Chapter 8 ALGORITHM

8.1 Solution

From Chapters 5, 6 and 7 it is clear that we can formulate a set of rules that will provide the break points between syllables. Even where further research is required, later refinements to the basic rules can be made if we choose a simple encoding method for the rules and generalised searching of them.

For this purpose we envisage searching the word from right to left using a set of tables containing the rules to be searched.

8.2 Outline algorithm

The system uses logical rules based on the morphology of Afrikaans working on the whole word and searching from right to left for a suitable break point. As soon as an acceptable break is found processing stops, i.e. no alternative breaks are looked for. Apart from the word itself, the main parameter required is a pointer to the last character which can be fitted on the line of text, taking into account the width of the hyphen. This is called the **optimal point**.

The hyphenation algorithm is written as a computer procedure which runs in two stages. The initial call ensures that the Prefix, Suffix and Consonant/Vowel tables are loaded into the program from external storage. Subsequent calls apply the actual hyphenation logic. The procedure extracts all the alphabetic characters and accents from the supplied word and only works on those, i.e. numerals, formatting and punctuation are ignored. Words of 3 or fewer characters are rejected. If a normal hyphen put in by the typist occurs in the word, then the word is rejected and the user must make a manual decision.

The procedure next searches for possible prefixes using one of the **prefix tables** of which there is one for each letter of the alphabet. If one of the table entries matches the beginning of the word, breaking is *attempted*. This means testing to see if the break is to the left of the Optimal Point. If the attempted break is to the right of the Optimal Point the procedure can exit without finding a hyphenation point. However,

if the break is to the left, the word may be broken at that point, providing there are no further acceptable hyphenations closer to the Optimal Point. In this case the operation is repeated. If no prefix was found, the procedure continues with a search for suffixes.

The suffixes are classified by the final letter, so the appropriate **suffix table** is accessed by determining the last character of the word. If a matching suffix is found during the search, a code is supplied with it which indicates where the break is to be attempted. Since multiple suffixes may occur, the return code is used to test whether the tables are to be re-entered for another attempt. In the latter case, the suffix already found is ignored when trying the tables again. Again, if the attempted break is to the right of the Optimal Point or if no suffix was found, the procedure continues with a search for logical breaks. Otherwise an acceptable hyphenation point has been located and the procedure exits.

The letter pattern at the Optimal Point is established consisting of strings of Vowels and Consonants. The patterns relevant to us are those consonant clusters surrounded by vowels. Thus VCV, VCCV, VC..CV provide the correct syllable definition. With each of these patterns there is an associated table which is arrayed to find a corresponding match at the letter level e.g. SPR is an unbreakable cluster. As a result of this search, the procedure may find a break, when it is tested as usual against the Optimal Point. If no match is found or the break is unacceptable, the focus shifts leftwards to the next syllable. At this point it is convenient to identify 'local' prefixes and suffixes again. However, the difficulty with identifying prefixes within words, leads us, as with the Atex algorithm listed in Chapter 3, to perform suffix searching at this juncture. Searching is repeated until the word is exhausted or the end of a prefix reached. In these cases the procedure exits with the appropriate result.

The user program, i.e. the program calling the hyphenation procedure, is then required to insert the hyphen and re-organise the text structure at the point indicated by the hyphen.

A word passed to the hyphenation procedure is assumed to be in ASCII code, and the first module of the procedure is responsible for extracting the usable characters for the purpose of the procedure. The implementor is invited to amend this to his own requirements.

The procedure is called by the instruction

afrhyp(word, optpoint, endofword, firsttime, result)

where

word	is a string name of the user's work area in the calling program which contains the word to be attempted.
optpoint	is an integer representing the Optimal Point relative to the beginning of the word.
endofword	is an integer representing the length of the word, where the first character of the word is counted 1.
firsttime	is a boolean whose value is TRUE on initialisation and FALSE thereafter.
result	is an integer giving the position of the character after which hyphenation may take place. If this is zero, no acceptable hyphenation was found.

Definitions

Word	letter string defined by address in **word** and **endofword** on entry to procedure.
Natural hyphen	a hyphen which occurs as part of the normal spelling of the word e.g. SOSIO-EKONOMIES
Optimal Point	the address of the last character which would fit onto the current line, allowance having been made for the insertion of a hyphen.
Subword	at any point during the procedure the string of characters stored in the local area **hword** beginning at address pointed to by **front** and terminated at address pointed to by **back**.
Datum	when the search procedure is called in order to search a stored table, datum is the address of the first letter of the string which is to be compared against entries in the table.
Table	a set of consecutive records, each record ending with the character 9. The table terminator is the character *.
Record	a set of logical tests whose function symbols are defined in Search.

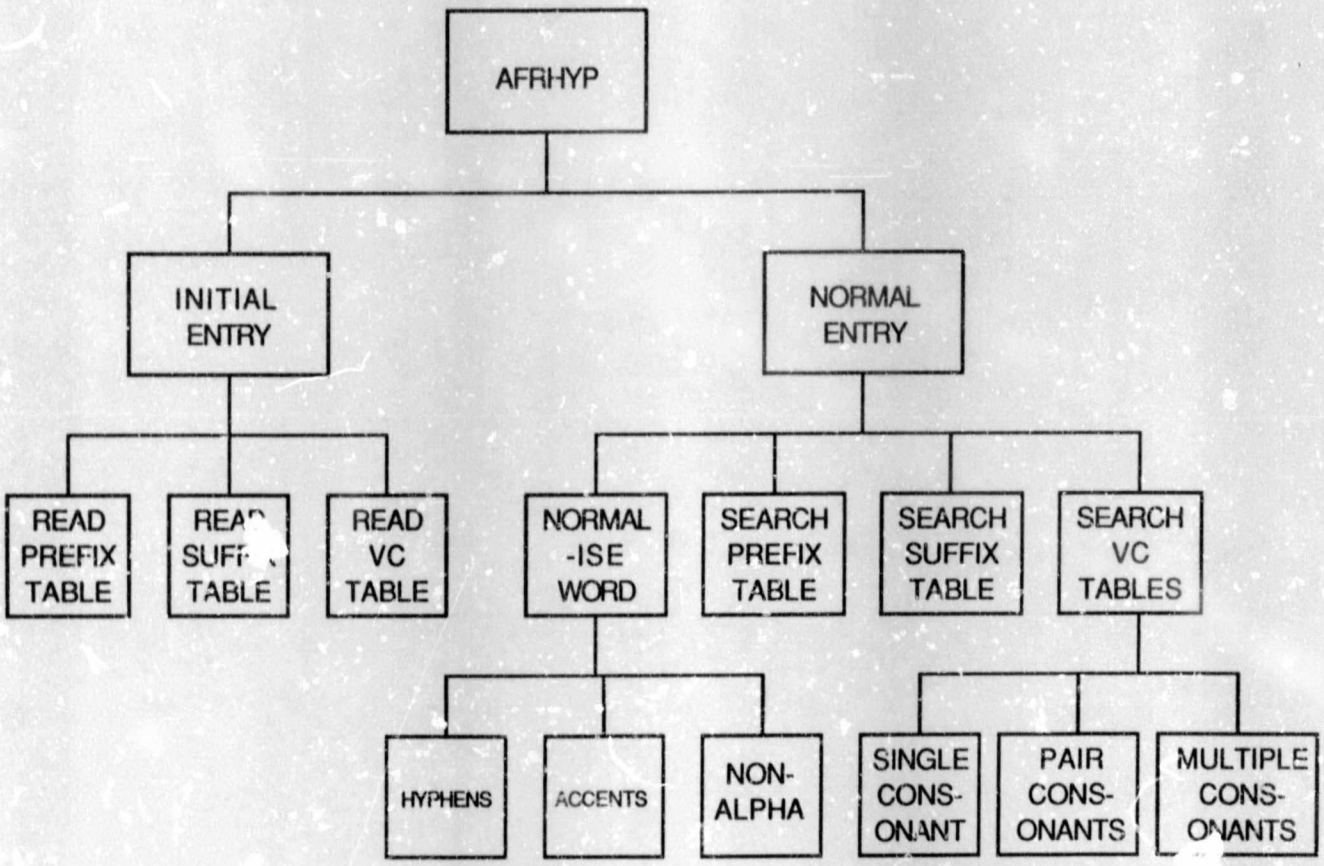

Figure 8.1 *Structure chart of the hyphenation procedure*

Throughout the procedure the following global variables are used.

hword is used to store characters extracted from the source word after all punctuation etc. has been removed. They are in the representational form required for further processing.

hadd is the array which holds the relative address in the original word of each extracted character stored in **hword**.

hopt is value of the Optimal Point in **hword**.

hend relative address of the last letter stored in **hword**.

hcv is the array which holds the vowel (=0) or consonant (=1) value of each character. Accents are stored as -1.

front position of the character forming the beginning of the current sub-word. Starts at the beginning of **hword** i.e. 1, and increases as each prefix is deleted.

back position of the character forming the end of the current sub-word. Starts at the end of **hword** i.e. **hend**, and decreases as each syllable is deleted.

table address of record containing the prefix, suffix, and V/C tables.

On initial entry, *afrhyp* reads 4 tables from a packed binary file. The first table contains relative pointers to the 26 entries in the prefix table. The second table contains pointers the 26 entries in the suffix table. The third table contains pointers to the 4 entries in the CV table. The last table contains the actual prefix table, followed by the suffix table, followed by the CV table.

On subsequent entries, the word is first normalised. This means

1. Non-essential characters are removed
2. Accents are put into their required form
3. All letters are converted to capitals.

If the word was found to be too short or contain a natural hyphen, the word is rejected and the procedure exits immediately.

The following example assumes that the word passed the normalisation, it is now stored as follows.

word	"ongeërg"_	
optpoint	6	i.e. the letter ë
endofword	10	i.e. the space

hword	**0**	**N**	**G**	**E**	**"**	**E**	**R**	**G**
hadd	**2**	**3**	**4**	**5**	**6**	**6**	**7**	**8**
hcv	**0**	**1**	**1**	**0**	**-1**	**0**	**1**	**1**
hend	8							
hopt	6							

The current sub-word is now set to the whole (**hword**) word and prefixes are searched for using the procedure *prefixscan*. If a prefix is found, then the **front** of the current sub-word is updated, and the process is repeated. The process terminates when either no further prefix is found (result=0) or the end of the prefix is to the right of the optimum point (**hopt**). Since such a break is not permitted, by definition, we have to retrieve the end of the previous prefix, if there was one. This is why the variable **previous_front** is used to store this value. If no earlier prefix was found, **previous_front** will still be at the beginning of the word, and the procedure exits without locating an acceptable break.

If there was a prefix, the correct break position in the original word is obtained from **hadd** and returned to the calling program.

Assume that the *prefixscan* completed with a zero result. We now search for suffixes using a similar technique. Here a negative result indicates that suffixes are being found and deleted, i.e. **back** is being updated. Either we run out of suffix matches (result=0), or a suffix deletion causes a break to occur to the left of the optimum point. This is therefore acceptable and is returned to the calling program in the usual way.

Assuming suffixes were exhausted to the right of the optimum point, the procedure now begins the *vcscan*. This acts in a similar way to the suffix search but using V/C tables. It moves leftward until either it bumps into the front of the current sub-word, when it exits without a break point, or it finds an acceptable break point when it returns it as usual.

This completes the main logic.

```
procedure afrhyp   (word      : string;        {address of first character of word          }
                    optpoint  : integer;       {relative address of Optimum Point           }
                    endofword : integer;       {relative address of end of word             }
                    firsttime : boolean;       {true for first entry, false thereafter      }
                    var return: integer);      {result returned to calling program          }
{Written by Quintin Gee, Computer Science Department, University of the Witwatersrand,      }
{Johannesburg, RSA.                                                                         }
{August 1987                                                                                }
{Hardware used: Apple Macintosh connect to fileserver via AppleTalk network.                }
{Software used: Lightspeed Pascal                                                           }
{Compiled version occupies 6996 bytes of memory                                             }
{Used by: calling program. with parameters as defined above                                 }
{Uses procedures: read_tables, normalise_word, prefixscan, suffixscan, vcscan               }
{Globals used: as defined below                                                             }

const
  limit                 = 64;
type {global types}
  index                 = 1..6500;
  letter                = 'A'..'Z';
  short_int             = 1..limit;
  pretables             = array[letter] of index;
  suftables             = array[letter] of index;
  vctables              = array[1..4] of index;
  longstring            = packed array[index] of char;
  datarecs              = record
                            pretable    : pretables;
                            suftable    : suftables;
                            vctable     : vctables;
                            data        : longstring;
                          end;

var {global variables}
  hword                 : array[short_int] of char;    {normalised word stored here        }
  hadd                  : array[short_int] of integer; {original addresses of extracted chars }
  hcv                   : array[short_int] of integer; {c/v value of characters            }
  hopt                  : short_int;                   {value of optimum point in hword    }
  hend                  : short_int;                   {relative address of last letter in hword }
  front                 : short_int;                   {current front of sub-word          }
  back                  : short_int;                   {current end of sub-word            }
  result                : integer;                     {return result from procedures called }
  previous_front        : short_int;                   {value of previous sub-word front so that }
                                                       {prefix searching can back-track    }
  table                 : datarecs;                    {memory-resident data record        }

begin {afrhyp}
if firsttime = true
   then read_tables
   else begin
          normalise_word(result);
          if result = 0 {then word may not be hyphenated}
             then return := 0
             else begin
                    front := 1;
                    back := hend;
                    repeat previous_front := front; prefixscan(result) until result <= 0;
                    if ((result < 0) and (previous_front <> 1))
                       then return := hadd[previous_front - 1]
```

```
                else begin {no more prefixes}
                        repeat suffixscan(result) until result >= 0;
                        if result = 0
                           then begin {no more suffixes}
                                   vcscan(result);
                                   if result = 0
                                      then if previous_front = 1
                                              then return := 0
                                              else return := hadd[previous_front – 1]
                                      else return := hadd[back]
                                   end
                           else return := hadd[back] {suffix left of optimum point. exit}
                        end
                end
        end
end;
```

8.3 Initialisation and Normalisation

This procedure reads the **word** letter by letter provided by the calling program stores the characters it requires in **hword**. The procedure shown below changes diaeresis and circumflex to double quote and @ sign respectively. It looks for hyphens and if found, rejects the word by setting **result** to 0. In addition, it changes lower case letters to upper case letters. All other characters are ignored.

The amended word is placed in **hword** and the corresponding address of each letter in **hadd**. If the word is too short it is also rejected. The vowel/consonant value is stored in **hcv**. It recalculates the position of the optimum point, and stores this in **hopt**.

This procedure will be different for each implementation since the implementor must translate his character representation as follows.

Accents	separated from their letter to precede the letter
Ligatures	separated into their constituent parts. The implementor is also responsible for re-constituting ligatures on exit.
Lower case	changed to upper case

```
procedure read_tables;
{reads prefix tables and their indexes; reads suffix tables and their indexes                }
{reads vc tables and their indexes                                                           }
{Compiled version occupies 106 bytes of memory                                               }
{Used by: afrhyp, with no parameters                                                         }
{Uses procedures: none                                                                       }
{Globals used: table.                                                                        }
```

```
var sourcefile          : file of datarecs;
begin
reset(sourcefile,'binarytable');
table := sourcefile^;
close(sourcefile);
end; {read_tables}

procedure normalise_word (var result : integer);
{Compiled version occupies 1448 bytes of memory                                    }
{Used by: afrhyp, with parameters as defined above                                 }
{Uses procedures: none                                                             }
{Globals used: word, optpoint, endofword, hword, hadd, hend, hopt, hcv.            }
{Returns:  0 if word may not be hyphenated, ≠ 0 otherwise                          }
var word_ptr            : integer;          {indexes original word                 }
    hword_ptr           : integer;          {indexes normalised word into hword    }
    finish              : boolean;

  function consonant (a : char) : integer;
  begin
  if a in ['B', 'C', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'X', 'Z']
     then consonant := 1 else consonant := 0;
  end;

begin
word_ptr := 1;
hword_ptr := 1;
result := 1;
finish := false;
hopt := optpoint;
repeat
case word[word_ptr] of
  '-' : begin result := 0; finish := true end;      {word contains a hyphen; reject it, exit }
  '"' : begin
        hword[hword_ptr] := '"';                    {change diaeresis to double quote        }
        hadd[hword_ptr] := word_ptr + 1;
        hcv[hword_ptr] := -1
        hword_ptr := hword_ptr + 1;
        end;
  '^' : begin
        hword[hword_ptr] := '@';                    {change circumflex to @ sign             }
        hadd[hword_ptr] := word_ptr + 1;
        hcv[hword_ptr] := -1
        hword_ptr := hword_ptr + 1;
        end;
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
  'X', 'Y', 'Z' :
        begin
        hword[hword_ptr] := word[word_ptr];
        hadd[hword_ptr] := word_ptr;
        hcv[hword_ptr] : = consonant(hword[hword_ptr])
        hword_ptr := hword_ptr + 1;
        end;

  'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' :
        begin                                       {assumes ASCII code              }
        hword[hword_ptr] := chr(ord(word[word_ptr]) - 32);
        hadd[hword_ptr] := word_ptr;
        hcv[hword_ptr] : = consonant(hword[hword_ptr])
```

```
            hword_ptr := hword_ptr + 1;
            end;
  otherwise begin
                if word_ptr <= optpoint then hopt := hopt - 1;
                if hopt < 2 then begin result := 0; finish := true end
                end;
end;
word_ptr := word_ptr + 1;
if word_ptr > endofword then finish := true;
until finish;
if hcv[hopt] = -1 then hopt := hopt + 1;
if hword_ptr < 5  then result := 0 else hend := hword_ptr - 1;
end; {normalise_word}
```

8.4 Syllable Scans

The procedure *prefixscan* searches for one prefix. It picks up the first letter of the current sub-word and accesses the prefix table beginning with that letter. If the first character in the table is *, the table is empty and the procedure exits with result 0, otherwise it searches the table for a match using the generalised *search* procedure. If no match is found by *search*, then *prefixscan* exits with result 0.

If a match is found, the break point indicated is tested against the optimum point. If the break is to the right of the optimum point, *prefixscan* exits with negative value. If the break is to the left, the **front** of the current subword is recalculated. *prefixscan* then returns a positive value.

```
procedure prefixscan (var result : integer);
{Compiled version occupies 276 bytes of memory                                          }
{Used by: afxhyp, with parameter as defined above                                       }
{Uses procedures: search                                                                }
{Globals used: table, front, back, hword, hopt, pretable, data.                         }
{Returns:  0 if table empty, no match, word too short. +1 if match found, and to left of optimum  }
{          point. +1 if match found, and to left of optimum point.                      }
var found               : integer;
    i                   : index;

begin
if back - front < 2
   then result := 0
   else begin
           i := table.pretable[hword[front]];
           if table.data[i] = '*'
              then result := 0
              else begin
                      search(i,front,found);
                      if found = 0
                         then result := 0
                         else if back - front <= found
```

```
                then result := 0
                else if front + found - 1 <= hopt
                        then begin result := +1; front := front + found end
                        else result := -1
            end
        end
end; {prefixscan}
```

The procedure *suffixscan* searches for one suffix. It picks up the last letter of the current sub-word and accesses the suffix table corresponding to that letter. If the first character in the table is *, the table is empty and the procedure exits with result 0, otherwise it searches the table for a match using the generalised *search* procedure. If no match is found by *search*, then *suffixscan* exits with result 0.

If a match is found, the break point indicated is tested against the optimum point. If the break is to the right of the optimum point, *suffixscan* exits with negative value. If the break is to the left, the **back** of the current subword is recalculated. *suffixscan* then returns a positive value.

```
procedure suffixscan (var result : integer);
{Compiled version occupies 258 bytes of memory                                  }
{Used by: afrhyp, vcscan, with parameter as defined above.                      }
{Uses procedures: search                                                        }
{Globals used: table, front, back, hword, hopt, suftable, data.                 }
{Returns 0 if table empty, no match, word too short, match found to be too close to front.
{+1 if match found and to left of optimum point. -1 if match found and to right of opt. point. }
var found               : integer;
    i                   : index;

begin
if back - front < 3
   then result := 0
   else begin
          i := table.suftable[hword[back]];
          if table.data[i] = '*'
             then result := 0
             else begin
                    search(i,back,found);
                    if found = 0
                       then result := 0
                       else if back - front <= found
                              then result := 0
                              else begin
                                     back := back - found;
                                     if back <= hopt then result := +1 else result := -1
                                     end
                    end
          end
end;{suffixscan}
```