UNIVERSITY OF MURCIA

International Journal of English Studies

IJES

www.um.es/ijes

Developing WordSmith

MIKE SCOTT*
University of Liverpool

ABSTRACT

WordSmith Tools, since its launch in 1996, has had a rather unusual history and the aim of this paper is to record some of the chief influences on its development. The paper thus presents and discusses the history of WordSmith Tools and its predecessors going back to the early 1980s when processors were much slower, memory very limited and disk space expensive. It is structured around the discussion of certain key issues, principles and themes. These include: "on the fly" processing, corpus flexibility, language independence, user control aspects, the unpredictability of design needs, a move towards remote processes, modular construction principles, software production management, pattern-seeking and the development of new corpus linguistic terms.

KEYWORDS: WordSmith, on the fly, modular construction, pattern-seeking

Address for correspondence: Mike Scott. School of English, University of Liverpool, Liverpool L69 3BX, UK. Tel: + 44 151 794-2928; Fax: + 44 151 794-2739; e-mail: Mike.Scott@liv.ac.uk

I. INTRODUCTION

The aim of this paper is to present an account of decisions taken by me as the author of a software package intended for lexical analysis of texts. As such, it will give an outline explaining some aspects relevant to the context in which the work was done, and then go into the principles underlying the design. This is not an account for software engineers, though; here, the aim is to remain as non-technical as possible, to make it understandable by the ordinary user of this or other similar applications which do such lexically and textually relevant tasks as concordancing, word-listing, analysis of key words, investigating aspects of vocabulary such as keyness, consistency, dispersion and collocation, clustering, etc.

The principles I shall be examining relate mostly to the ordinary user of text corpora who would probably be teachers or students, working at home, on their own PC, without access to any kind of sophisticated laboratory.

II. BACKGROUND AND PRINCIPLES

Why did I develop WordSmith when I already had a perfectly good job as a lecturer? I was supposed to write books and articles as my research, not to make software tools. And developing software can be quite a lonely task. Some time in 2004 or 2005 I discovered that I was the sole author of published software to be returned in the official lists of research undertaken at my university; I knew this because I had to request those responsible for our portfolio database to incorporate the category software; they had already included a range of printed matter, music, film scores etc, but software was not yet in. Our university, it seemed, developed software for internal use but did not publish it formally.

Most applied linguists – by training I am an applied linguist of the language-teacher variety – do not know programming languages or particularly want to learn one; there are computational linguists who do, of course, often formally qualified in programming, but they tend to work in very different ways. Theirs tends to be team-work, in sophisticated laboratories, using relatively expensive equipment and non-domestic operating systems and programming environments. Their research aims are lofty. This is emphatically not what WordSmith is like. Essentially, the impulse to write WordSmith came instead from a desire to practise a self-taught hobby, making tools I could understand and which would relate to my own theories, work and interests. Or those of my students.

I learned what programming is about, by purchasing an early programmable calculator in the late 1970s. The program was not stored, one typed it in each time one ran it, I think, and it did something ridiculous like calculating the descent of a moon landing module, nothing linguistic, but I began to see what might be meant by programming – and its links to programmed learning theories.

Soon thereafter, my work took me back to Brazil. A major influence there came with two of the visits made by Tim Johns of the University of Birmingham, the Tim Johns who coined the phrase "data-driven learning" and was among the first to implement it. Tim knew

about the new kinds of personal computer which were just beginning to become available to the layman; boldly he suggested I get myself a "New Brain" – not a transplant but a make of computer costing about £100 at that time, and of course less sophisticated than the chips in very low cost gadgets nowadays.

This computer was a challenge. When I got to my place of work in Florianópolis, Brazil, I realised there was no software I could run on it, although I had lashed out on a fancy new printer costing at least double the price of the computer. It was either store the new computer and expensive dot matrix printer in the wardrobe, or learn to program. I probably cursed Tim as I struggled with my first proper programming language, Basic, and a few mysterious extra commands involving something called peeking and poking, which I largely ignored. One upshot of that was a word-processing program, extremely crude, but the only one I had any access to, and serviceable within the constraints of the computer (both program and saved texts highly unreliably read off cassette tape, for example!).

A few years later, Tim came back to Brazil. He had already agreed to publish MicroConcord with Oxford University Press, and as is their wont they were pressing him to deliver. Tim showed his version to me; I was impressed. He was leaving versions with various universities. Just before leaving he asked me if I would provide what we now call support to the users and I found myself trying to understand the program and rise to this challenge. To cut a long story short, some time thereafter, Tim and I agreed that we would both design and I would write a new implementation following from his original design, and after negotiations with and suggestions from Simon Murison-Bowie of OUP, who wrote an excellent manual for it, MicroConcord eventually saw the published light of day in 1993.

II.1. The "on the fly" principle

MicroConcord[1.] was a concordancer written to run on the widespread operating system of the time, DOS. The language I wrote it in was Turbo Pascal, with the time-critical sections in Assembler[2.]. It could not produce more than about 1,600 concordance lines, but it worked quickly and well, and incorporated a rudimentary system of studying collocates. It worked by reading the source texts, seeking out the search-word or phrase and storing each hit, along with its context, in the PC's RAM, its memory. This is called an "on the fly" strategy, opposed to one which pre-processes the corpus and allocates a code to each running word, so that all subsequent searches merely need to seek out a number instead of a word. Seeking a number such as 653 or 31965 is many times quicker than seeking out the search-word letter by letter.

In May 1991, at an ICAME meeting in Ilkley, I remember discussing the on-the-fly strategy used by MicroConcord with Jeremy Clear of OUP, then engaged in British National Corpus development. Cobuild, which he later worked for, and almost all other large-scale computational projects, used a pre-processing strategy, and Jeremy claimed that on-the-fly method I was using would inevitably prove too slow to be of use. By 1993, he had to

recognise that much smaller-scale project could manage well with the on-the-fly method. This method is still alive and kicking today in WordSmith.

II.2. Flexible corpus

With an on-the-fly strategy, because there is no immediate need to pre-process the corpus, one is very likely to be able to choose any reasonable text or set of texts. This means a flexible corpus. Instead of being tied to the current standard corpus, such as the BNC, one's choice of texts to work with can also be on-the-fly. This proved popular with domestic users wanting to process their own texts. It is also coherent with a wish to analyse *any* text, not just some officially-selected or officially-sanctioned ones.

II.3. Language-independent

Another principle I have endeavoured to stick to throughout is language-independence. The software does not have its own special ways of processing the text depending on the language and ought in principle to avoid this. Early in developing WordSmith, I was faced with a need to write a word-listing program. There was another concordancer on the market rivalling MicroConcord, the Longman Mini-Concordancer, which allowed its users to see a word-list of the text it processed as well as the concordance. And I had a student from Malaysia, Fadilah, who wanted to analyse the vocabulary of her textbooks; for this a wordlist would evidently be useful: she would be able to see them all in alphabetical as well as in frequency order. Accordingly, I wrote her a program to read in a text file and print out all the words with their frequencies. At that time I began to realise that deciding what counts as a "word" is not a completely straightforward matter (Scott & Tribble 2006, Chapter 2) and that counting sentences is quite tricky too.

A major turning point came when thinking about the humble form "Mrs." If the text is in English, it is very likely that capital *M* followed by lowercase *r* and *s* and a full stop then a space and then another capitalised form does not count as an end of sentence. For that matter, my own knowledge of a few European languages told me that "Mrs. Smith" in German, French, Portuguese or Spanish would certainly be a case of an English lady's title and not a sentence break. But if I built into my software an assumption that *Mrs.* plus a space plus an initial capital letter does not mean sentence-end, I'd have to build in knowledge that *UN* means *U.N.* means *United Nations*, even though in some languages the initials would be reversed, but there again would those languages use an abbreviation (*NU*) as English does, or wouldn't they? Clearly there would be no way for me to know how Lithuanian or a host of other languages would work[3].

Accordingly, at that turning-point I opted for language-independence. I do not always manage to stick to this one hundred percent, but in general the software does not

have special handling for different languages. Where this is desirable[4], my solution is always to let the user specify how.

II.4. User control of settings

This is hardly unique to my software, but is an important principle. The whole philosophy of raising to user-visibility a quite complex set of settings which can be saved and examined and altered is important. I have to recognise that showing a lot of mysterious settings in the Controller does make the program more complex, less intuitive, less user-friendly. It also makes the Help more detailed. However, the typical user I am writing for may well want to know what defaults were in effect; certainly many of those who write to me do care about the defaults and often wish to vary them. I am unapologetic about this now: if you don't want to know about the details then don't look, but if you do you should be able to find out how the program does its work.

There is a scale of adjustability which applies to tools in general. In the case of a hammer, a chisel, or a spade, there is really nothing to adjust, no moving parts, and no settings to get wrong. The spade has a certain width and length of blade which will affect how deeply it can penetrate into the earth and how much soil it can lift, and there exist other differently shaped spades, but each one has no adjustable settings. It is at the "zero adjustment" edge of the scale. At the other extreme, there is a tool like those used for tuning the electronics of car engines, the kind one glimpses attached to Formula One machines. WordSmith has always had a number of settings "under the hood" but as time has gone by and version has succeeded version, it has acquired more user adjustability of the settings and better information about them in the "Notes" tab which appears with each output type.

The level of visibility is ordered by degrees. The most noticeable of the settings comes when the user cannot proceed without choosing or defining something, as in choosing texts or in the case of computing mutual information relationships for collocates, choosing a relevant wordlist. Next come settings which are straightforwardly visible in the Controller, where there are a lot of different tab-separated settings.

Figure 1: Main settings

As is visible in Fig. 1, under General there is a "restore last work" checkbox; if this is checked and the settings are saved, then in wordsmith.ini one will see a set of saved preferences such as

[GENERAL]
keep everything simple=NO
show toolbar=NO
show statusbar=YES
restore last file=YES
…

Figure 2: ini file settings

These settings go by default, that is they remain as they were when last saved. The user isn't pestered to answer any questions before they can proceed but they may have to decide to change a setting to get the desired result, e.g. when working with a different language.

Finally, a few settings in the .ini file are not visible as choices in the general settings but are buried away only in that file. This is done where a setting needs to be completely or partly private – only those I have informed need know how to change such settings. This applies, for example, where a special version of WordSmith is needed as was the case where a research project was using WordSmith with children and limiting their choices to certain text files[5], or in the case where a version of WordSmith was in use by the Guardian newspaper for generating keywords on their text archive in automated mode.

Despite the plethora of settings, intrinsically not easy to understand and appreciate, both MicroConcord and WordSmith were originally planned to be used by students like Fadilah, students who would probably be running the software on their own PC, not in an air-conditioned University laboratory with white-coated technicians and a booking system. The idea was that although some basic training might be needed since users might not have much corpus sophistication, they would not be faced with complex command language syntax but a fairly simple set of type-in boxes and OK buttons. In the event, I discovered that students, even of the primary-school variety, had few problems. The real difficulties were much more likely to come from their teachers, who not only risked losing face, but often were too sure of what could be done and should be done and could never be done.

II.5. Unpredictability

At first, like Margaret Thatcher, I too was sure I was right. Whatever I wanted for WordSmith would be the "natural" choice – I had simply not imagined any other alternative. But then people from all sorts of countries I had never been to started to write in, right from the first publication in 1996. I get messages of all sorts, always in English, nearly always very gentle, patient and courteous. Some of them call me "Mr. Smith", not knowing – and why should they? – that there's a pattern in *silversmith, blacksmith, goldsmith* etc. which leads to *wordsmith*. Sometimes their message was simply that there was a bug affecting a certain procedure in connection with the type of analysis they were carrying out. At others

they would ask whether it was possible for the program to do something differently. This is largely why WordSmith develops as it does, organically, with a new mini-version coming out every week or so. As these tiny alterations build up, eventually there is a qualitative change and it seems appropriate to consider launching a new major version, in exactly the way one announces a change of state from child to teenager or from teenager to adult, from single to married etc with a rite of passage.

The history of the major versions is as follows: version 1.0 came out in 1996, followed in 1997 by version 2.0. A couple of years later, version 3.0 came out, in 1999. It was not until 2004 that version 4.0 could be produced. This was a version incorporating major changes, adapting to Microsoft's move to 32-bit processing, to Unicode and a series of other alterations, and involved a complete re-write. Four years later, in 2008, out came version 5.0.

The aspect of unpredictability came in with my increasing realisation that, again like Margaret Thatcher[6], I was very often wrong. For example, I had simply assumed that any wordlist would necessarily fold all cases into one, let us say upper case, until some people asked me not to. I think this parallels what happened to me as a language-teacher. When I was starting out, I was sure I knew very well what was correct and what was wrong, and I must have told large numbers of people in Brazil so quite assuredly. Later on, I went to live near the US, teaching in Mexico, and I discovered a lot of ways of saying things that contradicted my earlier certainty, especially when first encountering people who seemed to be free to use almost every noun as a verb and vice-versa.

II.6. Standalone but download-only

WordSmith and its predecessor were originally designed to work on the user's PC in stand-alone mode, not needing to relate to central resources on a remote host server. However, as time has gone by a few of the facilities have migrated to the web. Since 1996 the application has been obtained only by Internet download, not on disk. All of the applications (the tools and other utilities) are installed on the user's local hard disk, and the help file comes with the download along with a good number of support files in the basic package; but the help system itself is duplicated on the web. Other facilities such as FAQs, a blog and a forum, and a few get-started and how-to help systems are only to be found at the website, as are the update package files so that users can check they have the latest version. At the time of writing, my belief is that that most WordSmith users are likely to have always-connected broadband internet, so the probability is that more and more of the support systems will gradually move to or get duplicated on the web.

The updating policy has meant that a new version can be issued every few days. This means that WordSmith never stands still for long, but like the hair on your head grows perceptibly longer, indeed whiter or curlier or straighter too, as time goes by. Sometimes it even gets a trim or a shampoo. The effect of this is to distance such a publication quite far from the publication conditions of a journal article or a book: an article will typically never

get a second edition, no matter what; and a book is very unlikely to get more than a couple. Both of these typically last, in the sense of being still read, years or decades after publication. Software that can be re-issued so quickly, on the other hand, is like a web-page, and these are well known not to last: one might reasonably suppose that software which changes often cannot last either. Luckily for me, though, MicroConcord and WordSmith have emphatically not been ephemeral. Both are still in use; WordSmith is still growing. I think it is past its adolescence but has yet to buy a home and settle down to solid respectability, it is still experimenting with life and checking out new avenues.

III. MODULAR DESIGN

WordSmith is written using Pascal and Assembler. Assembler is a brute to work with. It is a language which essentially forces the programmer to decide which part of the computer's chip will hold each code and do very simple operations like addition or multiplication using that part of the computer's chip[7]. An analogy: suppose you had to tell a robot to open the door. A human does it without thinking, but for the robot that presumably means something like approach the door, extend hand towards door-knob, turn knob until a click is felt, pull gently, etc.

In Assembler you'd be issuing lots of commands such as increase electricity current to motor by 1 volt, increase current to motor by 1 volt, push rod B with rod A by one unit of force, etc. This language has the effect of getting things done very quickly, but is a pain to think through. Pascal, on the other hand, is a so-called high-level language. In Pascal the commands might be more like raise electricity current to motor to 250 volts, push rod B with rod A till it stops at plate X, etc. It too requires the programmer to think through the whole process of opening the door in great and usually boring detail. Each tiny step or routine has to be done in the right order for the whole thing to count as successful opening of the door. In a Pascal environment (the same applies to other languages such as C#, C++, Visual Basic and Java) it is likely that a whole series of other processes will be going on as the electric current rises to 250 volts, for example checking that no wires have shorted across. The programmer does not have to worry about these additional processes, but they will slow things down somewhat, which is why I use some sections of Assembler – for its speed. Incidentally, this is the main reason that WordSmith and MicroConcord worked only with PCs or Apples running Windows: Assembler makes assumptions about the computer's chips themselves.

The environment I use to prepare my Pascal-language programs is called Delphi. This is a so-called rapid-application development tool. What that means, for example, is that if I am designing a window with a menu and some buttons which can be pressed or clicked, I can very simply stretch any button to make it taller or wider, to change its colour or the word which appears on it. The Delphi environment already "knows" how to display the slight changes in shape or colour that make the button look as if it has been pressed. Thus when programming in Delphi it is easy to make windows bigger or smaller or add a simple

checkbox or calendar object, as a whole set of objects come with the environment. By analogy then it would be easy to change our robot's eyes from green to stripy pink, but the programmer still has to think carefully through in detail what is required for the robot to open the door.

The various WordSmith sub-programs use a modular design. Each tool is a separate program. There are sub-programs which take care of the communications between them, so that when you change a setting in the Controller the change is detected by Concord or KeyWords. The reason for the modular design is that the whole programming job is a one-man effort and I simply could not keep track of all the complex aspects of each if they all were part of the one main application.

Finally, the Help system is written using a wonderful tool called Help & Manual. This is something like a word processor which allows one at the click of a switch to generate a PDF, a Help file, or a series of linked and indexed web-pages.

IV. MANAGEMENT

Most software development is carried out by teams. That means having managers, targets, team meetings and complex rules written out in detail which specify how things will be done. A one-man effort, on the other hand, can quite easily be managed in an ad hoc way; indeed typically WordSmith is developed like that. If I decide I need to program the opening of a door as in our hypothetical robot example above, I start experimenting with routines to bring the robot near the door, to decide when it's near enough and hasn't crashed into it, etc. As I'm writing, I am still pretty unsure what will work, and I run into unexpected snags. The robot twists the door handle so far that it bends, for example. Or the robot's arm bends but the handle won't budge. These snags need to be corrected one by one. My code starts to contain a mix of sections which seem to be working well, sections I'm not sure of, sections I am not using but which must be kept because the draft idea seemed to offer promise, etc.

Not very different from the writing of an academic paper. Ideally, after it is all working well, I should go back over everything and write little notes to myself explaining what my reasoning was, and what the point of each routine really was. This is important – remember that a routine might be something akin to raise electricity current to motor to 250 volts and months later I will easily forget why the voltage has to be 250 and where it all fits into the overall process. In practice I do end up writing comments for to help my own subsequent de-bugging, but not enough of them, and usually only when I find myself months or years later wondering why the voltage needed to rise and by how much.

This sort of writing process helps me to understand why I disagree with Meyer (2002:141) who writes: “Of course, it is highly likely that the next generation of corpus linguists will have a much better background in programming. Thus, these corpus linguists will be able to use their knowledge of languages such as Perl or Visual Basic to write specific "scripts" to analyse texts, and as these scripts proliferate, they can be passed from person to person and perhaps make obsolete the need for specific text analysis programs to

be designed". It is true that scripts can be written to do a lot of jobs for analysing texts. And it is not rocket science, anyone can write them and some corpus linguists do so and find the skill extremely useful. The reason why it doesn't happen much, in my opinion, is that most corpus linguists are like most car drivers: we want to drive and we are prepared to buy the fuel and learn the highway rules and pass a standard test, but we are simply not interested in changing the tyres for bigger or thinner ones, or tweaking the engine's performance. As car drivers we keep to the roads; if we want to go across the hilltops, we tend to go on foot. We are less interested in the car than in the view.

V. PATTERN-SEEKING

What one sees leads me to the issue of patterning and pattern recognition. Tools are needed in almost every human endeavour, from making pottery to predicting the weather. Computer tools are useful because they enable certain actions to be performed easily, and this facility means that it becomes possible to do more complex jobs. It becomes possible to gain insights because when you can try an idea out quickly and easily, you can experiment, and from experimentation comes insight. Also, re-casting a set of data in a new form enables the human being to spot patterns.

This is ironic. The computer is an awful device for recognising patterns. It is good at addition, sorting, etc. It has a memory but it does not know or understand anything, and for a computer to recognise printed characters, never mind reading hand-writing, is a major accomplishment (WordSmith Help: section *Tools for pattern spotting*). Human beings are very good at noticing visual patterns, and this is why WordSmith's lists and plot displays are useful. In effect, the chief purpose of the software is to take a pre-existing shape, the text, then mix it all up and sort it all out, showing it in a quite different order. The computer does not see any patterns, but the human user does – and then gets some sort of insight.

V.1. New features, new terms

As WordSmith has developed there has been a constant need to create new terms. The very term "concordance" was initially quite problematic, since users were not used to seeing the same element repeated vertically down a page or screen, and they were not used to reading the concordance vertically. Luckily, Google and other search engines taught the world to look at displays not very different from a concordance.

Terms I have had to create include "cluster", "consistency", "standardised type-token ratio" and "key key word". For repeated strings I chose *cluster* because *phrase* might be taken to imply some sort of grammatical status[8] and because *n-gram* seemed too techie. *Consistency* was created to deal with the notion that a word might be found consistently across lots of different text-types or only within a narrow set (Paul Nation calls this "range" and has for many years offered a program with that name to investigate the phenomenon).

Within *consistency* then there was a need to distinguish between *detailed* (showing all the words of each text) and *simple consistency* (showing only how many texts each word occurred in). *Standardised type-token ratios* were needed because of the limitations of ordinary type-token ratios, whose values relate too strongly to the amount of text being processed. The term *key key word* was probably a failure. I originally intended it for words which are key in lots of texts, assuming that somehow these words would be generally key, somewhat as general nouns like *thing* and *car* and *people* are more widely applicable (and more consistent) than *pencil* and *dragster* and *taxidermist*. I imagined I might get KKWs like *police* and *wine* and *good*. Instead, I later discovered, KKWs were quite likely to be pronouns like *she*. This is because a good number of texts where the main person mentioned is a woman have *she* as a key word, and the number of such texts is typically greater than the number which has a more lexical word such as *people* or *car* as a key word.

VI. CONCLUSION

Writing software has become a long-term commitment. I have enjoyed it, but when I bought myself a "New Brain" in 1982 I never realised that I was signing up for a quarter century rollercoaster ride. I think I have been lucky to have been in there at the beginning. The rollercoaster is now ready to get off those tracks and head off across the scenery. I am looking forward to seeing where we get to next.

NOTES

[1.] It still exists and is a free download available at www.lexically.net. The corpora which came with MicroConcord, however, are not available any more, because the permissions sought at the time did not envisage this.

[2.] Turbo Pascal is a so-called high-level language which has a fairly large (a few hundred items) fixed vocabulary of formally defined terms such as `begin, end, while, for if`. Anyone who knows English can guess at what some of these instructions might mean. It is sometimes possible for a programmer to read a script in Pascal and understand what happens, like a conductor reading a musical score. Assembler has a much more restricted vocabulary, which is far more arcane. Its commands go straight to the various components on the computer's chips, much as if you had to steer a car by issuing a command like "turn the front wheels by 7 degrees left." It gets the job done very fast but I find it bends my head.

[3.] As a matter of fact I did end up learning a lot about some languages. One system I built into WordSmith 1-3 and gladly ditched for WS4 was knowledge of appropriate sort order. The correct sort order even for English is a bit tricky in a few cases such as *MacDonald* (is this put with *McFarlane* or just before *mad*?) Determining the sort order for Spanish and Danish is also slightly complex. For Hungarian it is more tricky, and may involve triplets of letters like *Mac* determining the order. When it came to WS4 I found a way of using Microsoft's routines instead.

[4.] For example in the treatment of hyphens or apostrophes, or in lemmatisation.

[5.] ESRC study (R000223900), 'An investigation into corpus-based learning about language in the primary school', 2002-4, with Paul Thompson and Alison Sealey. The project was an investigation of the potential of corpus-based teaching for children's evidence-based learning about language. Further information can be found on the project web page http://www.rdg.ac.uk/app_ling/sst.htm.

[6.] In Margaret Thatcher's case, it is far from sure that she ever realised or accepted she was wrong while in power.

[7.] The "peeking" and "poking" referred to earlier is where the programmer goes right down to the level of Assembler. What these mean is seeing what is in a certain part of the computer's memory (peeking) and replacing that with something different (poking).
[8.] Sinclair (2008: 407) points out that *phrase* if modified by *verb* or *noun* was taken over by early TG grammar but argues that the word unmodified, is "best left as a pre-theoretical term".

REFERENCES

Delphi: http://www.codegear.com/products/delphi

Help & Manual : http://www.helpandmanual.com

Meyer, C. F. (2002). *English Corpus Linguistics: an introduction.* Cambridge: Cambridge University Press.

Paul Nation's website: http://www.victoria.ac.nz/lals/staff/paul-nation/nation.aspx

Scott, M. & Tribble, C. (2006). *Textual Patterns: keyword and corpus analysis in language education*. Amsterdam: Benjamins.

Sinclair, J. McH. (2008). The phrase, the whole phrase, and nothing but the phrase. In S. Granger & F. Meunier, *Phraseology: an interdisciplinary perspective*. Amsterdam: Benjamins. 407-410.

Tim Johns' website: http://www.eisu2.bham.ac.uk/johnstf/index.html

