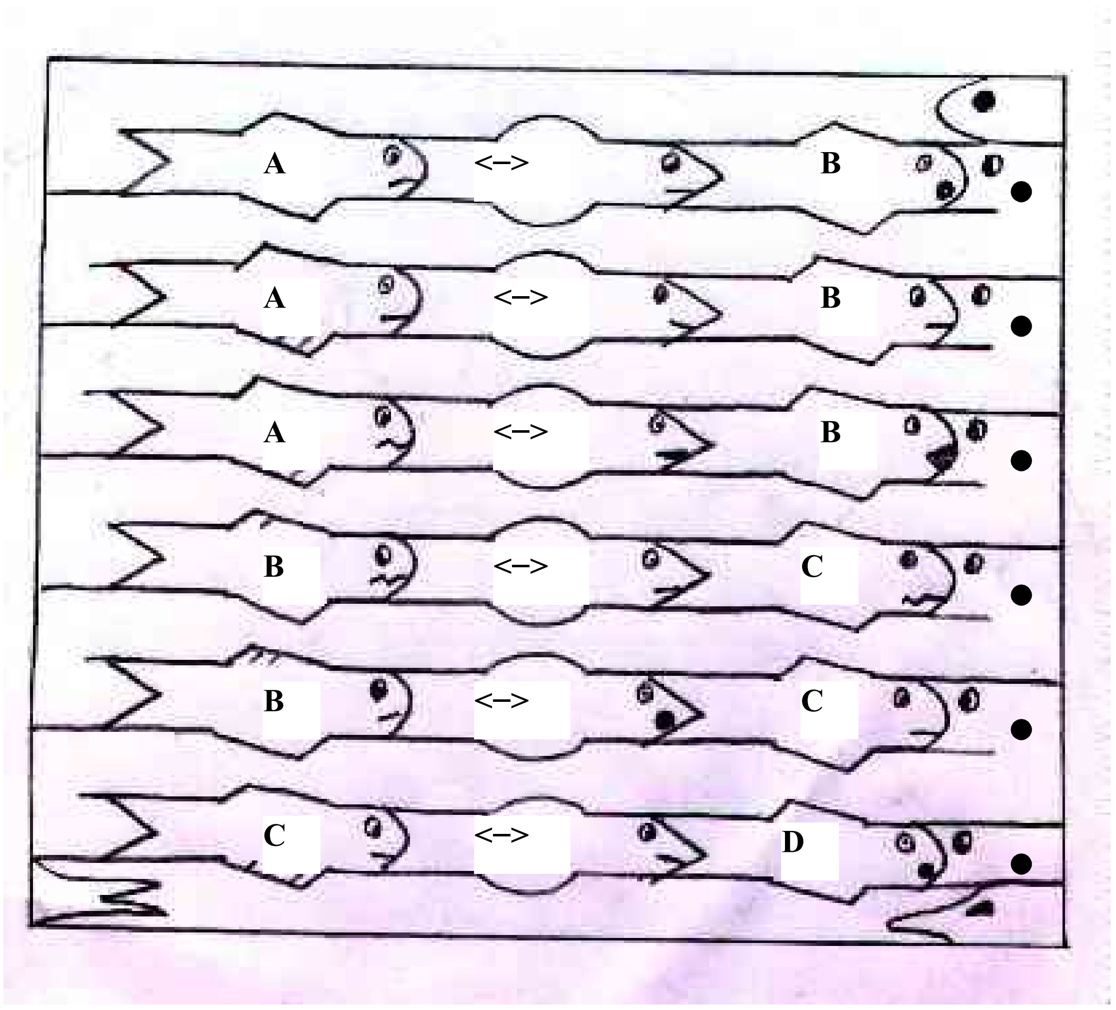
By "is swapped with" in an instruction we mean exchange the positions of the bubbles on the two counters on those squares – but only if they are in the wrong order – otherwise leave them alone.

What we have created is a language for writing algorithms. A program in this language consists of 6 "swap" statements. Our language has a syntax – we have said what each statement must look like – what words (ie tokens) can be used. The possible tokens are "is swapped with", "." and the letters labelling the squares on the board: "A", "B", "C" and "D". We have also specified the positions they can appear in: similar to the way the grammar of English specifies where verbs and nouns can appear. Notice that our language is not English though it looks a little like it. We are much more restrictive in what we allow to be written. In programming languages, symbols are sometimes used to mean operations. For example, in the above example the language might use <–> as an abbreviation for "is swapped with". The two instructions given above would be written is follows in this new language.

A <–> B.
B <–> C.

Now the tokens are the letters A, B, C and D and the symbols "." and "<–>".

The following may help you understand what we meant earlier that program tokens are like pieces in a jigsaw. Below are the pieces of a jigsaw similar to the one we looked at above. The shapes are the same but we have associated a different token with each shape. The board positions appear on identical shapes. The puzzle now is to construct a jigsaw that gives instructions that solve the original puzzle. You cannot change the pieces or what is on them, just move them around. They must of course only be put in positions that they fit. Any finished version of the jigsaw will be free of syntax errors in our language. It will therefore give a sensible sequence of instructions that can be followed in that it will tell you to swap the counters on two squares of the board. However, it may still contain errors that are not syntax errors: following the instructions could still do the wrong thing leaving us with the bubbles of the original puzzle in the wrong places. The version of the jigsaw below is like this – it is syntactically correct – the pieces fit, but following the instructions as in the jigsaw would not always solve the bubble puzzle. Try and rearrange the jigsaw so that the instructions it gives do always solve the puzzle.



Notice that the shapes of our pieces do not completely guarantee that the instructions follow the rules of the puzzle. In particular, it does not guarantee that the squares to be swapped are adjacent. We would need more complicated pieces (i.e. more complicated syntax rules) to prevent this. As it stands such errors cannot be detected just by checking that the pieces fit. With changes to the language they could be made into syntax errors. This kind of situation occurs in programming languages too – problems that can only be detected in one language when the program is run can be detected at compile-time in another language. We look at these other kinds of errors in the next section.

Semantic and Run-time Errors

Syntax errors are thus situations where what is written is incorrect use of the language concerned. What is written is just not English, or it is just not Java, or whatever. There are other kinds of errors that can be made when writing in a language: what is written is perfectly good English, for example. It just does not mean what the person who wrote it meant. Kennedy could have been trying to say "I am a doughnut", when he wrote "Ich bin ein Berliner" so someone checking his speech for spelling and grammar errors would not point out the problem. This is a **semantic error**. The error is in the meaning of the sentence and not in its spelling or grammar. The sentence "I was born on the 30th February" similarly contains a semantic error. The spelling and grammar is fine. It is perfectly good English. However when you consider its meaning, it makes no sense as there is no 30th February. It is a meaningless sentence.

For algorithms and computer programs, semantic errors are errors where the instructions are perfectly good instructions but where following them results in the wrong or unintended thing happening. Suppose that on being asked the directions to Kirkcroft Lane as in the previous chapter I gave the following instructions:

> "Go down the hill.
> **Turn left at the Co-op.**
> Go up the hill.
> Take the first turning on the left. (It is just after the Navigation pub).
> You are there."

These instructions would seem perfectly sensible to the driver as they were writing them down. Suppose, however, that there was no Co-op (I had forgotten – it was knocked down last year and replaced by a car park.) The driver would keep driving, until eventually they realised they were lost again. At that point they would give up trying to follow my instructions. In the worst case the instructions might lead them, if followed to drive over a cliff! The algorithm has "**crashed**". A **catastrophic run-time error** has occurred. My instructions were meaningless. They contain a semantic error. The driver could not tell this until they started to follow the instructions – so the error is a run-time error.

Semantic errors do not need to be catastrophic. They can just lead to the wrong thing happening. For example, suppose I gave the following instructions to the driver, getting my left and right mixed up on the first turning:

> "Go down the hill.
> **Take the first turning on the right.**
> Take the first turning on the left.
> You are there."

Assume the road did have a right turn at some point, from which there was a left turn. Not only would these instructions appear to be sensible, when the driver was writing them down, the driver could follow them without noticing anything wrong. They would turn into the final road, believing themselves to be at their destination. However, because my instructions contained a semantic error, they would be in the wrong place. Unlike the last example, the algorithm did not crash as this time a result was obtained – just the wrong result.

To take a different example, suppose the rulebook for a football tournament included the rule:

> "In the event of a draw at the end of extra time, the team that scored last will go through to the next round".

That is a perfectly good rule in the sense that it could be followed, but it almost certainly contains a semantic error. The person writing it probably meant to write:

> "In the event of a draw at the end of extra time, the team that scored first will go through to the next round".

As the compiler cannot spot such errors, they can remain undetected even when the instructions are followed. Notice also that such an error does not necessarily mean the wrong thing is *always* done. If no matches ended up in a draw after extra time, the above rule would never be followed, so the correct team would always go through to the next round. The error would just sit there in the rulebook in an instruction that was never read.

When writing computer programs that kind of error is very easy to make. The Y2K bug is perhaps the most famous. The instructions of many computer programs used 2 digits for the year: 63 meaning 1963 for example. For decades such programs worked perfectly well. It was only when we reached the year 2000 that the problems happened as some computers treated 00 as meaning the year 1900 instead of the year 2000 as intended. One of the Mars probes crashed on Mars due to a semantic error in its instructions about how to land. The problem there was that some of the programmers wrote instructions that calculated in metric units, whereas others assumed the same numbers were in imperial units. Semantic errors are **run-time errors**. They cannot be found during the translation. Instead they are found by following the instructions and determining what happens.

Underlying a language is the concepts it is being used to express. In English if we wish to express that a particular person (say Emma) is doing a specific thing (say eating), there are two basic concepts that the language must allow us to express. It must allow us to identify a specific person (Emma) and also identify what she is doing to it (eating). Consider an imaginary caveman language that did not have a concept of people though did have concepts of actions. The statement that Emma is eating cannot be expressed in such a language – the nearest the caveman could say would be “eat”. Who was eating would need to be expressed in some other way than using the language (by pointing at Emma) perhaps. A third concept you might need would also be of time. A film I saw as a teenager called Caveman only had dialogue in such an invented language. As you went into the cinema you were given a leaflet listing all the words used – a dozen or so. From what I remember it had words for food, sex, going to the toilet, you me – just the most important to a caveman. It did not have tenses however. You could therefore say “I eat” but that could mean both I am eating and I have eaten. Not all things can be expressed in all languages. If we want a language for expressing algorithms, what we need to do is work out what basic concepts we need to be able to express and then provide ways of expressing them in the language. The basic thing as we have said that we are trying to express in an algorithm is a series of actions and the order they will occur in. We must work out what kinds of actions we wish to perform and work out the components. For example, we wish to be able to move data from one place to another. We must consider what information we must be able to express if we are to specify something is being moved – as we will discuss later in this case we must be able to indicate what is being moved and where it is to be moved to. In the subsequent chapters we will look at a series of different kinds of concept we need to express in algorithms, breaking them down into the basic parts. Any programming language that uses that concept must have a way of expressing those basic parts. Incidentally, I hasten to add that Caveman was not a particularly good film, before you rush out to order the video.

Completeness
A set of instructions or algorithm is said to be **complete** if it covers all eventualities. It must tell you what to do whatever the situation. This is *not* quite the same as meaning “finished” in the sense of you having finished writing the instructions. For example, suppose I am asked by my wife to go to the bakers to buy her Chocolate Muffins as we have nothing for Breakfast. If that is all the instruction I have been given then I could have problems, as it does not cover all eventualities. The instructions:

1. Go to shop
2. Buy Chocolate muffins

3. Return home

do not tell me what to do if the shop does not have that cereal. The instructions are not complete. They do not tell me what to do in all circumstances. If the shop does not have chocolate muffins (which happens all too often). I can no longer do the task by following instructions. I must instead use my imagination and guess what the appropriate thing to do is (and get into trouble if I return home with the wrong thing). How do we make the instructions complete? We provide instructions for each eventuality that might arise.

```
1. Go to shop
2. If the shop has Chocolate Muffins
           then buy Chocolate Muffins
   else if the shop has Croissants
           then buy Croissants
   else buy nothing.
3. Return back home
```

The important thing for completeness is to have the final "catch-all" instruction: "else buy nothing". This tells me what to do if none of the situations specified actually happen: since we cannot easily list all possibilities. It means that whatever the shop stocks or does not stock, I will have an instruction telling me what to do: buy nothing.

Are the new instructions complete? What if the shop I go to is shut? Should I just return or go to another shop? The instructions again do not tell me. Providing instructions for all situations is near impossible in many real life situations. The way round this is to have a last default case that says for example "Give up" if none of the other rules apply.

An area where having complete rules is often important is in sports competitions. As I am writing this, England have just been knocked out of the Euro 2000 football tournament after losing to Romania due to a last minute penalty. Before the match most of the papers included rules for working out who would go through with Portugal whatever the result of the matches between Romania and England and Germany and Portugal. For example, the Guardian (Guardian 2000) explained it as follows:

> *England will qualify for the quarter-finals with Portugal if they win or draw with Romania.*
>
> *Defeat would put England out...and would mean Romania progressing if Germany fail to win or if Romania win by a greater margin.*

Implicit in this is the extra rule:

> *If England lose and Germany win by a greater margin than Romania then Germany will go through.*

Are these rules complete? In other words is there any combination of results from the two matches not covered. If this is so we could be in a position of not being able to say who goes through. This is possible so the rules are not complete. What if Germany and Romania win by exactly the same margin? Who goes through then? We need an extra rule – and the Guardian gave one:

If Germany and Romania win by an identical score, Romania will progress thanks to a better points coefficient from the qualifying competitions for France98 and Euro2000.

The rules are now complete. Whatever the scores of the two matches, we can say who goes through. As it happened, Romania won and Germany lost, so from the second rule Romania went through and England, sadly for Kevin Keegan were knocked out.

These rules are based on a more general set of rules about who goes through from each group. To be complete, those more general rules would also need to say what would happen in a similar situation if the two teams also have identical point coefficients from the qualifying competitions. This was only not an issue in the above because it was known that the Romania had done better in qualification so that could be built into the rules.

Humans have intelligence that they use to fill in the gaps in incomplete instructions. Computers on the other hand are not intelligent so complete instructions are very important. Programming languages are generally designed so that all instructions written in them are guaranteed to be complete.

Determinism

Another property that is often important of rules is that they are **deterministic**. By this we mean that if the conditions in which they are followed are identical, then exactly the same result will be obtained by following the result. You can determine what the result will be in advance if you know the rules and the conditions.

One place determinism crops up in real life is in games. Games are often split into two categories: games of chance and games of pure skill. An example of the latter is chess. If in two games the players make the same moves, then the outcome of the game will be identical and so could be predicted. In fact most chess players write down the moves of the games they play so that they can play the game back again later, and work out where they went wrong, or how they could have played the game better. In deterministic games, if you are clever enough you can theoretically work out in advance the best move to make from any position. For example, I (and others throughout history) have analysed Noughts and Crosses and know what moves to make in any situation. The book *Winning Ways* does this for a large number of games. Try and work it out for Noughts and Crosses for yourself! Determinism does not mean all chess or Noughts and Crosses games are identical, just that if the other player does the same things in two games, the rules ensure the same outcome if you also make the same moves. The most interesting games of skill are those like chess that have so many options at any move that their determinism is hard to utilise. The interest of the game is being able to make best use of the determinism.

An example of a game of chance is Roulette. It is non-deterministic. The outcome cannot be predicted in advance as it depends on the spin of the wheel. The whole point of a roulette wheel is to introduce non-determinism into the game. I can place my chips in exactly the same places on the table in two consecutive games. In one I could become a Millionaire and the other despite all the players doing exactly the same thing as last time I could lose everything. Think of some other games of chance and ask yourself, what it is that is providing the non-determinism. Non-determinism is in part about things outside the control of the instructions or players. If it was possible for some race of Aliens to predict the roulette wheel perfectly then the game would be deterministic for them. Crooked gambling houses may of course have crooked wheels that they can control. If they can control exactly where the ball lands (with magnets perhaps) then the game is no longer non-deterministic. If there is any element of chance in a game, however much skill is otherwise involved, it is still non-deterministic as there are still situations where different outcomes could result.

Consider my wife's instructions for shopping given earlier.

```
1. Go to shop
2. If the shop has Chocolate Muffins
           then buy Chocolate Muffins
   else if the shop has Croissants
           then buy Croissants
   else buy nothing.
3. Return back home
```

If on two different weeks, I am sent to the bakers for breakfast, then if on both occasions the shop stocks the same things, I will be guaranteed to return with the same thing (or with nothing both times). The set of rules given earlier are deterministic. If the shop stocked different things on the two days I could of course return with something different – that is why we specify that "the conditions are identical". Here we mean the things stocked by the shop are identical.

```
1. Go to shop
2. If the shop has Chocolate Muffins
           then buy Chocolate Muffins
   else if the shop has Croissants
           then buy Croissants
   else buy anything.
3. Return back home
```

This set of rules is not deterministic: they are **non-deterministic**. What is the difference? The catch-all rule gives scope for different things to happen, even if the shop stocks exactly the same things on the two occasions. On the first occasion, I could return with apple doughnuts by following the rule, and on the second, following the same rule I could return with cherry flapjack. Even if she knows what the shop stocks on a particular day, she would not be able to predict with certainty what I would have bought.

Here is another set of rules that are **non-deterministic**

```
1. Go to shop
2. If the shop has both Chocolate Muffins and Croissants
           then buy Chocolate Muffins OR buy Croissants
   else buy nothing.
3. Return back home
```

Again even if my wife knew that the shop had muffins and croissants, she would not be able to predict which of the two I would pick on any given occasion. Note that determinism is different to completeness. The above rule does tell you what to do in each situation, but what it says to do is pick randomly.

Termination
A final property that it is often important for algorithms to possess is that they are **finite**. By this we mean that when we follow the instructions, we will always finish and produce a result. We will not follow instructions forever, never coming up with an answer. For example, when we type in a calculation in a calculator, we want it to give us an answer. We do not want there to be a situation where it calculates forever, never giving us the answer. We want the calculation to terminate. Non-termination of a set of rules is normally a **semantic error**. It is normally a result of a mistake in the way the rules were written. We will discuss termination in more detail later.