A word boundary algorithm for text processing

H. W. Thimbleby

Computer Systems Laboratory, Queen Mary College, University of London, London E1 4NS, UK

Many text display editors include 'word' function keys. The user effectiveness of such functions depends crucially on a well thought out idea of what 'words' are. This paper presents a carefully thought out scheme for word functions and gives the relevant algorithms for them in Pascal. The approach used here has been used with success for well over two years after examining and rejecting many alternatives.

It is beyond the scope of this paper to attempt a psychological assertion on the nature of words or word formation.

(Received September 1979)

Introduction

There seem to be as many sorts of text editor as there are establishments and machines capable of developing them; descriptions of them in the literature tend to present them as fait accompli. The purpose of this paper is to describe and justify a technique and its associated algorithm that may be used in any text editor. The algorithm can be used effectively in input processors or editors for program or natural language text.

First, word boundary algorithms are put into perspective by describing how they may be used and then the common alternatives summarised. This is not a review paper of editors; references to specific editors are largely for didactic purposes only. With this as background the general criteria for word boundary algorithms are discussed; then the preferred algorithm, which is finally presented concisely in Pascal, is developed. Two complete systems each using the algorithm are described by Thimbleby (1980) and Bornat (1979).

Background

The algorithm described here has been in constant use, in several systems, for over two years, after nearly a year of development. Over this period, and before, many users of varying typing skill (a significant proportion of whom could be described as very experienced with the system described here and as skilled typists, but not as touch typists) were consulted and observed, and our understanding of the problem refined. The emphasis of this paper is the design philosophy rather than the design strategy. So we are confident that the algorithm presented here is well worth consideration; the first section of this paper gives a brief description of how it may be used.

The algorithm is designed to provide for four immediately acting word functions: (move to next word), (move to last word), (delete to next word), and (delete back to last word). There are other less effective sets of functions, for example some editors only provide a single word delete function—which will frequently delete more than the user wishes. It appears crucial that the command set of an interactive system is coherent, though it is beyond the scope of this paper to describe complete systems in which the four functions may be embedded most effectively.

It must be stressed that the multi-access computer system on which our algorithm was developed could give essentially immediate feedback on a per keystroke basis, and it is very likely that when this is not possible alternative techniques should be used. We use terminals with a keypad for cursor positioning: left, right, up, down, word left, and word right keys is a minimal set. All these editing functions, and others (Coppen and Salama, 1979), can also be activated with single control keystrokes (such as 'control + a') from the terminal's standard QWERTY keyboard; this has the significant advantage that users need not reposition their hands to change between editing functions and direct typing—and the functions are available on all standard terminals. Pearson (1978) confirms that users view this as an advantage.

Notation and terminology

Display editors use a cursor which indicates to the user where all (local) text alterations are to occur. It is where any typing is echoed and is usually clearly visible—perhaps it blinks. This paper uses the convention x,y to indicate a cursor position, or a print position for a hardcopy terminal, between x and y; on most terminals the cursor, or print position, will be represented under the y. It is much easier to describe precisely what editing functions do (such as insert a character) if the cursor is considered as being between two characters, however it may be physically represented.

Throughout this paper the user's idea of a word and the implementer's are compared: the latter is called an atom only for the purposes of clarity and I have not used this term where the context makes the meaning clear. The effectiveness of a user interface is measured by the extent to which the user can forget that there is an interface (Hiltz and Turoff, 1978); the purpose of this paper is to define an atom that is sufficiently representative of a word that for users atoms are words. The success of the algorithm here is that indeed nobody (else) talks about it.

Control functions are represented by (function), and on the whole they can be implemented through single keys that the user can press, though they may be implemented by using control shifts on alphabetic keys (e.g. 'control + a'). Sometimes, due to system or terminal restrictions, control functions may require several key presses and delimiters, though this is undesirable.

How to use a word boundary algorithm

It is perhaps useful to distinguish certain classes of editing:

- spelling correction (the user edits single letters);
- word composition (the user manipulates the text as words, experimenting with different phrasings etc.);
- copy typing or copy editing [the user is primarily typing new material (possibly copy) with occasional corrections];
- script editing (the user follows written corrections to existing text); and
- 'cut and paste' (the user manipulates large textual sequences).

Word boundary functions are applicable mainly for the middle three classes and a viable text editor or word processor would certainly require facilities for spelling correction also. There can be seen to be a range of possible facilities to cover different
degrees in the sophistication of the users' tasks; what is helpful to users will also depend on their proficiency. Historically this has caused problems in that computer programmers, although often poor typists, are able to operate very complex editors to their satisfaction and this has sometimes led to confusion over what facilities are required in editors for lay users. This is a standard human factors issue.

Simple text input
Like many command processors, ours provided simple facilities for editing lines of characters as they were typed. Thus the last character typed can be deleted using \texttt{rubout}, and the whole line deleted using another control character. We added a further function \texttt{delete last word}, the effect of this was to delete the last atom that the user typed; elsewhere this has been called resynchronisation (Gilb and Weinberg, 1977).

Editing
An editor \texttt{em} (Coulouris, 1977) was then developed to take this idea further with a command called 'open mode'; similar sets of commands are provided in SOS (Wether and Savitzky, 1971) as 'intraline edit mode' and TECO (Murphy, 1966) in 'display mode'. Open mode allowed text to be inserted into a file and allowed the user to use a variety of \texttt{immediately} acting editing functions (unlike SOS or TECO whose commands generally require delimiters and typing search parameters) on a hardcopy terminal. An example dialogue in open mode is given is Fig. 1; the left hand column shows what the user is typing, the middle column shows how this may be represented on a hardcopy terminal and the righthand column shows how it is represented on a video display terminal. In the figure, the user types 'word editing now' initially and then rapidly edits this to 'word editing now'. Notice how, on the hardcopy terminal, a record of all the editing builds up—with the convention here when text is echoed between square brackets it indicates it has been deleted, and the \ at the end of a line merely indicates that the cursor has been moved backwards in a line without any deletions occurring. On the display terminal, there is no record of the editing; the editing process is clearer as all the text on the line can be seen directly, and — most importantly—because there is no room to clutter up screen space, the text on the line being edited does not move from adjacent text in the original. However, Fig. 1 shows just two of the many ways of editing text in this fashion.

Wraparound
When a user types a stream of text it is useful, especially when they are copy typing, if the system automatically 'wraps around' text as it reaches the end of lines. This requires the system to insert a new line at an appropriate position in the line, and this position can be found easily and correctly using a word boundary algorithm (though choosing the right word boundary still represents a minor problem).

Providing granularity for continuous pointing devices
Continuous pointing devices, like joysticks, can be quite difficult to align at small pieces of text. If a word boundary algorithm is used to produce a gravity field (i.e. making word boundaries easier to locate than other positions) users will find it much easier to move the cursor to these positions, and they can then develop faster editing habits—always making the same sort of changes in the same sort of way (as will be seen below)—as they improve their editing skills.

Alternative techniques for cursor positioning
Word motion functions are not the only way of moving the cursor. This section compares word functions with other major methods.

Step keys
Step keys move the cursor one character horizontally, or one line vertically, at a time. Horizontal motion very often becomes tedious when it is only possible to move one column per keystroke, and step keys often have a 'fast' mode. This is usually either a 'fast step' key which moves perhaps five columns at a time or a repeat facility (which might repeat the last keystroke at a higher than normal typing rate). Repeat keys are awkward to control on a slow system. Alternatively, the step multiplying factor can be provided by an explicit typed numerical parameter. Moving \texttt{five} columns or lines at a time, when insensitive to the text moved over and the absolute position of the final column, is a balanced value for a fixed multiplying factor. Tab functions are used to move the cursor to one of a set of (possibly interactively) predefined columns, and are extremely useful for laying out tables. The commonest general tab stops are set at no more than every eighth column.

Syntactic positioning
If the editor has some knowledge of the syntax of the language being edited then functions can be provided to speed up cursor motion over language structures. With a programming language this can be a great aid to the user: for instance the user can rapidly scan a program text at a particular level of nesting. In the editor EMACS (Stallman, 1979) there are two cursors: one denotes the editing position and the other indicates a matching lexeme, which helps in balancing brackets, and delimiting subexpressions etc.

A specific case of syntactic positioning is textual positioning; step key functions are augmented by functions which move to the delimiters of typographical features such as words, phrases, lines, sentences and paragraphs. This paper develops an algorithm as a basis for word functions; if functions are provided for larger textual objects they must cope with variations of natural language syntax in a sufficiently predictable fashion. (This problem could be overcome by providing functions to indicate syntactic delimiters, e.g. \texttt{Mark end of sentence}, but this will almost double the number of special purpose keys.) If the editor supporting these functions is not always used with natural language texts, then highly specific functions may be of less use. Unfortunately, this can easily lead to a plethora of functions if one function is provided for each feature, and some of them may never be used sufficiently often for the user to feel safe enough with them to risk operations, which are at best difficult to revoke, such as deletions. (This suggests that a \texttt{Delete command} meta command would be useful.)

Field positioning
To find the end of a sentence is actually quite difficult for both

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
  
\textbf{Typing} & \textbf{Hardcopy} & \textbf{Video} \\
\hline
\texttt{word editing now} & \texttt{word editing now} & \texttt{word editing now} \\
\texttt{move word left} & \texttt{word editing now} & \texttt{word editing now} \\
\texttt{delete word left} & \texttt{word editing now} & \texttt{word editing | now} \\
\texttt{editing} & \texttt{word editing | now} & \texttt{word editing | now} \\
\texttt{move word right} & \texttt{word editing now} & \texttt{word editing | now} \\
\hline
\end{tabular}
\caption{Cumulative terminal record}
\end{table}

250 THE COMPUTER JOURNAL, VOL. 24, NO. 3, 1981
© Heyden & Son Ltd, 1981

Downloaded from http://comjnl.oxfordjournals.org/ at Swansea University on February 3, 2014
human user and computer system: the user has to read text in detail and the system has to cope with a great variety of valid sentence endings and exceptions (e.g. 'I know Lt.-Col. Pewitt.' is one, not several sentences) and so it is usually more appropriate to have functions which move the cursor to structural 'landmarks' that can be seen without detailed reading. When such landmarks are predefined, either by graphical representations (e.g. lines or colours) or by textual delimiters, the positioning functions effectively move by fields. Typical field functions are: \langle move to next field \rangle, \langle move to previous field \rangle and \langle clear current field \rangle.

Search positioning
Editors like SOS provide a multi-key function to move the cursor either left or right up to a specified character. Thus \([-\,n]S\langle ch \rangle \) moves the cursor from the current position to the \(n\)th (default: first) occurrence of \(\langle ch \rangle \), and \([-\,n]K \langle ch \rangle \) deletes up to the \(n\)th occurrence of \(\langle ch \rangle \) (items between square brackets are optional). The two main disadvantages of this method are that (a) it is not immediate and (b) the user must read the text carefully and count, especially if the cursor is to be moved a fair distance. But its advantage is that the rest of the editor could use a similar command (there are commands in SOS to take multi-character search parameters) for all forms of string search and deletion, which would make the facilities of the editor easier to learn.

It should be noted that SOS-type commands are logically, but with ordinary terminals not physically, in a different character set than ordinary text: the letters S, K and so on represent characters in a non-document character set when in a command mode. This multiple use of the same keys for different functions will lead to user problems (Thimbleby, 1979).

Continuous pointing devices
A mouse, joystick, lightpen, or even a finger or eye can be used to point to positions on a display screen. The word motion algorithm described in this paper can be applied, as mentioned earlier, to continuous motion pointers to make positioning more repeatable.

Card et al. (1978) have shown that the mouse is the fastest device, of the ones they tested, for moving the cursor to a specific position and that the speed of using word motion functions improves more with learning than with the other methods. This may explain people's reluctance to try, and their irritation when trying, alternative forms of word motion functions after acquiring significant (and often hard-won) skill with one form. (Card et al. used four fast subjects, so their results may not be typical. Furthermore the step and text keys used were separate from the main keyboard, and their data include hand repositioning times.)

Highlighting
Highlighting an area of text means making it stand out (e.g. putting it into an alternative font—capitals, bold, inverse video, italic, another colour etc.). The highlighted area of text may then be treated as a single entity, for naming, copying, deleting or moving etc. But obviously to highlight an area either requires some way of specifying the boundaries of the area interactively (i.e. by using one of the techniques of this section) or that certain forms of text (such as paragraphs or fields in a form) are always highlighted (i.e. in fields). A typical interaction may be \langle start highlight \rangle \langle move cursor \rangle \langle end highlight \rangle ; the \langle move cursor \rangle would then 'paint' the highlighting.

Luther et al. (1977) describe an editor where text to be deleted is first replaced with special characters to indicate that that text is about to be deleted and when the user requests the display screen to be reformatted all such highlighted text is deleted.

What is needed in a word boundary algorithm?
The purpose of word boundary algorithms is to find the boundaries of the atom nearest to the cursor. An editor, for example, can provide user control functions to move the cursor to these boundary positions or delete text up to them. In simple cases it is obvious how an atom should be delimited: the first letter of a sequence of contiguous letters is the beginning of an atom, and its last letter is its end. Unfortunately this simplistic view of a word leads to trouble for the user, as will be seen. So, first, what are the criteria that must be met by a word boundary algorithm?

Predictable
It is crucial that word motion keys do not move the cursor in unexpected ways, this is especially true when word motion functions are augmented with equivalent deletion functions. It is obviously a disaster if more text is deleted than the user anticipates, but it is also a disaster if word motion moves further than the user anticipates because the user must then work out how to backtrack the cursor, which causes unnecessary delay. For an experienced typist, copying text and without visual feedback, Long (1976a) and Rabbitt (1978) show that the majority of all typing errors are corrected within three keystrokes of their commission or not at all. So it is better if atoms are sometimes too small rather than occasionally too large.

Clear
Cursor motion should be so obvious that the effect of the word function keys can be understood with no more said than 'this key moves one word right, this one left, and this deletes a word to the right and this to the left'. What word motion does should be both consistent and memorable. It should therefore be compatible (Murrell, 1976). Word motion and deletion functions are not merely to speed up editing by having the cursor move four times as fast, but the word (at some level) is a concept that already exists in the user's mind and being able to move the cursor to an atom directly and precisely— which mouse, joystick etc. alone cannot do—is a significant convenience.

Intuitive
Atom boundaries should therefore coincide (as far as possible) with the boundaries of words. If the cursor is at the beginning or end of a word it interferes with reading less than if it is somewhere within the word. It is also much easier to anticipate where the cursor will move to since the beginning and end of a word are far easier to locate than some arbitrary position within them, which might only be found by scanning individual characters within a word or by estimating a halfway position.

If the user wishes to move the cursor to the middle of a word it is easier for them to do this by explicitly moving the cursor in from one end of the word one character at a time, rather than moving it from a central position within the word. This is partly (but not entirely) because users tend to plan a sequence of actions ahead (on slow systems, users may even type ahead of system feedback), into 'response units' (Rabbitt, 1978; Shaffer, 1973), and this obviously becomes harder the more detailed the thought that has to go into it. In short, atom boundaries are more predictable if they do not occur within words.

Repeatable
Whatever algorithm is chosen, it must be repeatable. The algorithm is used to find one of a set of positions on a line of
text closest to the last cursor position. These positions must depend only on the text of the line, and not on the direction of movement towards them: moving left one atom at a time should position the cursor successively on the positions it would have landed on through rightwards motion, and vice versa. If the user overshoots a target position by using a word motion function too much, they can return to the target position simply by reversing the direction of cursor motion without having to correct the final position with single character cursor motion.

Habit forming
Also, the algorithm should encourage confidence in using similar sequences of keys to do similar edits, more or less regardless of the text. If a word is inserted between two others, both the letters of the word and one or two spaces (and perhaps punctuation) must be typed (depending on the insertion convention); it would be nice if, so far as possible, the same sequence could be learnt and used generally. In fact, users do tend to adopt a simple 'blind' sequence for such alterations and this sometimes causes them to type extraneous characters, such as too many spaces. By mistake: it would be rather nice if these extraneous characters were, as often as possible, made innocuous.

General
Finally, the algorithm for locating word boundaries should be general and not specific to certain forms of text or programming language, and work without making too many assumptions.

System requirements
The importance of immediate feedback to the use of these functions cannot be over emphasised (though fast response times are never a disadvantage, sometimes they are not strictly necessary). There is, of course, now considerable evidence that interactive systems must respond to user requests as fast as possible and preferably be perceived as instantaneously—especially for initial skill acquisition [see Klemmer (1971) for a review of typing research]; user effectiveness is inversely correlated with feedback time. Under conditions of delayed feedback (Long, 1975) there are significantly more errors (in a normal typing environment) but the difference is reflected in the percentage of errors subjects fail to correct. A delay in visual feedback disrupts error checking, although skilled typists adapt to delays after several minutes of normal typing. However, since under normal typing conditions skilled users probably adapt to the delay rather than by supplying substitute feedback (Long, 1976b—the delays were around 300 ms) it is not clear whether this observation can be generalised to an editing situation where feedback is of a more complex nature. Typists are able to type and check their accuracy concurrently on a per character basis, even at very high rates (e.g. 172 ms or faster—Rabbitt, 1978); in other words, typists are potentially able to correct many of their errors using a purely motor response. This can, of course, be done without identifying the exact nature of the error.† Such rapid error correction can only be supported using a scheme such as the one suggested here, which actually allows them to.
In other words, word based functions, as used here, support close coupled perceptual–motor control (if feedback is maintained as per character and is rapid enough) which other editing techniques do not. Since word function operations are parameterless, their effect can be, and should be, synchronised with user typing.

Unfortunately our operating system does not permit time measurements to be made more precisely than to within 1 s, and so it is hard to quantitatively evaluate functions of this sort. [Current work is involved with implementing modifications to permit adequate measurements to support (or refute) assertions made in this paper.] Card et al. (1980) give a keystroke level model for user performance but they do not reveal sufficient experimental details to make useful numerical comparisons. (Furthermore, their model assumes error free typing and one of the advantages of word functions is the ability to correct just typed words.)

Diagnostics
If the user issues an editing command which cannot be obeyed for some reason, such as (move left) at the left hand side of a line, then the system should respond with an audible diagnostic (or indeed, any stimulus in a non-visual mode), such as a bell. This will enable the user to type without paying undivided attention to the system's visual feedback; without noises it would be extremely risky attempting to copy type or copy edit, or make any alterations to text whilst looking at a document. Again, this feedback should be effectively synchronised with user typing.

Preferred word boundaries
An algorithm is now developed which will satisfy the above criteria.

Text consists of letters, digits, punctuation and blanks. Tab characters look like so many blanks, and it is my opinion that this is how they should be treated; tabs may still be typed with a single keystroke, with the effect of generating (or moving over) so many blanks. (Otherwise tabs are merely an inefficient data compression mechanism, and this aspect of tabs should be transparent to the user, if not altogether superseded by a more effective method.)

For the purposes of clarity, it will be temporarily assumed that text consists of letters and spaces alone; later this assumption will be revised.

So that atom boundaries coincide with word boundaries the space delimiting a word has to be included in either the atom to its right or left, or spaces have to be delimited on each side (Fig. 2). What happens at the start and finish of text (ellipses in the figure) will be looked at shortly. The third alternative can be dismissed immediately: cursor motion is twice as slow and it introduces two sorts of 'word' (text and blanks) which could easily let the cursor get out of synchronisation with the user's model, so the user has to be very careful about alteration strategies. Also, after a sequence of word motion commands the cursor will be either to the left or right of a space; methods to insert, replace or delete words will depend on whether the cursor is positioned at the start or end of a word. Deleting an atom will either concatenate two words or leave a double space. But by using either of the first two alternatives, if an atom is deleted using a (delete word) function (which will use the same definition of atom boundaries) both word and one space are deleted, so a double space is not left over between two words after a deletion.

Now, it happens that when the user types in a word after moving to a word boundary, the user will mimic the algorithmic

\[ \text{[space included at [right] of [letters \ldots]} \]
\[ \text{space} \text{included at [left] of [letters \ldots]} \]
\[ \text{spaces} \text{are delimited on [each] side} \]

\[ \text{Fig. 2} \]

252 THE COMPUTER JOURNAL, VOL. 24, NO. 3, 1981
© Heyden & Son Ltd, 1981
idea of a word. This happens because if the word boundary at which the user starts editing is to the right of a space, then the user must terminate the inserted word with a space. On the other hand, if the word boundary is to the left of a space, then the user will first type a space and then the word. Now, a superfluous space typed at the right hand end of a line can be ignored but a space typed at the left hand side of a line indents the whole line (when the space is inserted, or deletes some character if in replace/overwrite mode). To avoid this accidental indentation, a word boundary should be to the right of any spaces preceding a word. Hence we have chosen the first alternative in Fig. 2. This is also compatible with the 'natural' concept of a word: atom boundaries coincide with the start of words.

Our experience with this arrangement is that skilled users, as they type, tend to correct incorrectly typed words not on a character basis, except for correcting the last couple of letters of a word, but by deleting the whole word and retyping it entirely. This is probably because typing a word is done to a rhythm and can be done without the detailed feedback required for single character alterations. And with the word boundary to the immediate left of a word, as suggested, the preceding space need not be retyped when a word is deleted and itself retyped.

This means that right word motion leaves the cursor one space (in single spaced text, but see Fig. 9 later) to the right of preceding atoms. To be consistent this must be true also for the last atom on each line (Fig. 3). Even here, one space to the right of the last atom on a line, the user may insert a word using the same sequence as anywhere else in the line—namely \texttt{<word> <space>}. Indeed, the ease with which a word may be appended to a line is a further justification for this particular word boundary position. And not just to remain consistent but by following the preceding argument through again, there must be a similar word boundary to the right of the first space in a gap in the middle of a line (Fig. 4).

If the text on the line is indented, then there is a word boundary at the first column but not one space beyond it: this is because there is no word that can precede the first column of a line (Fig. 5).

Most text, of course, includes punctuation, and the definition of word boundaries must take this into account. It turns out that the only way to do this consistently with isolated punctuation characters is as in Fig. 6. How to treat sequences of punctuation, such as \texttt{...}, will be discussed shortly. The first alternative, namely that punctuation itself makes an atom, is preferable: on occasions it may split lines up too much but, as was pointed out earlier, this is definitely preferable to an alternative which occasionally has word boundaries further apart than expected. So, to the right of punctuation, the position of a word boundary is unchanged—it is immediately after a space, if there is one in the text. To the left of punctuation there is a further word boundary: having a word boundary at this position enables the user to delete the real word and punctuation separately (as they are in different atoms) and to move just to the left of punctuation with word motion functions so as to insert new text there, if desired, far more easily than through using the second alternative in Fig. 6.

If there are functions for moving or deleting the word to the right of cursor (which we think are necessary) the boundaries as described so far have an irritating problem. Consider Fig. 7. The cursor is positioned just to the right of the colon after \texttt{label} and the user types the function \texttt{<delete word to right>}. What the user expects is for all spaces between the colon and instruction to be deleted—but this is not what happens at all. The word boundary to the right of the cursor is only one space away, so the \texttt{<delete word to right>} will only remove one space, and the line will close up without the cursor moving (again, assuming immediate feedback is practical). The cursor will still be to the right of the colon and one space to the left of the nearest word boundary and so further \texttt{<delete word to right>} will only remove one space at a time. This case (which occurs rarely, but is quite unexpected when it does), is best handled by an extension: namely, for \texttt{right motion (and deletion) only}, if the cursor is under a space then the next word boundary (to the right) is under the nearest non-space to the right. This extension is quite consistent with the effect of word motion and deletion in less exceptional cases, and it is also safe if overshoot occurs.

Finally, there is a word boundary at the right hand margin of a line (Fig. 8). This provides an expected word boundary, mirroring the word boundary at the left hand margin of each line. An important aspect of the word motion functions is that they are symmetric: the right hand margin is usually easy to move the cursor to (perhaps there is even a direct \texttt{move cursor to far right} function) and from here \texttt{move word left} will work naturally—so it is essential that \texttt{move word right} goes back to the right hand margin position. (The rightmost boundary also enables a simple mechanism to move the cursor from the end of text on one line to the end of text on another.)

Note. If there is some form of wraparound (sometimes called 'word-wrap') then there will be no boundary at the far right of a line, as in Fig. 8, but word functions should treat the new line in exactly the same way as the wraparound for ordinary text. Thus, as wraparound occurs with no special characters typed by the user, a physical new line sequence is \texttt{ignored} by word functions. Probably this will be done by treating the notional new line as a space, unless the wrapped-around word is hyphenated. (A bad example of not ignoring the internal new line sequence was a display editor where, at the beginning of a line, the \texttt{<left one character>} first moved up one line and then typed a second time it moved to the far right of that line—as the cursor was moved successively over an invisible 'linefeed' and 'carriage return'.)

But what is a word?

It has been assumed that atoms were made up from letters, but this is not by any means always the case. In our experience of text and program editing (limited to using programming languages with no stopping conventions or alternative character sets) an atom is best thought of as being any combination of letters and digits. When letters and digits are run into each other, e.g. \texttt{37p} or \texttt{BAFE}, the alphanumeric string is

\begin{verbatim}
|word | boundaries | include | space | at | end | of | line |
\end{verbatim}

\begin{verbatim}
|word | boundary | in | a | gap |
\end{verbatim}

\begin{verbatim}
|first | words |
\end{verbatim}

\begin{verbatim}
|words, | punctuation, | and | more, | |
\end{verbatim}

\begin{verbatim}
|words, | punctuation, | and | more. |
\end{verbatim}

conceptually one ‘natural’ word—hence in the suggested atom boundaries do not occur between digits and letters. In programming languages, identifiers and constants are often made up from combinations of letters and digits—frequently the user would find it surprising if such strings were broken up into separate atoms. If this argument seems unconvincing, consider that the alternative—splitting these words—requires determining a predictable position within them, using some generally applicable algorithm.

All other characters (excluding spaces and also tabs (ASCII ‘control+I’), which should be treated as sequences of so many spaces for the purposes of the displayed text, as mentioned earlier) should be treated as punctuation. This means that an ellipsis or any other recognisable sequence of punctuation (such as ‘. . .’) is not elided into a single atom. It might also be thought that punctuation sequences that can terminate sentences (such as ‘. . . ? and ‘.’) etc.) should be followed by two spaces (as natural language text looks better with two spaces between sentences if words are single spaced). If possible, the double space should be provided by a text filling system (e.g. a text formatter) rather than a word boundary algorithm, as catering for these exceptions will make the word boundary algorithm less uniform and harder to trust. The most significant disadvantage of treating certain punctuation sequences (such as programming language lexemes) as ‘words’ is most apparent after word deletions when, although the user’s expectation of the result may be correct, never the ‘right’ amount is deleted—usually too much, even if there is some limit on atom length. When atoms may be punctuation sequences, the cursor almost always moves further than expected. Furthermore the verbal bias of human memory makes retying an accidentally deleted punctuation sequence (rather than a letter sequence) harder. Not all sequences of punctuation could reasonably make an atom (think of ‘.’)))}), for instance, and we have found it better to make no special cases even for when editing program texts.

We have used the same word motion functions for all our editing purposes, both for natural language documents and for programs in several programming languages. In our experience, it is far better to use only a single definition of word boundaries than one adjusted for each form of text, mainly because users do learn certain sequences for editing at a motor level and thus even minor variations in how an atom is defined causes considerable frustration: the user cannot move between editing text, program or data smoothly unless the same algorithm is used in every context. However, this is not to deny the usefulness of syntactic positioning (where the motion function moves the cursor multiples of lexemes). Hence the argument may not generalise to special purpose systems.

Summary of algorithm
The preceding paragraphs were a bit lengthy (even wordy) in order to rationalise our choice of word boundary algorithm. This section summarises them. Fig. 9 shows all the possible combinations of characters on each side of a word boundary.

The following algorithm, written in Pascal, declares two functions, leftword and rightword which take a character array and an index (representing the cursor position) into it. Both functions return a new cursor position, and may call the procedure ringbell if it is not possible to move the cursor. The cursor position returned is an index into the character array, pointing at the character position immediately to the right of the word boundary. The constants left and right are the indices of the left and right hand margins in the character array. (Most terminals have a problem displaying the cursor for the rightmost column; an unambiguous solution is to keep line[right] permanently blank (if right is the rightmost column). Further discussion of this detail is beyond the scope of this paper.)

Pascal functions to locate word boundaries

const BLANK = ' '; left = 1; right = 80;
type cursor = left . . right; line = array[cursor] of char;

function alphanumeric(c:char):boolean;
begin alphanumeric := c in ['a' . 'z', 'A' . 'Z', '0' . '9'] end;

function atwordlb(l:line; p:cursor): boolean;
begin if (p > left) and (p < right) then
begin if (l[p] = BLANK then
if p > left + 1 then
atwordlb := ((p-1) = BLANK) and
((p-2) < > BLANK)
else
atwordlb := false
else
if alphanumeric(l[p]) then
atwordlb := not alphanumeric(l[p-1])
else
atwordlb := true
end;
else
atwordlb := true
end;

function leftword(l:line; p:cursor): cursor;
begin if p > left then
repeat p := p - 1 until atwordlb(l,p)
else ringbell;
leftword := p
end;

function rightword(l:line; p: cursor): cursor;
begin if p < right then
begin if p+1 < right then
if (l[p] = BLANK) and (l[p+1] = BLANK) then
p := p + 1;
repeat p := p + 1 until atwordlb(l,p)
end else ringbell;
rightword := p
end;

For example, to find the column of a word boundary to the left of the current position, call leftword and this returns the index of the new cursor position. Note that if the returned cursor position is unchanged leftword calls a procedure that rings a bell (most terminals have one), and so the user is notified immediately that the cursor has not moved—even if he or she is not constantly watching the screen. This will happen if the cursor is already hard up in the left hand margin in this case.

<table>
<thead>
<tr>
<th>Cursor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(right margin)</td>
<td>(left margin)</td>
</tr>
<tr>
<td>(non alphanumeric)</td>
<td>(punctuation)</td>
</tr>
<tr>
<td>(not left margin and not space)</td>
<td>(alpha)</td>
</tr>
<tr>
<td>except for right motion/deletion from a space:</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 9 Word boundaries explained, for n-spaced text

Conclusion
Word motion and deletion functions using the algorithm of this paper have been in use for over two years, and previously many other less effective algorithms were also tried and evaluated. Experience gained with them was the basis for the section justifying the chosen algorithm. The algorithms have been embedded in a display editor and a command processor/editor and are currently being written into the operating system to be available for the user interface of every program.

Our experience suggests that the final algorithm given here is the best general purpose word boundary algorithm.

Acknowledgements
This paper would not have been possible without the help of many users, and the experience gained with Richard Bornat and his excellent display editor. Andy Whitefield and John Long have made many valuable suggestions. The referee made a number of helpful suggestions.

References

Books reviewed in this issue

| Cobol Programming | 221 | IJP: Intermediate Language for Pictures | 257 |
| Data Communications: An Introduction to Concepts and Design | 199 | Integrated Office Systems | 234 |
| Digital Hardware Design | 248 | Selecting EDP Audit Areas | 199 |
| Elementary Computer Programming in Fortran IV | 234 | Systems Analysis | 209 |
| Entity – Relationship Approach to Systems Analysis and Design | 248 | Systems Management | 199 |

© Heyden & Son Ltd, 1981

THE COMPUTER JOURNAL. VOL. 24, NO. 3, 1981 255