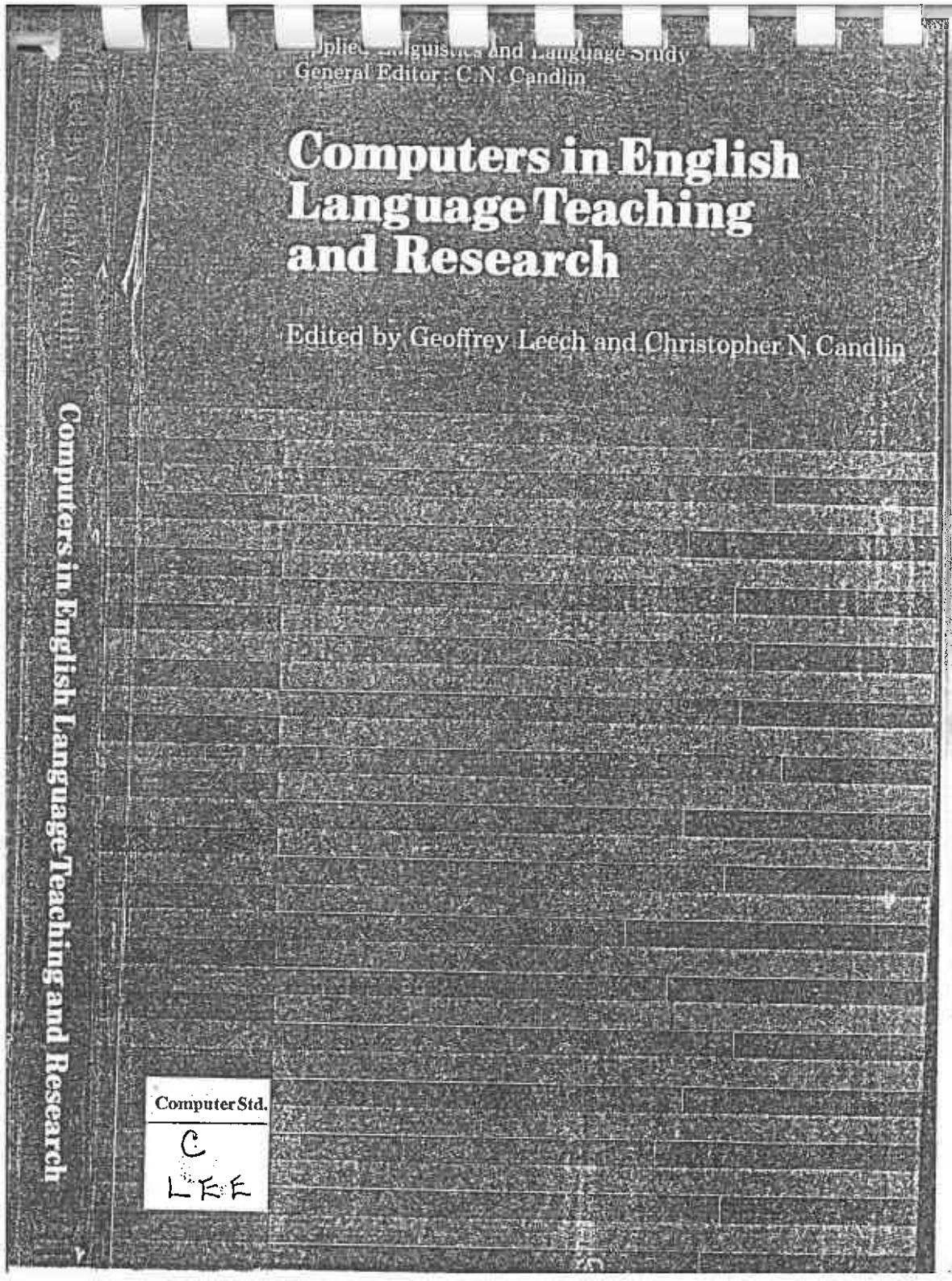
**Book Section:**
Atwell, ES (1986) Beyond the micro: advanced software for research and teaching from computer science and artificial intelligence. In: Leech, G and Candlin, C, (eds.) Computers in English language teaching and research : selected papers from the 1984 Lancaster symposium. Longman , 167 - 183. ISBN 0582550696




LECT PAP FR THE RITY COU S DSIL ON
"COMPUTERS IN ENGLISH LANGUAGE EDUCATION AND RESEARCH",
LANCASTER (PUBLISHED BY LONGMAN, 1986)

# Introduction to Chapter 12

Although, as Eric Atwell points out, the computing power required by workers in artificial intelligence or computer science in general far outstrips what can be expected to be at hand for language teachers in school interested in CALL, it is important not to be dismissive of the facilities presently available to the latter, especially given the rapid increase of power even in microcomputers. From this more optimistic position, then, Atwell provides readers of this collection with a very valuable service indeed. He documents and illustrates for the user concerned with language education how already available systems and programs can be put to use by the teacher, by the applied linguist, and, in principle, by the learner.

Atwell begins by exploring operating systems (VMS and UNIX), showing how they can be used to accommodate several users simultaneously, handling a series of complex tasks. There is obvious value here for the educational institution wanting to overcome the problem of accessibility and varied learning tasks. Associated with his discussion of operating systems, Atwell provides a glossary to a range of programs useful to the language teacher and learner, focusing especially on text editors, spelling checkers and style 'improvers'. One very valuable suite of programs discussed here is *The Writer's Workbench*, with language analysis facilities as immediately useful in the classroom as they have proved to be for independent authors. Once again we detect the growing move towards enabling learner autonomy with the aid of available software not necessarily conceived with the language learner in mind.

Finally, in this paper, Atwell tackles one of the problems heralded in earlier papers in the collection, that of the need to augment BASIC as the main programming language for micro users. As he points out, the usefulness and increasing availability of ADA, LISP, PROLOG, POP-11 and POPLOG should greatly increase the range of languages which can be made use of in the near future, even in relatively limited computing environments, by researchers into language analysis and language learning.

# 12 Beyond the micro: advanced software for research and teaching from computer science and artificial intelligence

Eric Atwell

Lecturers and researchers in British university and polytechnic computer studies departments generally have access to rather greater computing power than most linguists or English language teachers. As well as a range of micros, the computer scientist can use powerful multi-user *mainframes* and single-user *workstations* with very fast program execution speeds, a megabyte or more of main memory, hundreds or thousands of megabytes of disk memory, and correspondingly sophisticated software. However, because of the phenomenal rate of progress in computing, the computer science and artificial intelligence research and teaching tools of today may well be within the reach of other researchers and teachers tomorrow, or at least in a couple of years' time. This chapter looks at some of this software, concentrating on facilities which seem particularly relevant to English language teaching and research.

## Sophisticated operating systems

The operating system is the main program resident in a computer, which schedules and organizes the use of processing resources in such a way that the user is unaware of the 'nitty gritty' of how the hardware actually works at a low level. For a micro user, the operating system is used mainly for file-handling: for instance, the user can load a file from a disk by issuing a simple command, without having to know anything about how blocks of binary code are read and interpreted, or about the detailed sequence of signals sent between the processing unit and the disk drive. On a modern mainframe, the operating system has to be much more sophisticated, to cater for a large number of



users with widely varying requirements. Deitel (1983) explains this in more detail; here, we look at some of the most relevant aspects of advanced operating systems.

### VMS and UNIX

VMS and UNIX are two of the most widely available and popular operating systems in British universities and polytechnics today. VMS is available on VAX mini and mainframe computers; its great attraction is that, although a sophisticated multi-user operating system with a very wide range of facilities, it is very user-friendly to beginners and casual users. Wherever possible, commands are English words, e.g. PRINT to print out a file on the printer; SHOW TIME to display the current date and time on the terminal; SET PASSWORD to allow the user to set (or change) his or her password. As more experienced users wish to save keystrokes, all commands can be abbreviated to their first few letters, so long as this remains unambiguous (e.g. PRINT can be abbreviated to PRIN, PRI, or PR, but not P alone as other commands also start with P).

UNIX is also available on VAX computers, and on a wide range of other machines as well. UNIX was originally developed at Bell Laboratories; at the time, most operating systems were built round specific computers, and Bell researchers decided to develop a 'machine-independent' operating system tailored for the user rather than the machine. Bell still hold the trademark rights for UNIX, but the system was so good that UNIX-look-alike systems have been or are being developed by most other computer manufacturers. UNIX has the reputation of appearing rather strange to the beginner, mainly because many commands are short mnemonics whose meanings are far from obvious to the newcomer; for example, 'wc' (short for Word Count) counts the number of words and/or letters and/or lines in a document; 'mv' (MoVe) is used to change the name of a file; 'who' displays a list of all users currently logged-in to the system. However, once the user has learnt a basic vocabulary (a task which should not particularly daunt the linguist, armed with a UNIX primer such as Miller and Boyle (1984) or Bourne (1982)), UNIX turns out to be an ideal environment for developing software, as it has a particularly rich set of 'tools' and facilities to aid the developer.

### Concurrency

With a multi-user operating system, many users can interact with the computer apparently simultaneously. For this to be possible, the

# 12 Beyond the micro: advanced software for research and teaching from computer science and artificial intelligence

Eric Atwell

Lecturers and researchers in British university and polytechnic computer studies departments generally have access to rather greater computing power than most linguists or English language teachers. As well as a range of micros, the computer scientist can use powerful multi-user *mainframes* and single-user *workstations* with very fast program execution speeds, a megabyte or more of main memory, hundreds or thousands of megabytes of disk memory, and correspondingly sophisticated software. However, because of the phenomenal rate of progress in computing, the computer science and artificial intelligence research and teaching tools of today may well be within the reach of other researchers and teachers tomorrow, or at least in a couple of years' time. This chapter looks at some of this software, concentrating on facilities which seem particularly relevant to English language teaching and research.

## Sophisticated operating systems

The operating system is the main program resident in a computer, which schedules and organizes the use of processing resources in such a way that the user is unaware of the 'nitty gritty' of how the hardware actually works at a low level. For a micro user, the operating system is used mainly for file-handling: for instance, the user can load a file from a disk by issuing a simple command, without having to know anything about how blocks of binary code are read and interpreted, or about the detailed sequence of signals sent between the processing unit and the disk drive. On a modern mainframe, the operating system has to be much more sophisticated, to cater for a large number of



users with widely varying requirements. Deitel (1983) explains this in more detail; here, we look at some of the most relevant aspects of advanced operating systems.

### VMS and UNIX

VMS and UNIX are two of the most widely available and popular operating systems in British universities and polytechnics today. VMS is available on VAX mini and mainframe computers; its great attraction is that, although a sophisticated multi-user operating system with a very wide range of facilities, it is very user-friendly to beginners and casual users. Wherever possible, commands are English words, e.g. PRINT to print out a file on the printer; SHOW TIME to display the current date and time on the terminal; SET PASSWORD to allow the user to set (or change) his or her password. As more experienced users wish to save keystrokes, all commands can be abbreviated to their first few letters, so long as this remains unambiguous (e.g. PRINT can be abbreviated to PRIN, PRI, or PR, but not P alone as other commands also start with P).

UNIX is also available on VAX computers, and on a wide range of other machines as well. UNIX was originally developed at Bell Laboratories; at the time, most operating systems were built round specific computers, and Bell researchers decided to develop a 'machine-independent' operating system tailored for the user rather than the machine. Bell still hold the trademark rights for UNIX, but the system was so good that UNIX-look-alike systems have been or are being developed by most other computer manufacturers. UNIX has the reputation of appearing rather strange to the beginner, mainly because many commands are short mnemonics whose meanings are far from obvious to the newcomer; for example, 'wc' (short for Word Count) counts the number of words and/or letters and/or lines in a document; 'mv' (MoVe) is used to change the name of a file; 'who' displays a list of all users currently logged-in to the system. However, once the user has learnt a basic vocabulary (a task which should not particularly daunt the linguist, armed with a UNIX primer such as Miller and Boyle (1984) or Bourne (1982)), UNIX turns out to be an ideal environment for developing software, as it has a particularly rich set of 'tools' and facilities to aid the developer.

### Concurrency

With a multi-user operating system, many users can interact with the computer apparently simultaneously. For this to be possible, the

operating system must be able to organize a large number of processes running *concurrently*. This is actually achieved by a technique called *interleaving*: the system maintains a list of processes currently running 'simultaneously'; each process on the list can take a turn at getting the system's full attention for a few milliseconds, and then it must stop and wait until the system cycles round to its next turn. The problems of concurrent programming are dealt with in more detail in Ben-Ari (1982). Of course, the details of concurrency are kept hidden from users, so that an individual logging-in appears to have a *virtual machine* all to himself or herself. However, there are times when a single user would like to be able to run more than one process at a time, and UNIX makes this particularly easy. Firstly, adding '&' to the end of a command line causes the command to be run as a second process, concurrently with the main log-in process. For example, if a user has a program SLOWPROG which takes a long time to run, and he or she also wants to do some interactive work such as editing files, etc., then in most operating systems he or she will just have to wait until SLOWPROG has finished before typing any more commands; but in UNIX, the line

slowprog &

will cause SLOWPROG to be run as a separate process, leaving the terminal free for issuing other commands. Another use of concurrency is when a complex task is broken down into several programs, each passing their results on as input to the next program. For example, if a user wanted to count the number of spelling errors in a file MYTEXT, then he or she could run a spelling-check program over MYTEXT to produce an output file TEMP containing a list of errors, and then run a word-counting program over TEMP to output the number of words in this file. Under UNIX, there is no need for a temporary file between each process: programs can be *pipelined* to run concurrently and pass results directly from one process to the next, e.g.

spell mytext 1 wc

This means that quite complex tasks can be performed simply by pipelining the appropriate tools together; this facility is not available on other operating systems.

## General-purpose tools

Even in widely varying applications areas, there are a number of comparatively simple tasks that many users will carry out repeatedly; and so an advanced operating system like VMS or UNIX will include a number of general-purpose *tools* or *utilities* for such tasks. For example, often users may need to *sort* a list of numbers into ascending or descending order, or *sort* a list of words into alphabetical order; rather than have to write their own sorting program, they can use the SORT command to sort the file containing the list. Another useful tool is a command to list the *differences* between two files; for example, if a user writes a program which produces an output file OUTPUT1, and later amends the program and runs it again to produce an output file OUTPUT2, then he or she could straightforwardly uncover any differences between the files caused by the change in the program. A third commonly-available tool is a command to COUNT the number of lines and/or words and/or characters in a file.

### Text editors

These simple tools simply take one or more files as input, do some standardized processing, and produce the appropriate output file. More sophisticated tools allow the user interactive control over the processing. For example, all operating systems have at least one *editor* for editing the contents of files (many systems have several alternative editors!). The simplest type of editor is a *line editor*: the user edits the file by typing a sequence of editing commands, effectively a 'program' which is interpreted and executed interactively. Unfortunately, the file itself is *not* directly visible to the user unless he or she explicitly displays lines using the appropriate command; this of course is not particularly helpful, so nowadays many operating systems also offer a *screen editor* as an alternative to line editors. With a screen editor, the text of the file itself is displayed on the terminal screen, and the user can 'move around' in the file by positioning the cursor with special arrow keys; mainframe screen editors are much closer than line editors to the text editors available on home micros and word-processors.

### Help

Another very important tool is the *help* system. On most mainframe operating systems, it is possible to find out how to use a command by typing HELP followed by the command, e.g.

HELP SORT

This will cause information on the SORT command to be displayed. With a good help system, a beginner can use the facilities of an

operating system without constantly having to refer to printed documentation and reference manuals to check the exact spelling and usage of commands; as the reference manuals accompanying a modern operating system can fill a large bookcase, this is particularly important!

## UNIX tools for language analysis

The UNIX operating system is particularly well-endowed with tools which could be very useful to English language teachers and researchers. Apart from the general-purpose tools mentioned above which could be adapted for linguistic applications, UNIX includes a number of tools specifically designed for analysis of English language text files. In this section we examine these in detail. Of course, other operating systems may well have utilities roughly equivalent to some of the UNIX tools below (in particular, several alternative 'spelling-checkers' are available), but none that I know of have all of them.

### GREP

GREP prints out all lines in a file containing a specified string, e.g.

```
grep 'spell' myfile
```

This will print out all lines in the file MYFILE containing the string *spell* (including, e.g. *spellbound* and *misspelling*, but not *spelt*). In fact, the files searched by GREP need not be English text files, but this application is obviously the most relevant for the English language teacher or researcher.

The string searched for can contain various wildcards, e.g. a period '.' matches any character, a carat '^' matches the beginning of a line, and a dollar-sign '$' matches the end of a line. For more complicated searches, a variant of grep, EGREP, allows extended regular expressions as patterns to be searched for, e.g.

```
egrep '(any|some)(one|body)' myfile
```

This searches for all lines containing *anyone, anybody, someone*, or *somebody*.

### SPELL

SPELL takes as input an English text, and produces a list of probable misspellings.



The program collects words from a named English text file, and looks up each textword in its spelling list. If an exact match cannot be found for a textword, then an attempt is made to strip off any inflections, prefixes, and suffixes, and the putative root is looked up again; if a match still cannot be found, the textword is added to the output file of misspellings.

The user can choose between two standard wordlists, one for British English, and the other for American English. Alternatively, the user can nominate another list to be used – in practice, what usually happens is that users want to add more words to the standard list. So that the spelling-checker can look up each textword quickly, the spelling list is not stored as a straightforward textfile: the information is restructured into a *hash table*, a form that allows particular words to be found rapidly. If the user wishes to update a hashed wordlist, there is another tool, SPELLIN, which merges the list of additions into the hashed wordlist.

Of course, a 'spelling-checker' that simply checks each textword against a spelling list can never be perfect. To begin with, the system will continually throw up proper names, technical terms, etc., as misspellings. A user can compensate for this by adding these new words to the spelling list, but the spelling list will never be complete, and adding many rare words can cause other problems. For example, if a user wanted to write a paper on the problems of castrated male sheep, he or she might be tempted to add the word *wether* to the spelling list; but thereafter, misspellings of *weather* or *whether* would not be recognized!

Another problem arises with inflected and derived forms of words. To save space (and hence speed up searches), most derived and inflected forms are not stored explicitly in the spelling list: instead, if a textword is not found straightforwardly, affixes are stripped off, and an attempt is made to find the 'root'. Unfortunately, SPELL places very few restrictions on which affixes can occur with which roots, and this allows some misspellings (such as '*unboy*', '*intoing*', '*intoly*') to slip through the net. Because of this potential problem, users have the option of getting SPELL to output all words not literally in the spelling list; in this list, words which might plausibly be derived from roots in the spelling list have their putative morphological structure indicated. Also, SPELL uses a STOPLIST of common misspellings which might otherwise go unnoticed because they could plausibly be derived from words in the spelling list (e.g. '*thier*' = '*thy*' – '*y*' + '*ier*'); users can use their own versions of this too.

Note that this allowance for users to provide their own wordlists means that SPELL can be readily adapted to other tasks. For example, teaching texts are often written using a limited controlled vocabulary; if the list of words in this controlled vocabulary is used as the spelling list, a writer can readily check that his or her text conforms to the limitations.

## DICTION

DICTION performs a similar task to that of SPELL, but at the level of phrases rather than just single words – it searches a specified English text file for phrases which are often indicative of bad or wordy diction. Whereas SPELL produces a list of possible misspellings without context, DICTION prints out whole sentences, with the dubious phrase(s) highlighted by square brackets..

Of course, the choice of which phrases are to be pinpointed as objectionable is rather more subjective than the decision about what to put in the spelling list, and users are free to supply their own list of pet hates in addition to or instead of the standard file. As with SPELL, this means DICTION can be adapted to other applications; for instance, Cherry, who originally developed the DICTION program, has produced a variant SEXIST, which scours a document for potentially sexist phrases.

## EXPLAIN

DICTION merely produces a list of sentences with dubious phrases marked, leaving it up to the user to decide what corrections or changes to make to the original text. EXPLAIN is an interactive thesaurus which can be used to elicit suggested corrections for the phrases marked by DICTION. In fact, both DICTION and EXPLAIN use the same file of dubious phrases; in this file, each phrase is paired with a suggested correction, but DICTION ignores this second part to each entry. When users provide their own list of phrases to be searched for, they can provide suggested substitutions as well (although this is not essential).

## STYLE

STYLE reads an English text file, and prints out a summary of readability indices, sentence length and type, word usage, and sentence openers; in other words, STYLE attempts an analysis of the surface characteristics of the writing style of a document. The following is STYLE's analysis of this chapter:

```
readability grades:
     (Kincaid) 16.4 (auto) 18.3 (Coleman-Liau) 12.7 (Flesch) 15.2
       (35.4)
sentence info:
     no. sent 167 no. wds 5431
     av sent leng 32.5 av word leng 4.99
     no. questions 2 no. imperatives 0
     no. nonfunc wds 3420 63.0% av leng 6.32
     short sent (<28) 46% (76) long sent (>43) 22% (37)
     longest sent 127 wds at sent 164; shortest sent 3
       wds at sent 125
sentence types:
     simple 30% (50) complex 21% (35)
     compound 18% (30) compound-complex 31% (52)
word usage:
     verb types as % of total verbs
     to be 38% (199) aux 18% (96) inf 18% (94)
     passives as % of non-inf verbs 19% (83)
     types as % of total
     prep 11.0% (598) conj 4.0% (216) adv 5.5% (299)
     noun 29.3% (1591) adj 20.3% (1100) pron 3.4% (183)
     nominalizations 1% (73)
sentence beginnings:
     subject opener: noun (48) pron (6) pos (0) adj (31)
       art (24) tot 65%
     prep 16% (26) adv 10% (16)
     verb 1% (2) subconj 7% (12) conj 0% (0)
     expletives 1% (2)
```

To produce this analysis, STYLE has to analyse the text in a fairly 'intelligent' way. First, the text has to be divided up into sentences. This is not a trivial task, as

1. not all full stops mark sentence breaks (e.g. *3.14*, *Mr. E. Atwell*);
2. the text may not all be straightforward running prose – there may be 'non-sentences', like section headings, lists, or tables.

The next step is a simple surface syntactic analysis of each sentence; STYLE does this by running another program, PARTS. PARTS first assigns a set of possible parts of speech to every word in a sentence: each textword is looked up in a short wordlist of 350 common words. If it is not found there, an attempt is made to match the word against one of 51 suffixes indicative of specific word classes.

Finally, if this fails too, the textword is provisionally assigned the class UNK ('unknown'). PARTS then disambiguates words with more than one possible part of speech (e.g. *little*: adjective/adverb), and assigns a proper part of speech to words marked UNK. This is done, not by conventional parsing techniques, but by an algorithm which uses various heuristics about the expected local contexts of each part of speech. The designer herself stated that 'the method chosen for PARTS is best described as seat-of-the-pants' (Cherry 1980, p. 2), but it is surprisingly successful! Besides, the statistics output are only meant to be a rough guide, since style cannot be quantified precisely; so 100% accuracy is not required.

Having divided the text into sentences, it is fairly straightforward to calculate the various figures shown in the above example. The four readability grades are calculated using the formulae:

Kincaid: (11.8*sylperwd) + (0.39*wdpersent) – 15.59
Automated Readability Index (auto):
(4.71*letperwd) + (0.5*wdpersent) – 21.43
Coleman-Liau: (5.89*letperwd) – (0.3*sentper100wds) – 15.8
Flesch: 206.835 – (84.6*sylperwd) – (1.015*wdpersent)

The relative merits of these grades, and the details behind the other measures of style shown in the example above, are discussed in Cherry and Vesterman (1981). Users of STYLE can additionally ask for any or all of the following to be appended to the summary STYLE report:

1. a printout of the length and readability index of each individual sentence;
2. a printout of the text with each word on a separate line, with the part of speech assigned by PARTS next to it;
3. a list of all sentences that begin with an 'expletive', i.e. *There is*, *It is* (these sentence-openers tend to be overused in technical documents);
4. a list of sentences containing a passive verb;
5. a list of all sentences longer than a specified maximum;
6. a list of all sentences whose Automated Readability Index is greater than a specified limit.

Note that some of the above lists are liable to errors, since the linguistic analysis routines of STYLE and PARTS are necessarily crude and oversimplistic. Nevertheless, STYLE clearly has great potential for the English language teacher or researcher!



### The Writer's Workbench

This is a user-friendly package of English language text analysis programs , including SPELL, DICTION, SEXIST, and STYLE as well as several others. The package is not available as standard on UNIX systems; it is an 'add-on extra' for commercial word-processing applications, but obviously it would also be useful in English language teaching and research. More details of *The Writer's Workbench* are included in Cherry and Macdonald (1983); Cherry *et al.* (1983); and Macdonald *et al.* (1982).

## Programming languages

Most English language teachers and researchers who have some experience of programming have tended to stick to BASIC, probably because for them this is the most readily accessible language. Home micros come with some version of BASIC built in, and nearly all university computing services offer BASIC on their mainframes, so there may seem little point in attempting to program in another language. However, this may well be a rather short-sighted attitude for any linguist who aims to eventually progress beyond toy programs. The short-term overheads of having to learn a new programming language may be more than made up for by the time saved by programming short cuts available in the new language but not in BASIC; and besides, linguists may well relish the challenge of learning a new language! This section outlines some of the alternatives to BASIC currently available on mainframes, concentrating on what they have to offer the English language teacher and researcher. Horowitz (1984) gives a fuller overview of programming languages currently available.

Note that most of the software discussed so far is specific to UNIX, but the languages and subsystems mentioned in the rest of this chapter are available on a wide variety of machines and operating systems (including some personal computers and micros).

### Old favourites: COBOL, FORTRAN, PASCAL

The three alternatives to BASIC most widely available on mainframes are COBOL, FORTRAN, and PASCAL. COBOL (see, e.g. Parkin 1982; McCracken 1976) has sophisticated facilities to deal with the complex structured files and databases used in business environments; furthermore, the language was designed to have an 'English-like'

syntax. A program consists of a sequence of *sentences*, grouped into *paragraphs*, and each sentence must start with a *verb* (as all sentences are imperatives) and end in a full stop. FORTRAN (see, e.g. McCracken 1974) is well-suited to highly-efficient numerical computations, as required, for instance, in advanced computer graphics, or speech understanding or production systems. PASCAL (see, e.g. Cooper and Clancy 1982; Jensen and Wirth 1975) is the youngest of the three languages, and it has been adopted by most computer studies departments as the first programming language to teach students, as it encourages the design of efficient, well-structured programs.

These languages all have the advantage that there is usually plenty of available expertise to help with software design and debugging problems. Also, existing software can be incorporated into any new programs. For example, if a PASCAL or FORTRAN program involves statistical calculations, then it can use the standard subroutines or procedures provided in the system *library*, such as the NAG library of mathematical routines.

The main disadvantage of these languages from the linguist's point of view is the very poor facilities these languages have for manipulating strings or tree-structures. Although the users of COBOL, FORTRAN and PASCAL may sometimes look down on BASIC as a beginner's language, the advanced features offered by these alternatives may well prove poor compensation for the linguist.

## ADA

ADA is a comparatively new language, so not all operating systems will have an ADA compiler – yet! ADA was designed to combine the best features of a number of other languages, including COBOL, FORTRAN, and PASCAL; it is backed by the US Defence Department (and also by other NATO partners) as an eventual replacement for all other programming languages on their computers, so it is bound to catch on! As shown, e.g. in Price (1984) or Pyle (1981), ADA has better string-handling facilities than the three previous languages, and programs can access *packages* of additional functions (such as string-handling functions) which effectively extend the language in any desired direction. Another innovation is that ADA programs can include subprograms called *tasks* which will run concurrently. In fact, ADA's chief drawback is that, as it has so many advanced features that have no straightforward equivalent in BASIC, COBOL, etc., it takes a great deal longer to learn to use the language to its full



potential! This initial learning effort is clearly worthwhile when engineering complex systems involving thousands (or even millions) of instructions, but it is offputting to the casual computer user.

## LISP

LISP was the original language designed specifically for *list processing*. An *array* in BASIC or other languages is a structure containing a number of more basic elements; the number of elements must be fixed at the outset by a DIMENSION statement (or its equivalent), and all elements must be of the same type. A *list* is a much less constrained structure: the number of elements can change during the program run, and the elements can be of differing types – numbers, strings, or even other lists. This makes lists ideal for representing tree-structures, e.g.

```
(Sent (Subj the cat) (V ate) (Obj the mouse))
```

In LISP, *everything* is treated as a list; even the program itself is a list of function calls, each of which is a list of function names and arguments or parameters. The following function definition is a short example of this *functional* style of programming:

```
(define (printdouble x (print (plus x x )))
```

LISP has been described as 'the machine code of artificial intelligence'. Some beginners are put off by its strange notation (particularly the proliferation of brackets), but it is extremely versatile; Winston and Horn (1981) and Siklossy (1976) give fuller introductions aimed specifically at the non-mathematician.

## PROLOG

PROLOG is an acronym for PROgramming in LOGic. A PROLOG program is not a series of instructions to be obeyed in sequence, but rather a set of facts and inference rules stated in a notation based on Predicate Logic. As an example, the following denote the facts that Tom is the father of Harry and Harry is the father of Jim, and a rule defining what a grandfather is:

```
father(tom,harry).
father(harry,jim).
grandfather(X,Z) :- father(X,Y), father(Y,Z).
```

When using the PROLOG language, the user starts by *asserting* a number of facts and rules, which are added to a *database*. The user can then ask 'questions', which the PROLOG system will try to

answer by making inferences based on the facts and rules in the database; for example, to the question:

```
?-father(tom,harry).
```

the PROLOG system will reply 'yes'; and to the question:

```
?-grandfather(X,jim).
```

the PROLOG system will reply 'X=tom'.

PROLOG is widely used in artificial intelligence research, as it is particularly well-suited for encapsulating 'knowledge' that can be stated as a formal rule-system, such as a grammar of English. Clocksin and Mellish (1984) is the definitive introduction to PROLOG; and a version of PROLOG available under CP/M for micros, called micro-PROLOG, is described in Clark and McCabe (1984).

### POP-11

Much is made of the distinction between *interpreted* and *compiled* languages. Programs in interpreted languages such as BASIC run comparatively slowly because each BASIC instruction must be translated to machine code before it can be obeyed during a program run; however, programs can be developed *interactively* in small steps, with frequent test runs between changes. POP-11 achieves the best of both worlds with an *incremental compiler*: programs can be developed interactively, but every new procedure or function is automatically compiled as soon as its definition is complete, so that all subsequent procedure calls actually activate fast, compiled machine code. POP-11 programs are thus a series of procedure and function definitions, each one using previously-defined procedures as 'building blocks'. The language syntax itself is also incremental: not only can new procedures and functions be defined, but even new syntax words, e.g. *infix operators*. In general, the language is very powerful, with extensive facilities for string and list processing, pattern matching, database querying, etc.

A third 'incremental' feature of the language is that it is very easy for the beginner to learn the basics, and later develop his or her skills in more advanced features as and when they prove useful. POP-11 is embedded in the POPLOG teaching and research environment (see p. 182), which includes a powerful screen editor and numerous library packages. Most of the reference material on POP-11 is only available with this POPLOG system, but an overview is included in O'Shea and Eisenstadt (1984).



## Packages and subsystems

Some pieces of software available on an operating system such as VMS or UNIX cannot really be classified as general-purpose programming languages, but they are more sophisticated than most tools. For example, Heidorn *et al.* (1982) are developing an experimental text analysis system called EPISTLE which will include spelling, grammatical, and stylistic analysis all in one program; this program is so complex that currently it only runs on a large IBM mainframe. This section looks at some currently available examples of *packages* or *subsystems*.

### MINITAB

Several packages are available to help linguists and others with little computing background to use the computer without having to learn a programming language. MINITAB is a good example; it helps users to analyse statistical data, but users do not have to learn a complicated command language. In MINITAB, the user issues commands in pseudo-English, such as

```
READ THE FOLLOWING DATA INTO COLUMNS C1
AND C2
81 6
84 7
90 10
75 2
89 11
PLOT C1 AGAINST C2
```

and the MINITAB program responds by displaying graphs, plots, etc., to order. The syntax of MINITAB has very few restrictions: each command line must start with a verb like READ, PLOT, or ADD, and include the relevant column numbers C1, C2, etc.; but all other words are simply ignored, leaving users free to issue commands in almost any format they choose. Ryan, Joiner and Ryan (1982) give more details of MINITAB.

### Programming environments

Many older programming languages are catered for by an operating system simply with the addition of a *compiler*. For example, a PASCAL programmer uses the standard operating system editor to create a file containing the text of a PASCAL program, and then runs the

PASCAL compiler, which takes as input his PASCAL program file (the *source program*), and outputs a machine code equivalent program (the *object program*). BASIC and some other languages require something more, so that programs can be built up interactively. On most operating systems, typing the command BASIC will invoke a BASIC 'subsystem' – a mini-operating system including most of the file-handling and other commands found in a home micro BASIC system, such as NEW, SAVE, etc. There is a trend to develop similar *programming support environments* for other languages, including many more sophisticated aids to program development. For example, an Ada Programming Support Environment (APSE) will be available soon with facilities for keeping track of the numerous subprograms which may be written by different individuals collaborating on a large software engineering project.

### The POPLOG teaching and research environment

POPLOG is one such environment, built around the powerful programming language POP-11 (see above). It was originally developed at Sussex University, for teaching Arts undergraduates on the Cognitive Science programme, and it is particularly user-friendly for beginners. Users can ask for HELP on any command, and in addition there is an extensive TEACH facility. For example, if a student types TEACH GRAMMAR, a tutorial text on English grammar is displayed on the screen, interspersed with invitations to the student to try out some example programs which demonstrate phrase structure rules, sentence generation, parsing, etc. For the more advanced user, the POP 11 code of any of the demonstration programs can be examined using the SHOWLIB command; and detailed documentation and references can be examined using the DOC or REF commands. An overview of the POPLOG system is included in O'Shea and Eisenstadt (1984); although originally developed for teaching and research in cognitive science and artificial intelligence, the general POPLOG framework could readily be adapted for English and linguistics research and teaching.

## Conclusions

Most of the software discussed in this chapter is currently only available on university mainframe computers, but with the rapid increases in hardware technology, personal computers will soon be powerful enough to support at least some of it. In particular, UNIX (or scaled-down versions of it) is available on a growing range of personal computers and workstations, allowing linguists to access a wide range of tools and programming languages, and so to develop advanced software for English language teaching and research.