# 3 Computational aspects

In this chapter we present some background concerning the essential computational aspects of MT system design. We will assume in the reader a basic but not in-depth knowledge and awareness of computers. For this reason, readers who come from a computer science background may find that some of the material in this chapter, especially in the first five sections, rather familiar to them, and such readers might like to skip those parts which do not address new issues.

The chapter is in two parts: the first (sections 3.1 to 3.4) concerns some rudiments of computer science, while the second (section 3.5 onwards) turns to the aspects of **computational linguistics** which are particularly relevant to MT. Computational linguistics in general is concerned with developing computational tools for linguistic problem solving, and most of the techniques described in the second part of this chapter also feature in many applications other than MT.

## 3.1 Data and programs

In essence the computer is simply an electronic machine which can process large amounts of data quickly and accurately (in the sense that it will always produce identical results if asked to repeat the same task more than once). The physical unite which comprise a computer system, its **hardware**, include a central processor (which performs the basic operations), an internal storage space, or **memory**, and various **input** and **output** devices which allow the computer to communicate with the outside world. Among these are keyboards, printers and screens (visual display units or VDUs), typically used directly by humans for exchanging information with

the computer; but other 'input/output' devices include disk drives, magnetic tape readers and so on, which allow **data** to be stored externally, and to be read in and used as necessary. One particularly important type of data is the **software**, which tells the computer what to do. Software generally consists of **programs** (the American spelling is always used for this term in computer science, even in British English) which are sets of instructions telling the computer to read some data, perform some calculation, output some message, and so on.

Programs are written in **programming languages**, of which there are many different varieties. Programming languages enable the human programmer to write instructions clearly and concisely without worrying too much about the details of how the computer will actually go about implementing the instructions. They also enable programs written for one computer to be used on another computer, as long as the programming language in question is available on both the computers. The variety of tasks that a computer can be programmed to perform is of course vast, and different programming languages are typically aimed at making certain types of tasks more or less easy. Very general-purpose programming languages which slightly facilitate a broad selection of tasks are called **low-level** programming languages, while programming languages aimed at greatly facilitating much more specific tasks are called **high-level** languages, and accordingly have a much narrower range of applications. As we will see later, some MT systems are written in low-level languages, while some are written in high-level languages. The former have the advantage of being portable from one machine to the other, but they have the disadvantage of being more difficult for the human observer to understand, because so much of the program code has to do with mundane, non-linguistic aspects of the translation task.

Programs are typically stored somewhere in the computer memory or, more often, on some external device like a diskette, and then loaded when they are to be used. In this sense, program code is just an example of data. Indeed, there are special programs — usually written in very low-level languages — whose job it is to convert programs written in some higher-level language into a form that the computer can understand. Sometimes the process might be repeated so that what the higher-level program actually does is to interpret yet another program written in an even higher-level programming language. This happens quite often in computational linguistics.

## 3.2 Separation of algorithms and data

There is in general a very important distinction between programs and other sorts of data. A program is essentially a set of instructions (an algorithm) saying how to solve some problem or achieve some goal. The best programs express these solutions in the most general terms, so that they can be used for solving a whole set of similar problems in the same way. Imagine for example a program which was able to add 3 and 5 and give the answer 8. This program has some use, but it is rather limited. In particular, it does not help us add 7 and 3, for example. Now consider a program which will add ANY two numbers: this is a much more useful program. The only problem is that we have to tell it, each time we run it, WHICH

numbers to add. The program itself says HOW to add, but it is the data which must be supplied separately that tell it WHAT to add. This idea of **separating** the data from the programs is very important in computational linguistics and MT. As a simple example, consider a program which looks up words in a dictionary. This program will be much more useful if we can use it for different dictionaries (as long as they have the same format).

This notion of separation (sometimes called 'decoupling') of algorithms and data has many consequences in computational linguistics. It means for example that MT systems can be designed to a certain extent independently of the particular languages concerned; the algorithms wilt determine how the computer goes about translating, while the data will capture the linguistic details. This means that, in the ideal world at least, computer scientists can work on the algorithms, white linguists work on the data. In reality it is not so simple, but there are certainly examples of MT systems consisting of **translation software** which operates in a similar manner for different sets of data, e.g. dictionaries and grammar rules.

A further advantage of separation is that it helps to distinguish two essentially different types of error that may occur in large computational systems, namely errors arising from an incorrect algorithm, and errors in the data. The often-quoted example of someone receiving a gas bill for £0 is an example (presumably) of an algorithm error: the program algorithm should include instructions about how to deal with the special case of a zero bill. On the other hand, someone receiving a bill for £10,000 is probably the victim of a data error: the wrong meter readings have been given to the computer. In MT too we can distinguish between mistranslations resulting from the program algorithm doing things in the wrong order, for example, and mistranslations resulting from the wrong information, e.g. a typographical error in the dictionary or a missing rule in the grammar.

Another issue connected to the distinction between algorithms and data is the question of 'declarative' and 'procedural' information. Declarative information is data which consist essentially of facts or definitions of objects and relationships; procedural information concerns what we want to do with those facts. A simple example will clarify: we can imagine the fact that the verb *eats* is the 3rd person singular present tense form of the verb *eat* is stored somehow in a 'lexical database' (see below). The relationship between *tats* and ear, and also the association of 'singular' and 'present tense' (etc.) to *eats* is declarative knowledge. There may be several different 'procedures' which use this knowledge in different ways. For example, there may be a procedure for discovering whether or not a certain word or phrase is a plausible subject of the verb *eats* (e.g. by testing if it is 3rd person singular): this would be an 'analysis' procedure (cf. section 1.2 and Chapter 5); deciding that *eats* translates into French as *mange* (because *mange* is the 3rd person singular present tense form of the verb *manger* just as *eats* is of *eat)* is another (used in 'transfer', cf. section 1.2 and Chapter 6); and finally finding out what the appropriate form of the verb *eat* should be if the tense is present and the subject is 3rd person singular is yet another procedure (used in 'generation', cf. section 1.2 and Chapter 7). In its simplest form, the distinction procedural vs. declarative corresponds exactly to algorithm vs. data. But it should be understood that in many MT systems, the procedural information is coded in a special formalism which must be then interpreted by the computer using a standard programming

language: so the procedural information coded in the formalism is still data, but of a different kind from the declarative data which the procedures make reference to. The possibility of distinguishing information which is neutral and can be used in different ways by different parts of the system is obviously advantageous: often, in MT systems, the linguistic information is mainly declarative in nature, while the computational side of the MT process represents procedural information.

## 3.3 Modularity

So far we have talked in terms of programs or algorithms for solving some problem without really considering what it means to solve a problem. In particular, some problems might be harder or bigger than others, and parts of the solution to one problem might be useful for the solution of another. For these two reasons, it is usual in designing algorithms to introduce an element of modularity into the design, that is, to divide the problem to be solved into sub-problems, and to divide those sub-problems further, until there is a collection of small, self-contained, relatively simple problems with correspondingly simple algorithms. Furthermore, it is to be hoped that these mini-programs can be used as building blocks, or modules, in the construction of other programs solving related but slightly different problems.

Modular programming has an additional advantage of making it easier to locate and correct algorithmic errors. If the computation is divided up into small, manageable blocks, each representing a specific and easily identifiable process, it is easier to see exactly where in the larger process something is going wrong. Furthermore, in the best cases, the modules can be developed and tested separately. This is an obvious engineering principle found in many spheres of life, and in computing no less than anywhere else.

In computational linguistics and MT, it is generally held that the modularisation should be linguistically rather than simply computationally motivated. It is considered desirable, for example, that the modules represent sub-tasks that linguists recognise as self-contained, rather than tasks which happen to be convenient for a computer program to treat at the same time or in the same way. When a system gets very large, it is important to be able to keep track of what each part of the system is intended to do; and it is helpful if the computational design of the system in some way reflects an intuitive linguistic approach to the problem. Furthermore, *ad hoc* solutions are avoided: the computational convenience of combining sub-problems may be vetoed by the linguistically motivated modularity requirement. The apparent advantages of combining notionally distinct sub-problems in one module may be forgotten or no longer understood at later stages when a system grows more complex.

Another important aspect of modularity is the relative independence of the modules. This is essential if they are to be tested separately. But equally, when the time comes, it must be possible to fit the modules together smoothly. An essential part of modularity is, therefore, the design of the interfaces between the modules, and here is where the important question of data structures comes in. This is true in general in computing, although we shall chiefly be interested in

the kind of data structures that are especially useful for computational linguistics and MT (sections 3.6 and 3,8).

## 3.4 System design

Before considering more specifically the field of computational linguistics, there is an issue which does not always get the attention it deserves. This is the question of **robustness** in system design. What is often neglected when designing a computational system is now wrong decisions or unsatisfactory implementations are to be put right.

There are two approaches. One is to incorporate into the design some sophisticated **debugging tools** ('bugs' are errors in the program) that help system designers to get 'inside' the system when it goes wrong, to see what is happening. perhaps by working through the program step by step until the problem is revealed. The second, which is not necessarily an alternative, but may go together with the first, is the idea of incremental system design involving rapid prototyping. An outline system is built up in short, self-contained stages, keeping at each stage those parts that are basically right, but throwing away those which turn out to be obviously incorrect. The prototype program gives a framework for subsequent elaboration and avoids the risk of building a whole system in detail which has to be abandoned and started again.

## 3.5 Problems of input and output

Earlier, we pointed out that the computer communicates with the outside world via various input/output devices. In this section we consider briefly the nature of this communication, with specific reference to the problems that occur in computational linguistics and MT.

In general, input/output can take two forms. On the one hand, we have the loading of data such as programs, dictionaries and grammars, and the manipulation of data structures. This is data handling of a computational nature, and clearly we can expect such data handling to be designed and developed in a 'computer-friendly' way. On the other hand, we have the input of the texts to be translated and the output of the result, at least one of which will be in a 'foreign' language; in addition, in certain types of system, there is communication with a human operator at the time of the computation, in what are called interactive systems. It is these two latter types of problem that we shall address here.

### 3.5.1 Interactive systems

We consider first the notion of interaction. A major development in computer architecture was the introduction of facilities to write a program which would stop what it was doing, send a message to some output device (usually a VDU screen), wait for the human user to respond (by typing something in), and then continue processing using the new data provided by the user. It means that computer programs can be **interactive** in the sense that they can rely on real-time

communication from the outside world to tell them how to proceed next. In fact, the majority of present-day computer programs are interactive, in some sense, when the computer offers a list (or **menu**) of possible things to do next, and waits for the user to choose. In more sophisticated systems, the information provided by the human might be more significant, and might be used in much more complicated ways. Typically in MT, interactions are used to supplement the data that the system already has in its dictionaries and grammars, e.g. by asking for some information about a word in the text which is not in the dictionary (is it misspelled? is it a proper name? is it a noun or a verb? and so on). Other types of interaction in MT might result from the fact that the system has computed alternative solutions to some problem, and asks the human to decide which is the right one.

A major task of research in this connection is to make such interactions **user-friendly**, which means ensuring that users understand quickly what they are supposed to do, are able to do it without too much difficulty, and, of course, give the right sort of answer. For example, consider a program which enables a dictionary to be built interactively by adding new words to it as they occur in texts. To do this, users might be asked whether a new word is a noun or a verb, and if it is a verb, whether it is transitive or intransitive. Two problems immediately arise: does the "noun or verb?" question apply to the word as used in this particular text, or does it apply to all possible uses of this word in general? Many words (in English at least) can be either nouns or verbs depending on the context. Secondly, the user may not understand the terms 'transitive' and 'intransitive' and may not be relied upon to give a 'correct' answer.

### 3.5.2 'Foreign' languages

Computer technology has been mostly developed in the USA, one of the most thoroughly monolingual societies in the world. For this reason, problems arising from the use of the computer with languages other than English did not at first receive much attention, and even today this is seen as a special ('abnormal') problem. Many languages of the world use either a completely different writing system from English, or else a slightly different variant of the English writing system, involving in particular special characters (German 'ß', Icelandic 'ð'), combinations of characters (Spanish *ll*, Welsh *ch*), or — most commonly — characters modified with accents or diacritics. Each of these deviations from the comfortable norm assumed by American computer pioneers leads to problems at various stages in the computational process.

Let us begin with the case of languages using the Roman alphabet with extensions. Apart from the obvious problem of finding a user-friendly way of permitting special characters or accented characters to be typed in, and adapting screens and printers so that they can be output (and these two tasks alone are not insignificant), there are problems involving the handling of such data within the computer.

The simple representation of all the different characters is not so difficult Obviously, computers can handle more than 26 letters: on a computer keyboard there are many additional characters besides the 26 lower and upper case letters of the alphabet: there are the numerals 0 to 9, some punctuation marks, '$','%',

'*' and so on. In fact, the typical computer can handle 64, 128 or 256 different characters easily (the numbers are powers of 2, and come from the fact that computers represent characters internally as binary numbers with 6, 7 or 8 bits), so there is generally enough 'space' for all the characters — at least for alphabetical writing systems (see below). Input and output is handled by low-level programs which convert the character codes into something that a human will recognise (e.g. for most computers, the character with the binary code 01000111 (i.e. 71 decimally) looks like this: G).

However, human users expect their character sets to have certain properties; in particular as having a certain in-built order, which is used, e.g., when consulting dictionaries. But it should be noticed that the order can differ slightly from language to language. For a Welsh speaker, *d* is the fifth, not the fourth, letter of the alphabet; for a French speaker, there are five different letters *(e, é*, *è*, *ê* and *ë)* which however count as the 'same' when sorting words into alphabetical order. In the Scandinavian languages, the letters *æ, å, ø,* and *ö* come at the end of the alphabet, making the consultation of dictionaries confusing for those unfamiliar with the practice. All these differences are problems for which computational solutions must be found.

For languages which use alphabetic scripts other than Roman (Greek and Russian are obvious examples), a whole new set of code-to-character correspondences must be defined, together with their in-built order. (It should be noted that we cannot just transcribe the Greek or Russian into the Roman alphabet; often, Greek and Russian texts contain words written in Roman characters: there has to be some way of indicating within the text when the character set changes!) For widely used languages there will already be suitable programs, but for some languages — notably where there has previously been no great economic incentive to design a computer character set — an MT system will have to find some other solution. Furthermore, not all languages with alphabetic scripts are written left-to-right, e.g. Arabic and Hebrew, so any input/output devices making this assumption will be useless for such languages.

Some languages do not even use alphabetical writing systems. The best known examples are Japanese and Chinese, with character sets of about 3,000 or 6,000 respectively (or even more for erudite texts). The special problems that these languages pose for computational linguistics have occupied researchers, especially in Japan and China, for many years. Devising codes for all the characters is not the primary problem, nor even designing printers and VDUs that can display them. Much more problematic has been the question of input: clearly, a keyboard with upwards of 3,000 keys is barely practical, so other approaches had to be explored. In fact, solutions have been found, and it is even arguable that in solving these problems, the Japanese in particular have learned more about non-numeric problem solving than their Western counterparts, and may now be ahead of them in many fields related to language processing.

## 3.6 Lexical databases

Turning now to more linguistic matters, we consider some of the computational aspects of handling language data. The first of these concerns dictionaries. Obviously, a major part of any MT system is going to be its lexical resources, that is to say, the information associated with individual words. The field of **computational lexicography** is concerned with creating and maintaining computerized dictionaries, whether oriented towards making conventional types of dictionaries accessible on-line to computer users, or whether oriented towards compiling dictionaries for use by natural language processing systems. In the case of MT, this latter activity is made all the more complex by the fact that we are dealing with more than one language, with bilingual and multilingual lexical data.

Lexical databases can be very large. It is difficult to say accurately how many words there are in a language, but current commercial MT systems typically claim to have general-language dictionaries containing upwards of 15,000 entries, and to have technical dictionaries of similar sizes. Although with such claims there is often uncertainty about exactly what counts as a 'word', these figures are a good indication of the order of magnitude for the dictionary of a working MT system.

The fact that the dictionary is very large means that designers of MT systems must pay particularly close attention to questions of storage, access and maintenance. Even with clever compaction routines for encoding data in compressed formats, dictionaries are still too big for storage in the central memories of computer systems. This means that MT systems have to find ways of partitioning the data and reading it in as and when necessary in a convenient manner. In practice, MT systems often have several different dictionaries, some containing frequently accessed entries, others containing rarer or more specialised vocabulary. Because dictionary access can be slow and complex, it is often preferable to do it just once at an early stage in the translation process. Whereas human translators will consult dictionaries at many various stages of translation, many MT systems have a single dictionary look-up phase, when all the relevant information associated with the particular words in the source text is read into a temporary area of memory, where it can be accessed quickly during the later translation processes.

A second consequence of the size of MT dictionaries is that their compilation demands great efforts. As we shall see, the content of a dictionary for use by a MT system is quite unlike the content of a dictionary for human use. Therefore, although there are now many commercially available on-line dictionaries and word lists, they always have to be worked on to make them suitable for incorporation in an MT system — and for many languages MT system developers still do not have the benefit of on-line word lists to start with. To reinforce this distinction, computational linguists often prefer to use the term lexicon to refer to the dictionary part of a system.

It is particularly important to ensure consistency within an MT lexicon, so computational lexicographers do not normally work alphabetically through a vocabulary but typically concentrate on some linguistically homogeneous set of words, e.g. transitive verbs, abstract nouns or the terminology of a subject field. Furthermore, the lexicographers have to assume that the linguistic information to

be encoded has been agreed upon and stabilized. The grammar writers might want to change the way a particular linguistic phenomenon is handled, but it could imply changing 2,000 or 3,000 dictionary entries! For this reason, many computational linguists believe that it is a good investment to develop computational tools which aid lexicographers to build new entries. Consistency and completeness can be facilitated by providing menus or templates for entering the lexical data which is required, e.g. the gender of a noun, whether it has an irregular plural, whether it is 'mass' or 'count', whether its case endings are similar to other nouns of a certain pattern, and so on.

As already mentioned, the lexicons used in computational linguistics and MT are quite different from what humans understand by 'dictionary'. A monolingual dictionary for the human user usually contains information about parts of speech, irregular inflected forms, a definition of meaning in some form, and (often) some indication of the history of the word. In bilingual dictionaries, we find lists of translation equivalents, often loosely subdivided, usually on semantic grounds, and, in the case of 'idioms', an example sentence from which the reader is expected to infer correct usage. In particular, if some grammatical feature of the source word happens to be also true of the target word, it is taken to be understood. For example, the English word *grass* is usually a mass noun and has no plural; however, we do find the form *grasses* when the implied meaning of the singular form is 'type of grass'. The same is true of the French word *herbe,* but this will not be stated explicitly. Dictionaries for human users avoid stating the obvious: it is assumed that we know that the object of the verb *eat* is normally something edible, and that if it is not, then the verb is being used in some 'metaphorical' sense.

By contrast the lexicons used in MT systems must contain the obvious: all information required by programs must be available. MT lexicons include information about parts of speech but in a much more explicit form, as we have shown already in Chapter 2 and as later examples will amply illustrate. Likewise, meanings — if they are coded at all — must be in a form which the computer can use, e.g. in terms of semantic features or selection restrictions (as also shown in Chapter 2); the kinds of definitions found in dictionaries for human use are not practical. Similarly, the bilingual lexicons for MT must express precisely the conditions under which one word can be translated by another, whether grammatical, semantic or stylistic, and in each case, the circumstances which license one or other translation have to be coded explicitly.

These differences will become apparent in the following sections on parsing and especially in later chapters when we discuss the linguistic problems of MT.

## 3.7 Computational techniques in early systems

It is appropriate at this stage to look briefly at the computational techniques used by the 'first generation' MT systems. As mentioned in section 1.3, it is often said that a relatively unsophisticated approach to the computational side of the problem of translation was one of the reasons for the comparative failure of the earliest MT systems. We shall look briefly at a typical example and its shortcomings.

Figure 3.1 shows an explanation of a procedure (actually coded in a low-level programming language) for translating *much* and *many* into Russian. The procedure is effectively a flow-chart, since each step includes an indication of which step to go to next, depending on whether the task at that step succeeds or fails. For example, the first step is to check whether the word before *many* is *how:* if so, go to step 2, otherwise go to step 3. Step 2 indicates the appropriate Russian word (with some additional grammatical information), and the 'go to 0' shows that this is the end of the procedure. Step 3 is a similar test, with 4 following if successful and with 5 following if not; and so on for the subsequent steps.

| | |
|---|---|
| 1(2,3) | Is preceding word *how?* |
| 2(0) | *skol'ko* (numeral, invariable) |
| 3(4,5) | Is preceding word *as?* |
| 4(0) | *stol'ko že* (numeral, variable) |
| 5(7,9) | Is current word *much?* |
| 6(0) | Not to be translated (adverb) |
| 7(6,11) | Is preceding word *very?* |
| 8(0) | *mnogii* (adjective, hard stem, with sibilant) |
| 9(8,12) | Is preceding word a preposition, and following word a noun? |
| 10(0) | *mnogo* (adverb) |
| 11(12,10) | Is following word a noun? |
| 12(0) | *mnogo* (adverb) |

Figure 3.1 Translation of *much* and *many* into Russian

Readers who have some familiarity with the BASIC programming language available on many microcomputers will immediately recognize this style of programming where you have to 'jump around' in the program. This makes it quite difficult to find out what is happening, and makes debugging (troubleshooting) particularly difficult But notice, in particular, that there is no easily discernible underlying linguistic theory: it is difficult to tell from looking at this program what the linguistic analysis of *much* is. Nor can the procedure be easily generalised for translating other, perhaps similar, words or sequences.

This style of MT programming was, of course, almost inevitable given the lack of high-level programming languages at the time. In the next sections, we consider the kinds of techniques that are now available and which overcome the weaknesses of these early attempts.

## 3.8 Parsing

A major component of computational linguistics in general, and hence a feature of the computational aspect of MT, comes under the general heading of parsing. This word refers to computer programs which take as data a grammar and a lexicon, as input a text (e.g. a sentence) and produce as output an analysis of

the structure of the text. In discussing parsing, it must be understood that several familiar words have a special meaning. By **grammar**, we understand a set of rules which describe what combinations and sequences of words are acceptable, i.e. whether a given input is 'grammatical'. Grammar rules are usually expressed in terms of the categories (or parts of speech) of the words, though the rules may also refer to features associated with words and phrases (see also sections 2.5.1 and 2.8.3). The information about what categories and features are associated with individual words is stored in the lexicon. Usually, too, the rules imply a structured analysis, where certain sequences of categories are recognised as forming mutually substitutable constituents or groups of words, which in turn combine to form bigger units (cf. section 2.8.2),

As an introduction to what this means in practice, consider a very simple grammar for (a subset of) English. As we saw in Chapter 2 the usual way to write grammars is as sets of **rewrite rules**:

(1a) NP → det (adj) n

(1b) NP → pron

(1c) S → NP VP

(1d) VP → v NP

Rule (1a) states that a noun phrase (NP) can consist of a det[erminer], an optional adj[ective] (as indicated by the brackets) and a noun (n). Rule (1b) is another rule for a noun phrase, indicating that a NP can consist of just a pron[oun]. The rules in (1c) and (1d) are for the recognition of other constituents, S[entence] and VP (verb phrase), where the latter consists of a v[erb] and a NP.

Rules of this nature are familiar to linguists (Chapter 2), but in the context of this chapter, we illustrate briefly how computer programs can be developed to take these rules and apply them to texts. Here only a flavour of some of the techniques can be given; a full account of all the possibilities would fill a book on its own (see section 3.10 below).

## 3.8.1 Top-down and bottom-up

In parsing techniques a basic distinction is made between top-down approaches and bottom-up approaches. In top-down parsing, the parser starts at the most abstract level and attempts to flesh out the structure by building downwards towards the lowest level, i.e. to the words themselves. In the case of our example above, this would mean that parsing starts with the node S and rule (1c). For this rule to apply, the parser must find a NP which is followed by a VP. But the NP requirement means that rule (la) or rule (1b) apply, and they mean in turn that either a pronoun or a determiner followed by an optional adjective and a noun have to be present in the input Likewise in order to satisfy the VP requirement there must be a verb followed by another noun phrase (again as defined by rules la or 1b). In bottom-up parsing on the other hand, the parsing starts with the words and attempts to build upwards. For example, take the input sentence:

(2) He loves the women.

The string of categories attached to the words is 'pron v det n'. The parser looks for rules that allow the combination of some of these categories into higher-level abstractions, applying rules perhaps in the sequence shown in (3).

(3) pron v det n

| | |
|---|---|
| NP v det n | using rule (1b) |
| NP v NP | using rule (la) |
| NP VP | using rule (1d) |
| S | using rule (1c) |

For the simple grammar above, the difference of approach may not seem important. It becomes significant, however, when there are large grammars and many potentially ambiguous sentences. Suppose we enlarge our grammar as follows (4) in order to account for prepositional phrases (PP) and to deal with other possible types of noun phrases (NP):

(4a) S → NP VP PP
(4b) S → NP VP
(4c) VP → v NP
(4d) NP → det n
(4e) NP → det adj n
(4f) NP → det n PP
(4g) NP → det adj n PP
(4h) NP → pron
(4i) PP → prep NP

We now see that there are effectively two different types of S (with or without the PP) and five basically different types of NP, two of which involve a PP, which in turn involves a NP. Because this grammar contains such **recursive** rules, it predicts an infinite variety of possible structures. When parsing in a top-down mode, there are immediately two alternative predictions about what the S will be, and both start with a NP, for which there are in turn five possibilities. Even before anything is consulted in the lexicon there are ten possible analyses, any number of which could turn out to be completable,

## 3.8.2 Backtracking

In both top-down and bottom-up parsing, it is not always the case that the first decision taken is the right one. If we consider using the grammar in (4) to parse (2) in a top-down manner, applying the rules in the order they are given, we can see that rule (4a) entails a 'false start': it will not lead to a successful parse because there is no PP in (2). Similarly all (4d-g) will lead to failure since the first NP in (2) corresponds to the structure given by (4h). Even with this simple example it can be seen that the computer has to be able to revise any 'decisions' it may have taken. This is called backtracking. To illustrate the need to backtrack in the case of a bottom-up parser, consider an input string which is structurally ambiguous, such as (5). By 'structural ambiguity' is meant that the rules allow

two alternative solutions depending, in this example, on where the PP *(with a telescope)* fits into the structure.

(5) He saw the girl with a telescope.

In one case, the NP consists of just *the girl* (i.e. rule 4d is applied) and the VP is *saw the girl* (rule 4c); as a result, the parser recognises that the PP is part of an S of the type NP VP PP (rule 4a). In the other case, the PP is taken to be part of the NP *(the girl with a telescope)* by applying rule 4f; the resulting NP is part of a VP (rule 4c again) and the parser will recognise an S of the NP VP type, by rule 4b. Again if the rules are applied in the order in which they are given (this time looking at the right-hand-side to see if the rule is satisfied), the first solution reached will be the one predicted by (4a). This will be because in processing *the girl...*, the NP defined by rule (4d) will be successfully identified before recognition of the NP defined by (4f). In order to obtain both solutions, the process will have to backtrack to the point at which rule (4d) applied and test whether any other rules might also be applicable.

There are essentially two ways of causing a parser to consider all possible solutions, depending on the strategy adopted each time a choice of alternatives is encountered. On the one hand, the parser can take one possible solution to its ultimate conclusion (successful parse or 'false trail') and then return to the last choice point and try the alternatives, until each of these is exhausted. In each case, successful paths must be somehow 'remembered'. This approach is called **depth-first** parsing. The alternative is a strategy where all options are explored in parallel, so that both false and true trails are kept 'alive' until the last possible moment. This **breadth-first** approach means that there is no need for backtracking as such.

### 3.8.3 Feature notations

On the basis of the kinds of rules in (4), parsing alone cannot decide whether an analysis is plausible or not; all possible combinations of categories and groupings are attempted whatever the context (cf. also section 2,9.1 on context-free grammars). A parser applying the rules in (4) would find both sentences in (6) acceptable, even though they contain what we (as English speakers) recognise as simple grammatical errors.

(6a) They loves the women.

(6b) The man love the women.

One way of making parsing more subtle is to introduce **feature notations** on rules. For example we might further specify that the NP which is the first constituent of an S (the subject) must agree in number with the VP, thereby accounting for the acceptability of (2) as against (6a) and (6b). In order to do this, we need to use 'variables' in the rules. A **variable** (used here in the programming sense) is something which stands for or represents a real object: it is a slot waiting for a value to be assigned to it. An obvious example of the use of variables comes from algebra, where the *x*s and *ys* in equations are variables which stand for numbers. In programming languages, variables are effectively locations in the computer's memory area where values can be stored temporarily, for the purpose of some calculation. For example, when we say "Think of a number; now double

it...", this is equivalent in computational terms to assigning a numerical value to a variable — *x,* say — and then calculating the value of 2 x *x*

In computational linguistics, we often use variables to stand for words, or grammatical categories, or features; in the case of the grammar in (1), the specification of number agreement between subject and verb can be stated as in (7a); the number feature of the NP comes from its n (7c) or pron (7d), while the number feature of the VP comes from its v (7b).

(7a) S → NP[num=$X] VP[num=$X]

(7b) VP[num=$Y] → v[num=$Y] NP[num=?]

(7c) NP[num=$Z] → det (adj) n[num=$Z]

(7d) NP[num=$Z] → pron[num=$Z]

It is always important to distinguish variables from real values, and different programming languages or rule-writing formalisms have different conventions for doing this. A common convention is to preface variables with a special symbol such as '$' (as we have done here), or '_'. In some formalisms the use of capital or lower case letters is distinctive. The important point is that once a value has been assigned to a variable, it keeps that value for the rest of the rule; because similarly-named variables in different rules will not necessarily have the same value, it is often good programming practice to use different variable names in neighbouring rules, so as to avoid confusion.

To illustrate how the rules in (7) operate, consider a sentence such as (8).

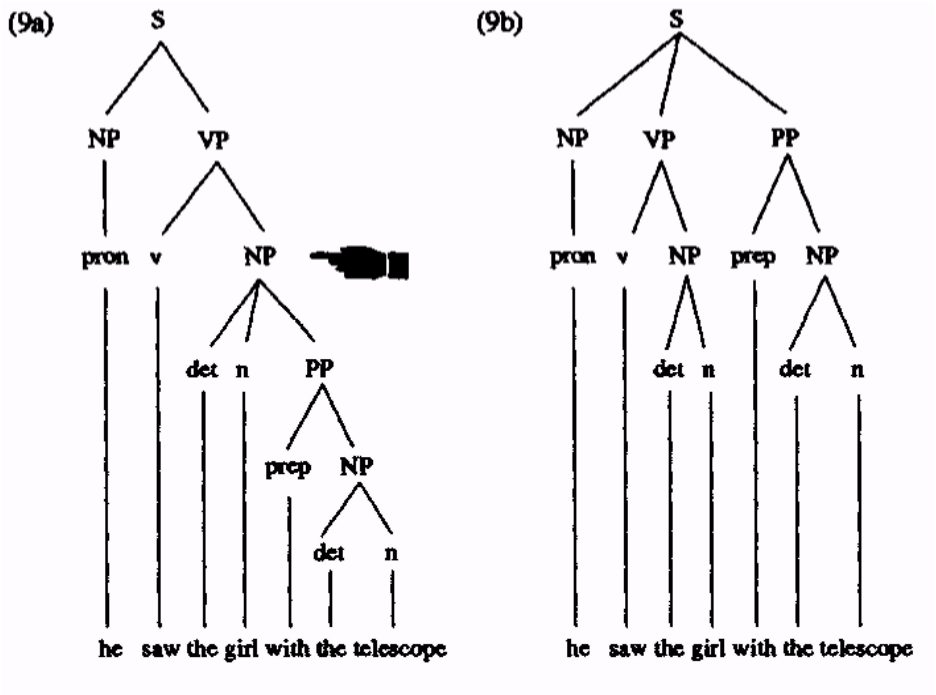
(8) The women like nun.

We see that the value of $X in (7a) must be the same for the NP the *women,* and for the verb *like*. In a top-down mode, the value for 'num' will be set by the completion of rule (7c), where it is the word *women* that tells us the NP is plural. Now the processor will be looking for a plural VP, and will find it in *like him,* by dint of *like* being plural (7b). Notice that the value of 'num' for the object NP *him* in (7b) is not relevant here, even though it has been calculated by the NP rule (7d). In a bottom-up parser, the feature constraints act as an additional filter on the rules. So for example in *a man like a bear,* although *a man* is an NP, and *like a bear* is a possible VP, they do not agree in number, so (7b) fails, and, presumably, some other rule identifies the whole thing as an NP, with *like* functioning as a conjunction.

As we shall see, MT systems often use formalisms which are even more complex than this, though the details need not concern us for the moment

## 3.8.4 Trees

In these examples of parsing, the end results are not very informative: the sentence is or is not grammatical, is or is not syntactically ambiguous. For MT we need more information than that: specifically, we need to know what the internal structure of the sentence is. To do this, it is necessary, while parsing, to make a record of which rules were used, and in which order. This is the purpose of the tree structures familiar to linguists (cf. section 2.8.2). If we take again our ambiguous example (6), there are two parse trees associated with this sentence as in (9).




These two trees capture the fact that in (9a) the NP rule (40 has been used which includes a PP to build the right-most constituent at the level marked with a pointer, whereas in (9b) the PP is 'attached' at a higher level using rules (4d) for the NP and (4a) for the S. These two trees represent the two different analyses in the sense of recording two different 'parse histories'. In linguistic terms, they correspond to the two readings of the sentence: one in which the PP is part of the NP (i.e. the girl has the telescope), and the other where the PP is at the same level as the subject (i.e. the man has the telescope). As we shall see in later chapters, these two competing analyses may lead to different translations.

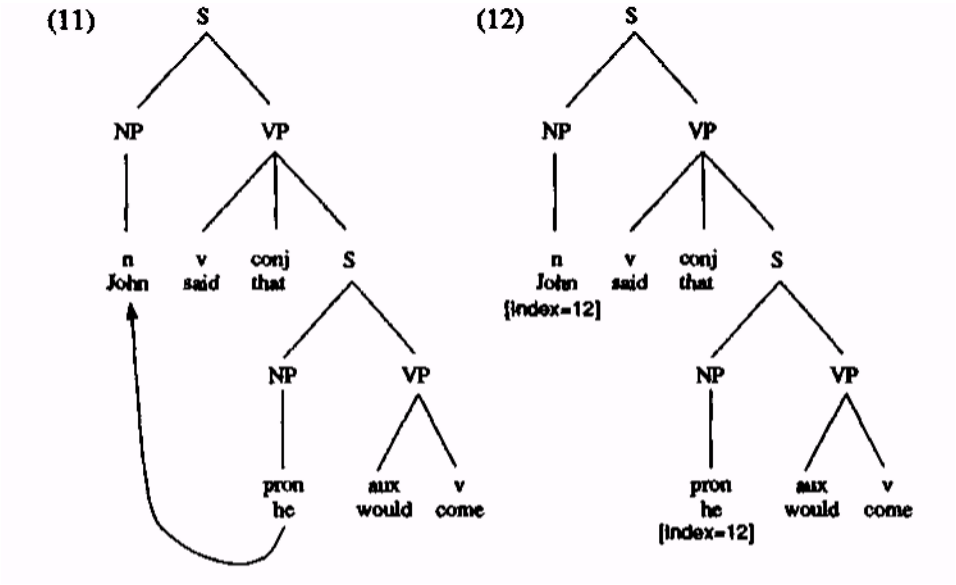
For convenience, a bracketed notation for trees is sometimes used: the equivalents for the trees in (9a) and (9b) are shown in (10a) and (10b) respectively.

(10a) S(NP(pron(he)),
VP(v(saw),NP(det(the),
n(girl),
PP(prep(with),NP(det(the),n(telescope))))))

(10b) S(NP(pron(he)),
VP(v(saw),NP(det(the),n(girl)),
PP(prep(with),NP(det(the),n(telescope)))))

The tree structures required may of course be much more complex, not only in the sense of having more levels, or more branches at any given level, but also in that the labelling of the nodes (i.e. the ends of the branches) may be more informative. This might be the case where feature annotations are included, e.g. adding information about the grammatical nature of the constituents — singular or plural, gender, etc., as illustrated in (7) above. Quite often, we also need to indicate relations between branches of the tree, e.g. to show that *he* is the subject

of the verb in examples (5) and (6) above, that a verb and a particle realty belong together (e.g. *pick* and *up* in *pick the book up),* or that a pronoun is referring to another noun phrase in the same sentence (e.g. *John* and *he* in *John said he would come).* In order to do this, the tree structure must become more complex, with 'pointers' leading from one branch to another (11), or with 'co-indexed' branches, perhaps with arbitrary numerical values (12).




Tree structures are widely used in computational linguistics and MT, not only for showing the results of parsing, but also for other representations of texts. In particular, it is quite common for the tree structure resulting from a syntactic parse to be manipulated by further rules into a tree structure suitable for generation in the target language. The procedure is reminiscent of the transformation rules familiar to linguists (section 2.9.2), and requires very powerful rule formalisms which define tree-to-tree mappings (often referred to as 'tree-transduction'), i.e. which define the circumstances under which one tree structure can be transformed into, or mapped onto, another. Examples of such rules are given particularly in Chapter 6.

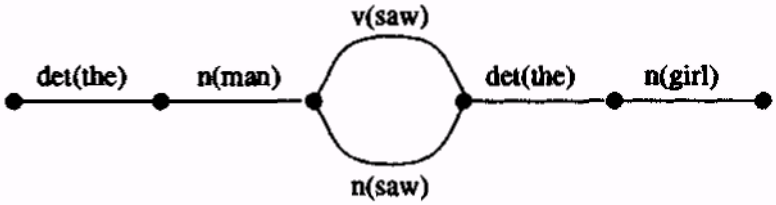
## 3.8.5 Charts

Trees are not the only data structures used in computational linguistics or in MT systems. A particularly useful data structure found in a number of systems is the chart. Its value is that it allows alternative solutions to be stored simultaneously in a convenient and accessible manner. It is therefore a standard way of implementing a breadth-first parsing strategy (see 3.8.2 above).

Roughly speaking, a chart is a set of points or nodes with labelled arcs (sometimes called 'edges') connecting them. The nodes are in sequence ('ordered'), and the arcs can go in one direction only ('forward'). Two of the nodes have special status, namely the first, or 'entry' node, which has no arcs leading into it, and the

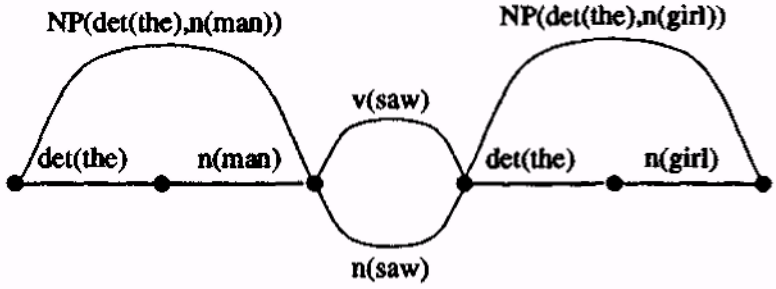
last, or 'exit' node, which has no arcs leading from it When used in parsing, the chart is set up initially with one arc for each possible reading of each word of the sentence, as in (13) for the beginning of (6); notice that the potential category ambiguity of the word *saw* is represented by two arcs labelled with the alternate possibilities.

(13)

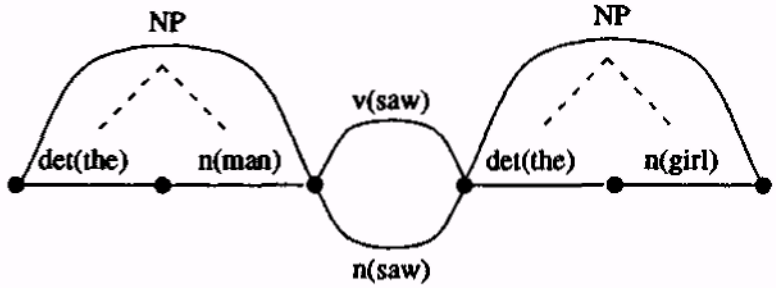



As each rule is applied, a new arc is added, connecting the outermost nodes of the constituent that is built, and labelled with the appropriate tree-structure. In (14) we can see the chart after NP rules have been applied. The labelling can consist of copies of the trees — usually given in the bracketed notation — as in (14a), or of 'pointers' to the arcs which have been subsumed, as in (14b).

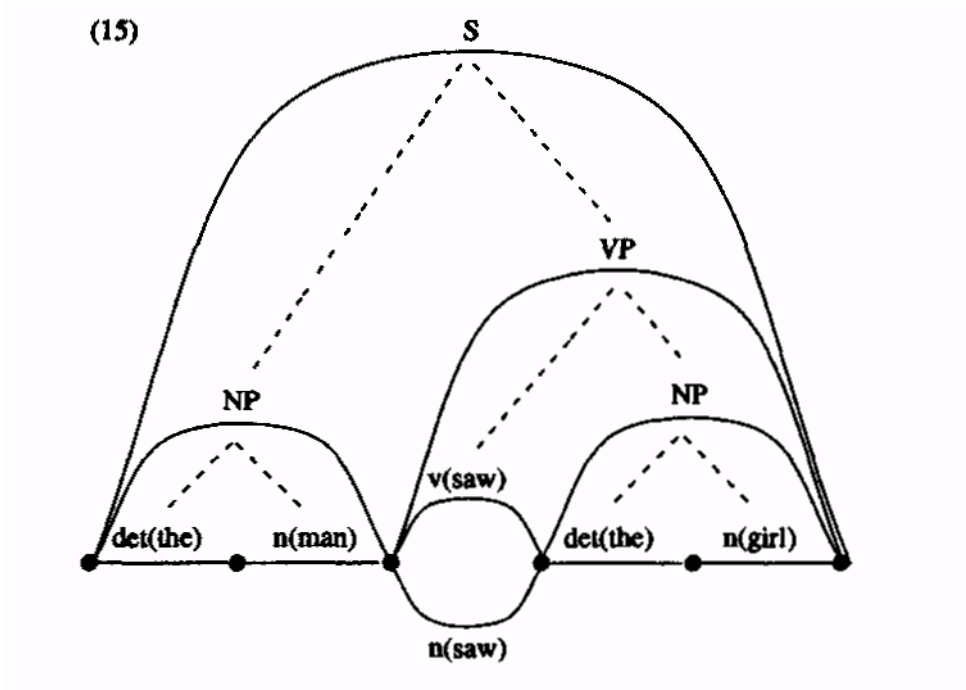
(14a)




(14b)



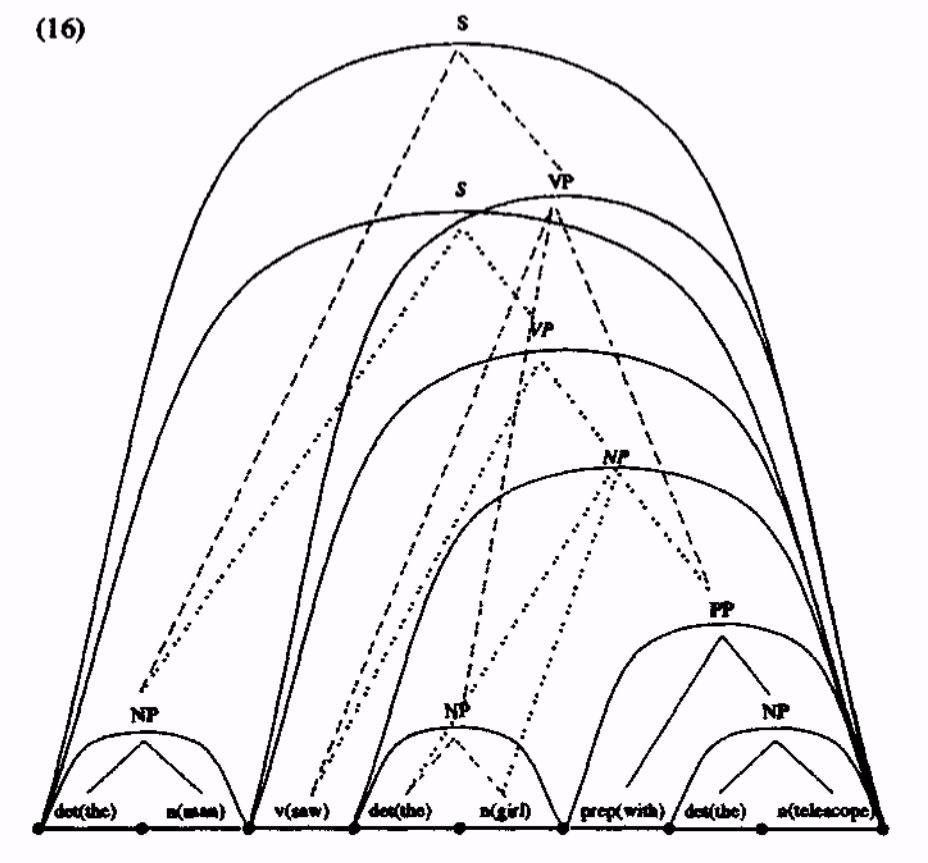
Continuing in this way, applying the other rules, a complete parse is represented by a single arc spanning the whole chart and labelled S, as in (15).




It should he noted that the arc labelled with the a reading for *saw* is present in each of the charts, even though it is not used in the construction of further arcs. In (16) (opposite) we give the (very complex) chart which would be built for the ambiguous sentence (5): there are two S arcs corresponding to the two readings, each with different substructures. (In fact, the chart has been simplified by leaving out some arcs which would be built but which would not be used, e.g. the arc for the n reading of *saw* and a false S arc spanning *the man saw the girl.)* A major advantage of the chart approach is that the parts of the two structures which are common to both readings are represented just once, and are 'shared'. These include all the lexical arcs on the 'bottom', and the arcs labelled in bold and drawn with solid lines. The two readings are otherwise distinguished by italic arc labels and dotted lines ('.. ..') for the reading 'girl with the telescope', and normal labels with broken lines ('- -') for the reading 'saw with the telescope'.

## 3.8.6 Production systems






Closely connected to the use of a chart as a data structure is the technique for implementing grammars called a **production system.** The idea is that the rules which say which sequences of arcs can lead to the construction of new arcs are all put together in a single database, in no particular order. This is the **rule set,** and the rules, or 'productions', can be thought of as 'condition-action' pairs, where the condition part is a 'pattern match' on the chart, and the action is the building of a new arc. By a **pattern match** is meant that the computer looks to see if the sequence of arcs specified by the rule can actually be located in the chart. It is up to the computer, and in particular the so-called interpreter part of the production system, to decide which rules can be applied and when. It should be noticed that the rules are **non-destructive** in the sense that, after a rule has been applied to a certain sequence of arcs, the old arcs are still there, and available to other rules. It means that care has to be taken that the interpreter does not apply the same rule more than once to the same sequence of arcs. Obviously, when many alternative rules might apply at the same time, the whole process can be very complex, especially when all combinations of rule sequences have to be tested. The crucial point is that the computational tasks of pattern matching and rule testing can be separated from the essentially linguistic tasks of stating

which conditions can actually trigger which actions, i.e. separating the work of the computer and the work of the linguist, as described in section 3.2 above.

Finally, it should be noted that this architecture can be used not only for parsing but equally for other MT processes. The condition part of a production rule can be arbitrarily complex, it can specify the arcs to be looked for in as much detail as necessary. The only constraint is that the arcs must be contiguous. There is also no requirement that rules refer to sequences of more than one arc. Thus, for example, 'transformational' rules can be written which take a single arc with a specific labelling, and build a new arc with the same start and end points, but with different labelling. More detail about a particularly influential production system approach in MT is given in section 13.5 on the GETA-Ariane system.

## 3.9 Unification

In section 2.9.7 we described linguistic theories which make use of 'unification' of feature-based representations. This approach is closely connected with the emergence of **unification** as a computational technique, as found in programming languages such as Prolog, now much used in computational linguistics and MT. In fact, there are some important differences between unification as a computational device and unification as understood by linguists, which we will clarify below. However, the basic idea is similar in both cases.

Unification is essentially an operation carried out on data structures, and depends crucially on the idea of variables, described above. In the examples in (7), we saw variables used for feature values, but they could equally be used to represent words or whole structures. The idea of unification is that two structures can be unified if (and only if) they are identical, once values have been assigned to variables. So for example the structure with its accompanying conditions in (17a) — where variables are indicated in capital letters, preceded by '$' — can be unified with the structure in (17b), with the variables instantiated accordingly:

(17a) np(det($DET),n($N))
*if* number($DET)=$NBR *and* number($N)=$NBR *and*
gender($DET)=$GEN *and* gender($N)=$GEN

(17b) np(det(la),n(table))
*where* $DET="la", $N="table", $NBR=sing, $GEN=fem

An entire programming style has been developed based on this idea, and is found in the much-used programming language Prolog. A parser written in Prolog consists of a set of descriptions of well-formed structures containing variables which are given values according to associated rules. In fact, the rule in (17a) is very similar to a Prolog 'clause'. If we put it together with some other rules written in the same formalism (which is not pure Prolog, but could easily be translated into it), you will see how a simple parser might be built (18).

(18a) s($A,$B)
*if* $A=np(...) *and* $B=vp(...) *and* number($A)=number($B)

(18b) np($D,$N)
*if* $D=det(...) and $N=n(...) *and* number($D)=$NBR *and*
number($N)=$NBR *and* gender($D)=$GEN *and* gender($N)=$GEN

(18c) vp($X)
*if* $X=v(...)
(18d) vp($Y,$Z)
*if* $Y=v(...) *and* transitive($Y) *and* $Z=np(...)

In MT systems which use this idea, the structures which are unified in this way can be much more complex; in particular, the nodes of the trees represented in the rules can carry much more information, in the form of bundles of **features** (see section 2.8.3), which the very powerful but intuitively simple unification mechanism can operate on. In this way, complex data structures can be built up and manipulated by combinations of rules which individually express only simple relationships. Here the idea of the production system, described above, can be brought in: the 'rule set' is made up of independent rules about feature compatibility and tree structure, and there is an interpreter which figures out in which order the rules should be applied. The whole process continues until some specific configuration is arrived at, or no more rules apply.

## 3.10 Further reading

Within the space of one chapter we cannot cover, even superficially, all aspects of computer science, and not even all of computational linguistics. However, many of the available text books on computational linguistics assume a computational rather than a linguistic background in their readers, so the best sources for clarification and elaboration of the issues brought up here are general computer science textbooks, of which there are hundreds.

Here we give mainly further references for some of the more particular points made.

The question of character sets and processing languages which do not use the Roman alphabet is a major concern of researchers in the field of linguistic computing (as opposed to computational linguistics), and is addressed in journals such *Literary and Linguistic Computing, Computers & the Humanities,* and other journals. On the question of alphabetization, see for example Gavare (1988).

Computational lexicography is similarly a huge field. Two collections of papers, Goetschalckx and Rolling (1982) and Hartmann (1984), give a good view of current issues.

The procedure shown in Figure 3.1 is from Panov (1960).

For a discussion of parsing, see almost any computational linguistics textbook. De Roeck (1983) is a very simple introduction to some basic concepts, and Winograd (1983) is quite readable. For rather more advanced discussions, see King (1983), Sparck Jones and Wilks (1983), Thompson and Ritchie (1984), Dowty *et al.* (1985), Grosz *et al.* (1986) and Whitelock *et al.* (1987).

Charts and chart parsing in particular were first developed by Kaplan (1973) and Kay (1976). Winograd (1983) and Varile (1983) deal with the topic in a perspicuous manner.

Production systems were developed for use in cognitive modelling by Newell (1973). For a brief discussion see Davis and King (1977) or Rosner (1983);

for more detail see Waterman and Hayes-Roth (1978) or Stefik *et al.* (1982). The classic example of a production system for the purposes of MT is the system developed by Colmerauer (1970), the so-called 'Q-systems' formalism (see Chapter 12), which is in fact the forerunner of the programming language Prolog.