Rigorous Re-Design of Knuth’s Solution to the Common Words Problem

Prabhaker Mateti
Department of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435
pmateti@wright.edu  (937) 775 5114

Abstract  We reverse engineer Knuth’s solution to the Common Words Problem (CWP) as an example of how the designs of intricate programs might be presented using rigorous justification. The cwp and Knuth’s solution use data structures known as dictionaries, and hash tries, and notions such as lexical structure. These have been the main source of ambiguity. We give precise definitions for all these in a design specification language called ÔM. We explicitly define all our objects and also exhibit the design hierarchy that we were able to reverse engineer from his solution.

march 2013; selected pages only
# Contents

1 **Introduction**  
1.1 Goals of this Paper ........................................... 1  
1.2 On Design ..................................................... 2  
1.2.1 Vertical Refinement ...................................... 2  
1.2.2 Horizontal Decomposition ............................... 3  
1.2.3 Rigorous Description of Software Designs .......... 3  
1.2.4 Formal Languages for Software Design ............... 4  
1.3 Structure of the Paper ...................................... 4  

2 **The Common Words Problem**  
2.1 Problem Domain: Text ....................................... 5  
2.1.1 Occurrences of a Word .................................. 6  
2.1.2 \( k \) Most Frequent Words ............................. 7  
2.1.3 Output .................................................... 8  
2.2 Overview of CWP Designs .................................. 9  
2.2.1 Design D0 with a Generic Container of Words ........ 9  
2.2.2 Mapping Input Text to Words ........................... 10  
2.2.3 Design Levels ............................................. 12  

3 **Design D1 Using a Bag of Words**  
3.1 Design D1 .................................................... 14  

4 **Dictionaries**  
4.1 Tables ......................................................... 14  
4.2 Design D2 ..................................................... 15  
4.3 Mapping a Dictionary to a Bag of Words ............... 16  

5 **Sorted Sequences of \((w, n)\) Pairs**  
5.1 Design D3 .................................................... 17  

6 **N-ary Trees**  
6.1 Search for a Word .......................................... 19  
6.2 Design D4 ..................................................... 20  
6.3 N-ary Tree to Bag of Words ............................... 21
1 Introduction

Program documentation is an art. Unfortunately, we have only a few examples. [Knuth 84] suggests treating it as ‘literate programming’, and has contributed several examples, both large (\TeX [Knuth 86a], Metafont [Knuth 86b]) and small ([Knuth 84] and [Bentley 86]).

The tradition pioneered by [Kernighan and Plauger 76] is continued in such books as [Comer 84], [Tanenbaum 87], [Wirth and Gutknecht 92] and [Fraser and Hanson 95], which include complete listings of source code, along with cogent explanations of why they work. These are impressive accomplishments. But they neglect to emphasize design descriptions, and concentrate on implementation details.\footnote{Update!}

The June 1986 Programming Pearls column [Bentley 86] posed the following problem:

“Given a text file and an integer \( k \), print the \( k \) most common words occurring in the file (and the number of their occurrences) in decreasing frequency.”

We refer to this statement as the \textit{Common Words Problem} (\textit{cwp}), and the desired program as \textit{cwp}.

1.1 Goals of this Paper

There is a fundamental difference between Knuth’s examples, and the books by others mentioned above. Knuth’s examples are meant for peers to read, understand and evaluate, whereas the above books are for students to emulate and learn the programming techniques. The designs of Knuth’s examples get buried in a myriad of surface details. The descriptions in the other books are too imprecise.

There are now a large number of large open source projects, with thousands of pages of documentation, but with hardly any design descriptions.

What should design descriptions of software contain? How should they be organized? These two questions are implicitly answered, for the \textit{cwp} problem, by the material of this paper and its organization. We convey our concerns for the precise expression of software designs by reworking the \textit{cwp}. It was solved
2.2 Overview of CWP Designs

We will be presenting seven levels of design (see Section 2.2.3) all sharing the structure shown below.

2.2.1 Design D0 with a Generic Container of Words

Our main concern at the highest level of design is functionality.

```plaintext
module D0(k: nat, fin: word, fout: word) := (  
    import module lex;
    import module cow;
    init (  
        var itx := file-content(fin);
        lex.init(itx);
        var cw := cow.init(lex.nextlexeme);
        cw.build-all-words();
        file-content(fout) := cw.find-frequent-words(k);
    )
)
```

The cow module provides a container of words.\(^7\) It needs to be supplied with function nextlexeme that takes one nat argument and returns a pair (nat, nat). The variable cw is initialized to being empty. The parameterless procedure build-all-words() inserts into cw all the words found in the content of file fin. The file named fout will have its content reset to the string built from the k most frequent words.

\(^7\) Don't expose itx. Give lex.init the fin.
module cow(
  function nextlexeme(nat) (nat, nat) ) := ( 

    init ( 
      var cw := ... empty ... ;  
    )

    let old-word(w) == ... w is in cw ... ;  
    let incr-count(w) == 
      ... w is already in cw, now with an extra occurrence ... ;  
    let add-new-word(w) == ... w was not in cw, now it is ... ;

  procedure build-all-words() := ( 
    var m, n: nat;  
    var i: nat := 1;

    while (  
      (m, n) := nextlexeme(i);  
      if (m > n => break);  
      let w == itx[m..n];  
      if (  
        old-word(w) => incr-count(w);  
        else => add-new-word(w)  
      );  
      i := n + 1;  
    )
  )
)

In the above, the “comments” enclosed in ellipses are formal comments expected to be resolved by supplying actual code in ÔM later.

2.2.2 Mapping Input Text to Words

Function nextlexeme examines itx[i..], without modifying it, and establishes the borders of the next word. We wish to construct the cw incrementally by adding each word delivered by nextlexeme.
nextlexeme: A Spec

The following is a specification, not a design, of function nextlexeme.8

```plaintext
function nextlexeme(i: nat) := value (m: nat, n: nat)
is such that
( ( itx[m..n] in word,
  i <= m <= n <= #itx,
  n < #itx -> itx[n+1] in delimiters,
  set(itx[i..m-1]) <= delimiters
 )
 or
 (m > n <-> set(itx[i..]) <= delimiters)
);
```

nextlexeme: A Design

We now present a design of the above that maps the given sequence of characters in the input file into a sequence of words as and when needed is described here. This module is not further refined.9

```plaintext
module lex(itx: text) := (  
  assert (itx[#] in delimiters, itx[#-1] !in delimiters);

  procedure nextlexeme(i: nat) pre (i < #itx) :=
  var (m: nat, n: nat) such that (  
    m := i;
    while (itx[m] in delimiters => m := m + 1);
    n := m;
    while (itx[n] !in delimiters => n := n + 1);
    n := n-1;
  )
);
```

8. ÔM: In the context of sets, the token <= stands for the subset-of relation.
9. Where did we make sure that itx[last] is a delimiter?
2.2.3 Design Levels

In what follows, the data structure \( \text{cw} \) will be refined several times. Our first design D1 uses a bag of words as \( \text{cw} \), which is quickly refined into an ordered set of pairs. Each pair consists of a word, and its frequency count in the bag in order make the operations old-word(\( w \)), incr-count(\( w \)), add-new-word(\( w \)) efficient. The set is ordered alphabetically by the spelling of the words it contains.

The representation is progressively refined from a table (designs D2 and D3) to an \( n \)-ary tree (design D4), to a trie (designs D5 and D6), and finally to a hash trie (design D7).

In all the designs, we build-... first then we find-... This suggests that we may choose one representation for \( \text{cw} \) during the build-..., and a different one for the find-..., transforming the representation once between the two. During the find-..., in order to efficiently discover the most frequent word, it would be best if \( \text{cw} \) were a set of word-count pairs ordered not alphabetically but by the frequency counts. Knuth does this by progressively converting, in situ (i.e., without using additional memory), the (hash) trie into a linked list. This is perceived by many readers as tricky and presents us with one more layer of abstraction, from design D5 to D6.

3 Design D1 Using a Bag of Words

Even though the specification (Section 2.1.2) is considering all the words when it says for \( x: \text{word} \ldots \), in the design we need consider only the words occurring in the input text, \( \text{itx} \). On the other hand, we must examine each and every word of \( \text{itx} \), otherwise we may miss the most frequent word, or have wrong frequency counts.

It is our goal to generate a/the piece of text that satisfies the output requirement of the specification progressively as the value of the variable \( \text{otx} \), which stands for file-content(fout).

The following design satisfies the \( \text{cwp} \). Functions nextlexeme is specified in Section 2.2.2.

```plaintext
module bow
   ( function nextlexeme(nat) (nat, nat) ) :=
```

12
10 Function mostfrequent examines the bag \( bw \), and returns a most frequent word and its frequency. Note that we deliberately choose not to uniquely specify which word is to be returned when there are several equally frequent ones in \( bw \).

```plaintext
function mostfrequent() pre (#bw > 0) :=
value wn: wornat such that
( wn.n = wn.w #in bw,
  for x: bw (x #in bw <= wn.n)
);
```

```plaintext
procedure find-frequent-words(k: nat) := var otx: text ( 
  var w: word;
  var i, n: nat;
  otx := [];
  for i in {1..k} ( 
    if (bw = {} => break);
    (w, n) := mostfrequent();
    bw := bw - {w ** n};
    otx := otx | w | [blank] | itoa(n) | [newline]
  )
)
```

10. The empty parens need explanation. Maybe old-word should be “true”?
3.1 Design D1

The design D1 is essentially the same as D0.

```plaintext
module D1(k: nat, fin: word, fout: word) := ( 
    import module lex; 
    import module bow; 
    init ( 
        var itx := file-content(fin); 
        lex.init(itx); 
        var bw := bow.init(lex.nextlexeme); 
        bw.build-all-words(); 
        file-content(fout) := bw.find-frequent-words(k); 
    ) 
)
```

4 Dictionaries

Conceptually, a dictionary is a collection of objects organized in a particular way to ease subsequent search of these objects. Each object in such a collection is attached various attributes of interest. For our purposes here, the only attributes of interest are its spelling and its frequency of a word.

4.1 Tables

A few subtleties aside, the `tables` of ÔM can model the dictionaries nicely. A table is a set of like tuples whose first elements are all distinct. A tuple is similar to a sequence but may contain dissimilar items. If $T$ is a table, $T.i$ denotes the collection of all the items of the $i$-th column of $T$. $T[e]$ denotes that tuple of $T$ whose first component is $e$; thus, $T[e].1 = e$ always.

```plaintext
module dict ( 
    function nextlexeme(nat) (nat, nat) ) := ( 
```
4.3 Mapping a Dictionary to a Bag of Words

function dictionary(bw: bag of word) :=
    value d: table wornat such that
        ( for w: set(bw) ( d[w].n = w #in bw ),
          for w: d.w ( d[w].n = w #in bw )
        );

The above defines a relationship between d and bw. Clearly, we want all the words in the bag bw appear in the first column of the table d with the correct count: for w: set(bw) (d[w].n = w #in bw). The second line is requiring that whatever words are in the first column of the table, their occurrences count in the bag be correct. The second line could have been written equivalently as for w: d.1 - set(bw) ( d[w].2 = 0 ).

In other words, we are allowing for the possibility of non-bw words to appear in the table. This happens to be a significant and insightful jump in the design process.

As can be readily seen, dictionary(\{| |\}) = {}. Suppose dwn = dictionary(bw). Then after add-new-word(w) and after the if-statement in build... we will have the same relationship holding with the updated values for dwn and bw.
5 Sorted Sequences of \((w, n)\) Pairs

Let us consider a design where we have the dictionary continually sorted for ease of searching for a word. During the building up of this “dictionary” it will be maintained as a sequence of tuples, var \(\text{alpha-wnq}: \text{seq wornat}\), sorted based on the alphabetic order of words.

\[
\text{function alphasorted(q: seq wornat) := (for i: 2..#q (q[i - 1].w <= q[i].w));}
\]

5.1 Design D3

Design D3 refines D2 by using module \(\text{alpha-sorted-dict}\) instead of \(\text{dict}\). D3 does not refine the procedure \(\text{find-frequent-words}\) of D2 further.

```
module alpha-sorted-dict(value itx: text) :=
  import module lex(itx);
  var alpha-wnq : seq wornat := [];

  let old-word(w) == (w #in alpha-wnq.w > 0);
  let incr-count(w) == (alpha-wnq[i].n := 1 + alpha-wnq[i].n);
  let add-new-word(w: word) ==
    alpha-wnq := value uqwn: wornat such that
    ( set(uqwn) = set(alpha-wnq) + { <w, 1> },
      alphasorted(uqwn) );

  procedure build-all-words() := as-in module D2;
  post set(alpha-wnq) = dwn;

  procedure find-frequent-words(k: nat) := as-in module D2;
```

Note the post-condition \(\text{set(alpha-wnq)} = \text{dwn}\). The dictionary \(\text{dwn}\) was allowed to contain certain words with zero counts; hence, \(\text{alpha-wnq}\) also will. But neither \(\text{dwn}\) nor \(\text{alpha-wnq}\) have indicated specifically what the characterization of these zero-count words are.
6 N-ary Trees

An ordered $n$-ary tree is a rooted tree, where each node has at most $n$ ordered subtrees; see Figure 1. In cwr, we store in each node a letter, and a count. The path from the root to a node yields a word made up of these letters. The cnt field of a node contains the number of times the word represented by the path to this node occurs in the input text itx. The cnt fields of some nodes may be zero since not every prefix of a word occurs as a word in itx.

We also would like to alphabetically order the subtrees of every node based on the letters in their roots.

```haskell
type ntree-ao :=
tuple (  
  ltr: letter,  
  cnt: nat,  
  stq: seq ntree-ao  
) such that (  
  for all t: ntree-ao  
    ( sorted(t.stq.ltr) )  
);
```

Figure 1: Example $n$-ary tree
\[ p \leq \#w \text{ and } i = 0 \text{ otherwise.} \]

### 6.2 Design D4

D4 refines \( \text{old-word}(w) == (w \text{ #in alpha-wnq.w > 0}) \) of the preceding section into \( \text{search}(w).2 = \#w + 1 \). The initialization \( \text{var } nt: \text{n-tree-ao} := (' ', 0, []) \) produces an \( n \)-ary tree that has just one node (the root) containing the letter blank, the count zero, and the empty sequence as its \( \text{stq} \).

```plaintext
module D4(k: nat, fin: word, fout: word) := (  
  import type ntree-ao;  

  init (    
    var itx := file-content(fin);    
    var nt: ntree-ao := (' ', 0, []);  
  );  

  import module lex(itx);  

  let (tw, pw, iw) == nt.search(w);  
  let sq == tw.stq;  

  let old-word(w) == (pw = \#w + 1);  
  let incr-count(w) == (sq[iw].cnt += 1);  
  let add-new-word(w) == (    
    let m == min (    
      \#sq + 1 \} +    
      \{j -: 0 < j < \#sq + 1, w[pw] < sq[j].ltr\};    
      sq[ @ m := mk-ntree-ao(w[pw..]) ]);  
  
  procedure build-all-words() := as-in D3;  
  procedure find-frequent-words() := as in D3;  
)
```

20
7.1 The trie

For our discussion here, cellid is any arbitrary type that has a “sufficient” number of values. We define four tables whose keys (i.e., the first components of its elements) are values from this type. A trie is a subset of cellids, and the four tables that collectively satisfy certain constraints. These constraints amount to requiring that the structure we are defining better be a binary tree.

```plaintext
type cellid;
value emt: iletter := 1 + max(#upletter, #loletter);
value hdr: iletter := 0;

value nilid : cellid := new-cellid();
value rootid: cellid := new-cellid();

type trie := table (cid: cellid,
ltr: iletter,
cnt: nat,
nxt: cellid,
hic: cellid)
```

Figure 2: Example plain trie
Knuth [Bentley 86] used our hdr value as his emt and vice-versa.

7.1.1 req-nilid-rootid()

We reserve two values, that we name as nilid and rootid, from the cellid. Every value $t$ of type trie will be such that nilid and rootid are in $t.cid$.

function req-nilid-rootid(t: trie) := (  
  t[nilid].ltr = emt, t[rootid].ltr = emt,  
  t[nilid].cnt = 0, t[rootid].cnt = 0,  
  t[nilid].nxt = nilid, t[rootid].nxt = nilid,  
  t[nilid].hic = nilid, t[rootid].hic = nilid 
)

7.1.2 the-children-are-ordered()

The children of a node $u$ are ordered based on the letters they contain. The list of children starts with hic($u$), and the rest is given by nxt$i$(hic($u$)). The last child has no next. If a cellid $u$ has no children, hic($u$) = nilid. Otherwise, the value $v = hic(u)$ is the cellid of the child $v$ of $u$ containing the highest letter among the children of $u$. If $y = nxt(x)$ is not nilid, we require that $x$ be a sibling of $y$, that is parent($x$) = parent($y$), and ltr($y$) < ltr($x$). Obviously, nxt$i$(d) = nilid, for some $0 \leq i \leq \#letter.$
becomes h. See Figure 3. The parent of h is u. The node h of course has no children, and there is no meaning yet for either hic(h), or ltr(h). We define hic(h) as u, and ltr(h) = dot. Thus, for all cellids x, other than the rootid, either hic(hic(x)) = x or hic(x) = nilid.

```plaintext
type ringed-trie := trie except (  
  function childrenq(u: cids) :=  
    value q: seq cids such that  
    if (  
      hic(u) = nilid => q = [];  
    else => (  
      q[1] = hic(u),  
      ltr(q[1]) = hdr,  
      hic(q[1]) = u,  
      nxt(q[1]) = q[#],  
      for i: 1..#-1 (  
        q[i] = nxt(q[i+1]),  
      )  
    ) )  
)
```

Figure 3: Example Trie with Rings PPPP...PPP TDB
7.3 Design D5

module D5(k: nat, fin: word, fout: word) := (  

    import type ringed-trie;

    init (  
        var itx := file-content(fin);
        var t: ringed-trie := {};
    );

    import module lex(itx);

    let (vi, pn, ui) == t.search(w);
    let old-word(w) == pn = #w;
    let incr-count(w) == cnt(nxt(ui)) += 1;
    let add-new-word(w) == t.mk-ring-trie(ui, pn+1);
    procedure build-all-words() := as-in module D4;

    procedure find-frequent-words() := ... see below ...;
)

7.4 Frequency Sorting of the Words

7.4.1 find-frequent-words() with foq

We now refine the procedure find-frequent-words of D2 further. We introduce (temporarily) an extra field foq to our trie that will contain a “linked-list”, frequency-ordered, of all the words with non-zero counts.
### 8.1 Cell-Ids Refined

The cell-ids now become natural numbers with a certain numerical relationship among the parent and children. Suppose the cell-ids u and v are siblings. Let \( acn \) be a function, yet to be discussed, that maps cell-ids of the preceding section to cell numbers. We will select the mapping \( acn \) in such a way that the integer

---

#### Figure 4: Example hash trie [Bentley 86](p 479)
 hic(r + d) := hic(r);
 if (hic(r) != nilid => hic(hic(r)) := r + d);
 ltr(r) := emt;
 r := nxt(r);
)
)

As before, add-new-word(w) == hash-trie.make(...). This adds tuples to
the global hash trie var t. The mk-hash-trie is more complex than mk-ring-trie
because we cannot merely choose any new but arbitrary cellid for the letters to
be inserted.

procedure hash-trie.make(ri, ui: cellid, k: nat) := ( 
 var pi, hi, ni: cellid;

 oh := hic(ri);
 hi := compute-loc(oh, w[k]);
 if (hi != oh) => relocate-children(oh, hi));
 pi := hi + w[k];

 ui := hi + ui - oh;
 insert-ltr(w[k], pi, ui);

 for j: nat := k+1 .. #w ( 
  hi := compute-loc(oh, w[j]);
  ni := hi + w[j];
  insert-ltr(hdr, hi, hi);
  insert-ltr(w[j], ni, hi);
  hic(pi) := hi;
  pi := ni;
);
 cnt(pi) := 1;
);

Function compute-loc returns a possibly new location for the header implying
that the siblings group needs to be relocated; nh equals oh if there is no such need.
procedure hash-trie.next-hdr-loc(oh: cellid) := var nh: cellid
if (oh = last-h => nh := 0;
    oh = trie-size - NC => nh := NC + 1;
    else => nh := oh + 1)
)

procedure hash-trie.compute-loc(oh: cellid, a: ltr) := var nh: cellid (nh := oh;
if (ltr(h + a) /= emt =>
while (nh := next-hdr(hn);
    if (will-they-fit(a, oh, nh) => break);
))

In the function below, a node containing the a: iletter will become the child of a certain node p, whose header is presently at oh and we wish to move it to nh. The a is chronologically the latest child to join the siblings. Function will-they-fit is true iff the cells d units away from each of the children of p are vacant. The distance d can be a negative integer.

function hash-trie.will-they-fit
(a: iletter, oh, nh: cids) := value b: boolean (let q == siblings(oh);
let d == nh - oh;
pre ({oh, nh} * {rootid, nilid} = {});
post b = (ltr(nh + a) = emt, for u: q (ltr(u + d) = emt));
);

One letter code is inserted by insert-ltr().

procedure hash-trie.insert-ltr
(a: iletter, an: cellid, pn: cellid) := (ltr(an) := a;
    cnt(an) := 0;
    hic(an) := nilid;
The procedure find-frequent-words is the same as before except for \texttt{fop} is replaced by \texttt{nxt}. “After \texttt{trie-sort} has done its thing, the linked lists \texttt{sorted[largecount]}, \ldots, \texttt{sorted[1]} collectively contain all the words of the input file, in decreasing order of frequency. Words of equal frequency appear in alphabetic order.” \cite{Bentley 86}

8.7 Design D7

```plaintext
module D7(k: nat, fin: word, fout: word) := ( 

import module lex;
import module hash-trie;

init ( 
    var itx := file-content(fin);
    lex.init(itx);
    var t := hash-trie.init(lex.nextlexeme);
    t.build-all-words();
    file-content(fout) := t.find-frequent-words(k);
)
);

module hash-trie ( 
    function nextlexeme(nat) (nat, nat) ) := ( 

    <hash-trie.*>

    let (vi, pn, ui) == t.search(w);
    let old-word(w) == pn = #w;
    let incr-count(w) == cnt(nxt(ui)) += 1;
    let add-new-word(w) == t.mk-hash-trie(...);

    procedure build-all-words() := as-in module D6;
```
structs, we have been able to precisely “specify a design solution” to the common words problem.[Diby 90]

References


