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.

Contents

1 Introduction **1**
- 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 **4**
- 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 **12**
- 3.1 Design D1 . . . 14

4 Dictionaries **14**
- 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 **17**
- 5.1 Design D3 . . . 17

6 N-ary Trees **18**
- 6.1 Search for a Word . . . 19
- 6.2 Design D4 . . . 20
- 6.3 N-ary Tree to Bag of Words . . . 21

7 Tries **21**
7.1 The `trie` . . . 22
7.1.1 `req-nilid-rootid()` . . . 23
7.1.2 `the-children-are-ordered()` . . . 23
7.1.3 `parent-is-unique()` . . . 24
7.1.4 Search for the Word . . . 25
7.2 Tries with Rings . . . 25
7.2.1 Insert New Word . . . 27
7.2.2 `word-from-trie()` . . . 27
7.3 Design D5 . . . 28
7.4 Frequency Sorting of the Words . . . 28
7.4.1 `find-frequent-words()` with `foq` . . . 28
7.4.2 Preparing to dispense with `foq` . . . 30
7.5 Design D6 . . . 31

8 Hash Tries **31**
8.1 Cell-Ids Refined . . . 32
8.2 Word from the Hash Trie . . . 33
8.3 Initial Hash Trie . . . 33
8.4 Search of a Word . . . 34
8.5 Insertion of a Word . . . 34
8.6 Sorting the Words by Frequency . . . 37
8.7 Design D7 . . . 38

9 Example Evaluation **39**
9.1 Critique of Knuth's Solution . . . 39
9.2 ÔM . . . 39

10 Conclusion **40**

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.[1]

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 Common Words Problem*(CWP), and the desired program as `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 implicilty answered, for the 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 CWP. It was solved

1. Update!

by that grand master Knuth himself as an example of literate programming and as an example usage of WEB. Knuth's solution had served its purpose. In fact, its reviewer McIlroy found "... Knuth's program convincing as a demonstration of WEB and fascinating for its data structure." The complexity of that fascinating data structure, called the hash trie, was a major osbtacle in its understandability. On the other hand, thanks to this complexity, Knuth's program provided enough material for the WEB system in describing "real" and complex systems. An alternate solution to the CWP was given by Hanson [Van Wyck 87].[2]

This paper deals with the *expression of the design ideas already present* in Knuth's solution. It does not take sides on the question of whether hash tries are too complex a solution to be used for simple problem such asCWP. If hash tries are indeed appropriate in a situation, and a designer chose to use them, how should the concept of hash tries and its incorporation into the solution be presented is what this paper deals with. Our goal is to explain the design with precision.

1.2 On Design

A design document captures the end result of having designed something. The decomposition of the given problem into subproblems, and "judging" the effectiveness of the decomposition, the previous approaches to similar problems that the designer considered — all these must be recorded. In the long run, other designers can benefit from such clarification and cataloguing of design principles, and greater understanding of the structure of design descriptions.

A design is a plan of realizing an object from given primitives, materials and environment while meeting its specifications. A software design is a "program" written in an appropriately chosen language. A software specification and design needs to be captured in machine processible form. Since design documents are meant for humans also, it is equally important to include natural language descriptions of the formal objects manipulated in the design.

1.2.1 Vertical Refinement

At the highest level, a software design can be (ought to be?) state-free and highly declarative. Intermediate level designs will look similar to programs in a proce-

2. lecture I watched on video has a good summary.

dural language, but manipulate abstract data types such as sets, bags, and tables. At the most detailed level, our designs are "only a step away" from a modern programming language.

The levels are based on *abstraction*. As to when something is abstract or concrete is ultimately a subjective matter, but it is based on the architecture of the target computer and programming language. Also, abstraction is not an either-you-have-it-or-you-don't item. We call the process of adding detail to an abstract thing to make it more concrete *vertical refinement*. Reification is an alternate term that [VDM] uses.

1.2.2 Horizontal Decomposition

We call the splitting of the given problem into subproblems, and grouping the internals of each subproblem *horizontal decomposition* giving us modules. A design module describes

- compositions of parts
- functionality of parts
- data structures + their properties
- algorithms + their properties
- reasons for design decisions made
- perceived "good" (possibly "unprovable") properties

1.2.3 Rigorous Description of Software Designs

The problem and Knuth's solution use structures that are 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. Natural languages, and so-called pseudo-code are unsuitable. It is important to note that our definitions of these structures are not only precise, and rigorous but also *formal*. Most definitions in combinatorics and algorithms books, including those of Knuth, are *informal*.

1.2.4 Formal Languages for Software Design

We use our formal language ÔM, but without getting wholly wrapped up in ÔM. We describe essential details of ÔM as needed.[3]

1.3 Structure of the Paper

This paper is organized into three parts. Section 2 clarifies the description of CWP by providing a set of precise definitions, which remove the ambiguities from the original informal specification of the structure of the input. Section 2 covers all that is needed to precisely describe the lexical structure of the input.

Section 3 of the paper presents background "design knowledge" appropriate for the description of Knuth's solution to CWP. It defines objects such as dictionnaries, tries, hash tries, and some function representations used in the solution of CWP.

Sections 4 – 8 is the third part of the paper dealing with the actual design specification. It presents a hierarchy of designs, each described at a different level of abstraction. It also describes the interrelationship between two successive designs in the hierarchy. Also, along with the increasing level of detail, more concrete representation of the abstract objects used in the early design are progressively introduced. Thus, the first design in the hierarchy uses a table as the representation of the dictionary, describes input/output relationships, and assumes the natural number k to be some value. The second design also uses this representation of the dictionary; however, it provides a more more operational view of the designs by explicitly expressing how one obtains the output from the input in a more constructive fashion.

Finally, there is an overall evaluation of the design example.

2 The Common Words Problem

As posed, the CWP problem is imprecisely stated. For instance, it is not clear what sequences of characters constitute valid words, whether a mere difference in character cases makes two words to be considered different, how the integer k is to be input, what to do if the input file contains less than k words, more than

3. Delete this section?

k words occur all with the same frequency, or if k is negative. Nothing is said about the format of the output. Many of these points have been raised by David Hanson (as reported in [Van Wyck 87]). We make the problem statement more precise by resolving these issues in the next section.

Some would point out that a "precise" specification is not possible to express in any natural language. True as this may be, we quickly understand, roughly perhaps, the details when explained in a natural language using well-accepted technical terminology, and defining our new terms. It is worth remembering the points made by [Meyer 85].

2.1 Problem Domain: Text

A *text* is a sequence of characters. A line is a non-empty sequence of characters such that the last character is a newline, and no other character in it is a newline. A *text file* is an association of a file-name with a file-content. The file-content must be a text. The file-name must be a word.

```
type text := seq char;
type T;

function set(q: seq T) :=
  { q[i] such that 1 <= i <= #q };

value upletter: text := ...; // for English: ['A'..'Z']
value loletter: text := ...; // for English: ['a'..'z']
assert set(upletter) * set(loletter) = {};
value letters := set(upletter) + set(loletter);
value delimiters := char - letters;
```

A *word* is a non-empty sequence of letters, upper or lower case. [4] We do not wish to say something like "A word is a maximal subsequence of one or more contiguous certain-kind-of-characters of the input file" because we must be able to talk about a word independently of a text file.

4. ÔM: Sets and sequences are built-in ÔM data structures. If `q` is a sequence, `#q` is the length of `q`, `q[i]` gives the `i`-th item of `q`, and `q[#]` gives the last item of `q`. The vertical bar `|` between sequences denotes catenation; `[]` denotes the empty sequence. The expression `c ## s` yields the smallest index position `i` such that `s[i] = c` if `c` occurs in `s`, `0` otherwise.

```
type  word := (seq letters) - { [] };
```

The `lettercode` function gives the ordinal position of a letter in the alphabet. Its definition exploits the fact that one of the two terms in `(a ## loletter) + (a ## upletter)` will be 0.TBD: Explain at symbol. ;;

```
function  infix "=" (s, t: word) :=
  (lettercode@(s) = lettercode@(t));

function  infix "<" (s, t: word) :=
  for some z: word (t = s | z)
  or
  for some i: 1..min(#s, #t)
    ( s[1..i-1] = t[1..i-1],
      lettercode(s[i]) < lettercode(t[i])
    )
  ;
```

Uppercase and lowercase letters are considered equal. Two words are *equal* if they are string-equal after ignoring the lower/upper case distinction. For example the two words `ugLy` and `Ugly` are equal.

2.1.1 Occurrences of a Word

What does it mean to say that a word *w occurs n* times in the text *t*? For example, how many times does "hi" occur in "hihihoho hi"?

```
function  noccur (w: word, t: text) :=
  if #t = 0 => 0
  :: w = t  => 1
  :: =>
    noccur(w, x) + noccur(w, y)
    where x, y: seq char, d: delimiters
       such that (t = x | [d] | y)
  fi
```

Note that `noccur("hi", "hihihoho hi")` $= 1$.

2.1.2 *k* Most Frequent Words

```
function sorted(q: seq T) :=
  for i: 2..#q (q[i-1] <= q[i]);

function rsorted(q: seq T) :=
  for i: 2..#q (q[i-1] >= q[i]);

type wornat := tuple (w: word, n: nat);
```

Consider `wnq`, a sequence of k word-number pairs, with the following properties. Its first "column" lists `k` words without duplication. Its second column (a sequence of numbers) is sorted in the non-increasing order. Further more, a word `w` is listed in `wnq` only if no word, not also listed in `wnq`, occurs more frequently in the input text than does `w`. [5]

```
function to-wornat(itx: text, k: nat) :
  value wnq : seq wornat such that (
    k = #wnq = #set(wnq.w),
    sorted(wnq.n),
    for w: word (
      w in wnq.w
      ->
      for x: word
        (x in wnq.w or noccur(x, itx) <= noccur(w, itx))
    )
  )
```

How should we define "k most frequent words"? Suppose $k = 5$, and no word occurs more than 8 times, and there are 10 distinct words occuring exactly 8 times each. Would any 5 words from the 10 be acceptable?

The above definition of `wnq` yields a subtle way out for the special case when the input text does not contain `k` distinct words. It allows, in only this case, `wnq` to contain words that do *not* occur in the input. Obviously, the frequency of such words must be 0.

5. ÔM: In ÔM, `wnq.w` yields a sequence made out of the `w`-components of the `wnq` sequence.

The design problem is essentially to construct `wnq` from the given input text `itx` and number `k`.

2.1.3 Output

Call the program by the name `cwp`. Via a command shell, it is invoked as in

```
cwp  <k-num> <input-file-name> <output-file-name>
```

The number k is given as an explicit argument, not as part of the input file being scanned, to the program. `<k-num>` above stands for the radix-10 word of the number k. We do not care to deal with `stdin`, `stdout`, nor with input output redirections.

The output of the program is a text file. The file-content of the output file must consist of exactly k lines. Each line consists of exactly two words. The second word is a radix-10 word of a non-negative number. This number is the count of how many times the first word occured in the input file content. The first words of the k lines must all be distinct.

If the input file content has less than k different words, the `cwp` is free to chose arbitrary words with zero as their counts.

Each word in the output is printed along with its frequency count on a separate line. The word is separated from the the frequency count by a fixed number, say one, blanks. No word is to be repeated. The number must not have leading zeroes.[6]

```
function maptotext(q: seq wornat) :=
  if (
    q = [] => [];
    true   =>
      q[1].w | " " | itoa(q[1].n) | "\n" | maptotext(q[2..])
  )
```

Considering the command issued as an invocation `cwp(k, fin, fout)`, we decsribe the functionality in terms of the pre- and post-conditions of the `cwp`.

```
file-content(fout) =
```

6. result type of maptotext?

```
maptotext(
  to-wornat(file-content(fin), k)
);
```

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.

```
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. Dont 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]

```
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 proesent 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]

```
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 delimters => 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 `cow` will be refined several times. Our first design D1 uses a bag of words as `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 `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 `cw` were a set of word-count pairs ordered not aplhabetically 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: word ...`, in the design we need consider only the words occuring in the input text, `itx`. On the other hand, we must examine each and every word of `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 `otx`, which stands for `file-content(fout)`.

The following design satisfies the CWP. Functions `nextlexeme` is specified in Section 2.2.2.

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

```
(
  init (
    var itx := file-content(fin);
    var bw: bag of word := {| |};
  )

  let old-word(w) == ();
  let incr-count(w) == ();
  let add-new-word(w) == (bw := bw + {| w |});

  procedure build-all-words() := as-in module D0;
)
```

[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`.

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

```
 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.

```
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.

```
module dict(
  function nextlexeme(nat) (nat, nat) ) := (
```

```
  init (
    var dwn: table wornat := {};
  )

  let old-word(w) == w in dwn.w;
  let incr-count(w) == dwn[w].n := 1 + dwn[w].n;
  let add-new-word(w) == dwn := dwn + { (w, 1) } ;

  procedure build-all-words() := as-in D1;

  procedure find-frequent-words(k: nat)
       pre (#dwn >= k)
    := var otx: text (

    value count-wnq: seq wornat := such that (
      bag(count-wnq) = bag(dwn),
      rsorted(count-wnq.n)
    );
    assert (#count-wnq >= k);

    otx := [];
    for var i: nat in {1..k} (
      let (w, n) == count-wnq[i];
      otx := otx | w | [blank] | itoa(n) | [newline]
    )
  )
)
```

The `count-wnq` is specified by describing its properties. We never "do a design" for it because in the later refinement of the overall design D2, the `count-wnq` disappears.

4.2 Design D2

D2 looks identical to D1 (Section 3.1) except we now use `dwn` instead of `bow`.

```
module D2(k: nat, fin: word, fout: word) := (

  import module lex;
  import module dict;

  init (
    var itx := file-content(fin);
    lex.init(itx);
    var dwn := dict.init(lex.nextlexeme);
    dwn.build-all-words();
    file-content(fout) := dwn.find-frequent-words(k);
  )
)
```

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 occurences 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 alpha-wnq: seq wornat`, sorted based on the alphabetic order of words.

```
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 `alpha-sorted-dict` instead of `dict`. D3 does not refine the procedure `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 `set(alpha-wnq) = dwn`. The dictionary `dwn` was allowed to contain certain words with zero counts; hence, `alpha-wnq` also will. But neither `dwn` nor `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 CWP, 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`.

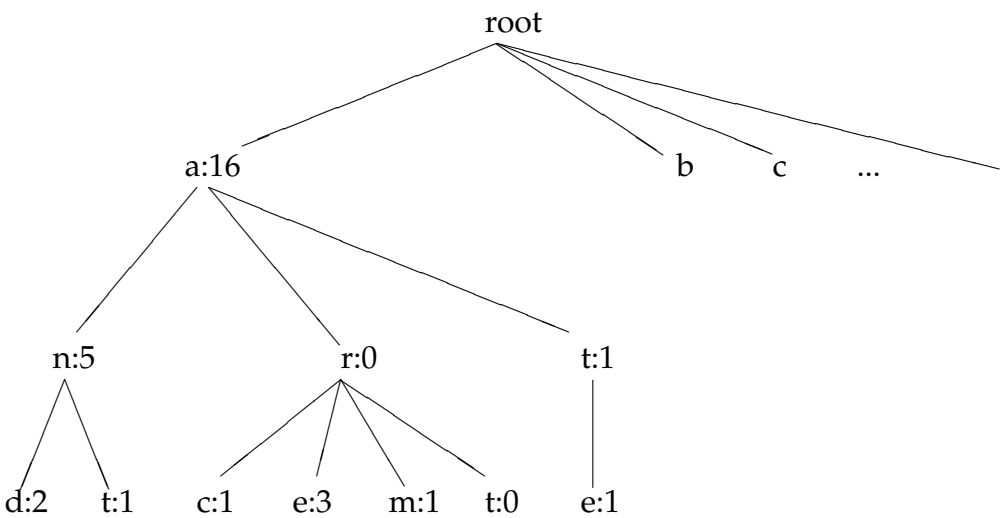

Figure 1: Example n-ary tree

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

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

The tree of Figure 1 is the result several operations: `add-new-word` with `a`, `an`, `and`, `ant`, `arc`, `are`, `arm`, `at`, `ate`; `incr-count` with `a` 15 times, with `an` 4 times, with `and` once, and so on. Note the node with `t:0` shown as a leaf in this picture. Using the operations of `add-new-word` and `incr-count` it is *not* possible to have a *leaf* node with 0 count; it is an internal node and the subtree below that node (e.g., due to an `add-new-word("article")` is not shown. However, the tree of Figure 1 is, as-is, a valid n-ary tree.

As an example insertion of a new word, let us consider `arduous`. After `old-word("arduous")` says `false`, we make a subtree by invoking `mk-ntree-ao("duous")`, which must become a subtree of node `(r:0)` between the subtrees at `c:1` and `e:3`.

6.1 Search for a Word

We `search` for a given word `w` as follows in the tree `this`. The search routine does not alter the tree in any way.

```
procedure search(w: word) :=
 var (t: ntree-ao, p:nat, i: nat) (
  t := this;
  p := 0;
  while (
    i :=  w[p+1] ## t.stq.letter;
    if (i = 0 => break);
    p := p + 1;
    if (p = #w + 1 => break);
    t :=  t.stq[i]
  )
);
```

The procedure returns a triplet value `(t, p, i)`. Its first component `t` is an n-ary tree rooted at the last node searched. The second component `p` is a natural number. It is set so that the path from the root yields `w[1..p-1]`, and either `p = #w + 1`, or the letter `w[p]` is not among the children of `t`. Thus,

`p = #w + 1` if the word is already present, and `i` is set so that the root of the i-th subtree contains the letter `w[#]`,

`p` $\leq$ `#w` and `i = 0` otherwise.

6.2 Design D4

D4 refines `old-word(w) == (w #in alpha-wnq.w > 0)` of the preceding section into `search(w).2 = #w + 1`. The initialization `var nt: ntree-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 `stq`.

```
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;
)
```

In ÔM, `q[ @ k := e ]` stands for an updated sequence `q`, where all its items at indices `i` and above are shifted to one-higher index positions, and the `i`-th item becomes `e`.

```
procedure mk-ntree-ao(w: word) pre (#w > 0) :=
 var t: ntree-ao (
  t := (w[#], 1, []);
  for var i: nat := #w - 1 downto 1 (
    t := (w[i], 0, [t])
  )
);
```

6.3 N-ary Tree to Bag of Words

For any value of type `ntree-ao` we can find a corresponding value of `alpha-wnq`. The converse, however, is false. This happens due to zero-count words. As a trivial example, the following possible value of `alpha-wnq` is unrepresentable as an `ntree-ao`: `[("are", 1)]` in contrast to `[("a", 0), ("ar", 0), ("are", 1)]`. Recall that neither `dwn` of Section 4.3 nor `alpha-wnq` of Section 5.1 have indicated specifically what the characterization of the zero-count words that they may contain is. For every word `w` with a count greater than 0 occuring in a `ntree-ao`, all the prefixes of `w` will also occur. The counts of these prefixes may or may not be zero depending on whether they have occured as independent words.

7 Tries

A trie is a binary tree that has a certain relationship to our n-ary tree. This relationship is analogous to that discussed in Section 2.3.2 of [Knuth 97]. Figure 2 is a trie made from that of Figure 1. In terms of the figures, we see that the left most edges of each node are retained, but the other edges from a node to its subtrees are replaced by "horizontal" edges from one node to its sibling.

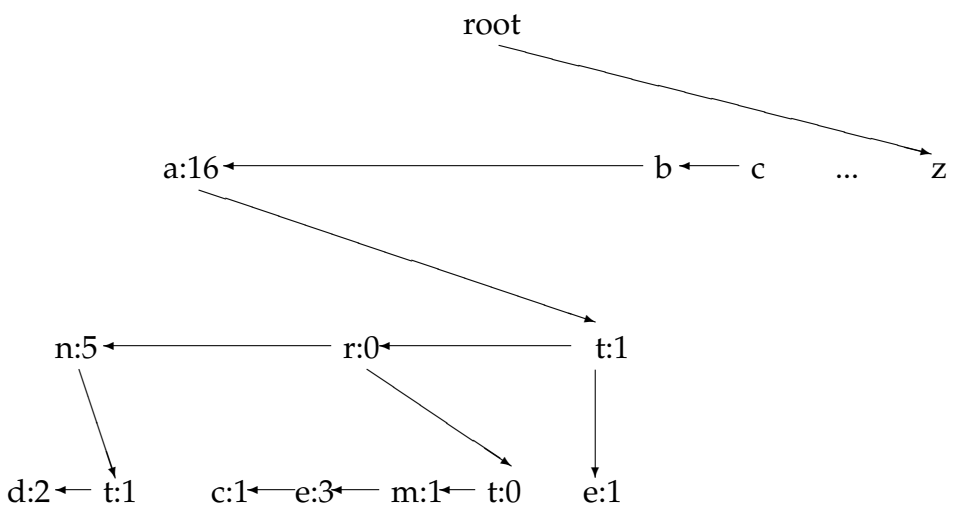

Figure 2: Example plain trie

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.

```
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
```

```
 ) such that for t: trie (
    is-finite(t.cid),
    the-children-are-ordered(t),
    parent-is-unique(t),
    req-nilid-rootid(t)
);

let cids == t.cid;              // a few abbreviations
let cnt(x) == t[x].cnt;
let nxt(x) == t[x].nxt;
let hic(x) == t[x].hic;
```

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 $\texttt{nxt}^i\texttt{(hic(u))}$. 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, $\texttt{nxt}^i\texttt{(d) = nilid}$, for some $0 \leq i \leq$ `#letter`.

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

function parent(u: cids) := if (
  u = rootid => nilid;
  u = nilid => nilid;
  else => value p: cids such that (u in childrenq(p))
);

function the-children-are-ordered() :=
  for u: cids (
    let q == childrenq(u),
    for i: 1..#q - 1 (
       ltr(q[i]) < ltr(q[i+1])
  ) );
```

7.1.3 parent-is-unique()

```
function parent-is-unique() :=
  for u, v: cellid (
    set(childrenq(u)) *  set(childrenq(v)) != {} <-> u = v
  );
```

Note that because `cids` is finite, there exists a finite i such that $\texttt{parent}^i\texttt{(u)} = \texttt{rootid}$ for all `u` except `nilid`.

7.1.4 Search for the Word

The procedure `ltr-among-children` is a refinement of `w[p+1] ## t.subs.ltr` of Section 6.1. The procedure is guaranteed to terminate because `ltr(nilid) = hdr < a`, for any `a`. Note that `x` is "behind" `y` in this routine.

```
procedure ltr-among-children(a: letter, p: cellid) :=
  var (x, y: cellid) (
  (x, y) := (p, hic(p));
  while (ltr(y) > a =>
    (x, y) := (y, nxt(y))
  )
);

procedure search(w: word) :=
  var (vi: cellid, p: nat, ui: cellid) (
  vi := t.rootid;
  p := 0;
  while  (
    let a == w[p+1];
    (var wi: cellid, ui) := ltr-among-children(a, vi);
    if (ltr(ui) = a => break);
    p := p + 1;
    assert (ltr(ui) = w[p], ui = nxt(wi));
    if (p = #w => break);
    vi := ui;
  )
);
```

7.2 Tries with Rings

We now enhance such tries into those that have circular lists of siblings. Suppose `u` is a parent whose highest child is `v` and lowest child is `w`. In the unringed tries, `hic(u) = v`, and `nxt(w) = nilid`. In `ringed-tries`, we introduce an extra cellid `h`, called a header, for each parent with children, so that `hic(u) = h, nxt(h) = v`, and the `nxt(v)` etc. remain as they were, except `nxt(w)` which was `nilid` now

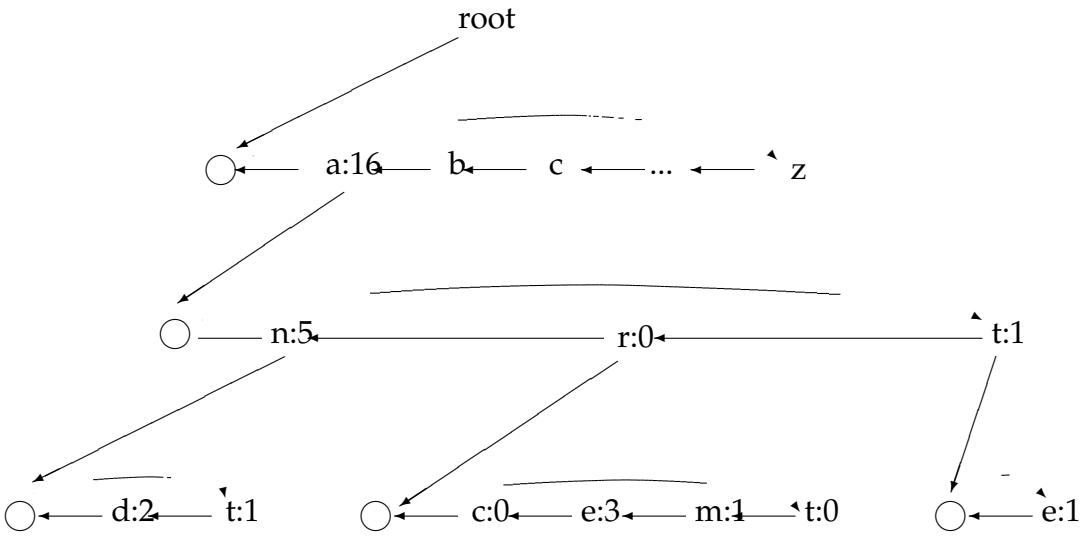

Figure 3: Example Trie with Rings ~~PPPPPPPPPPPPPPPPP TBD~~

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`.

```
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]),
    ) ) )
)
```

7.2.1 Insert New Word

As before, `add-new-word(w) == mk-ring-trie(ui, pn+1)`. This adds tuples to the global trie var `t`.

```
procedure mk-ring-trie(ui: cellid, k: nat) := (
  var pi := new-cellid();
  t += {(pi, w[k], 0, nxt(ui), nilid)};
  nxt(ui) := pi;

  for j: nat := k+1 .. #w (
    var hi := new-cellid();
    var ni := new-cellid();
    t += {(hi, hdr,  0, ni, nilid)};
    t += {(ni, w[j], 0, hi, nilid)};
    hic(pi) := hi;
    pi := ni;
  );
  cnt(pi) := 1;
);
```

7.2.2 `word-from-trie()`

It can be seen readily that a cellid corresponds to a node of the n-ary tree. The word it represents is quite simple to compute. To get the parent of any node `u`, start from `u`, go to its header `h` by traversing the `nxt` "fields", and the `hic(h)` yields the parent.

```
procedure header(u: cids) := var u (
  pre (u != rootid, u != nilid);
  while (ltr(u) != hdr => u := nxt(u));
);

let parent(x) == hic(header(x));

function word-from-trie(p: cids) :=
  if (
```

```
    p = rootid => [];
    p = nilid => [];
    else =>
       word-from-trie(parent(p))
       | [ltr(p)]
  );
```

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.

```
type trie := table (
  ...
  foq: cellid /* new/temporarily */
)
 such that  ...
);

let foq(x) == t[x].foq;

procedure insert-into-foq(p: cids) (
  var q: cellid := rootid;
  while (cnt(p) < cnt(foq(q)) => q := foq(q));
  foq(p) := foq(q);
  foq(q) := p;
);

procedure rsort-the-trie()  := var q: cellid (
  q := rootid;
  foq(q) := nilid;
  for p: (cids - {nilid, rootid})
    if (cnt(p) > 0 => insert-into-foq(p))
);

procedure find-frequent-words() := (
  pre (#dwn >= k) ;
  var q: cellid := foq(rsort-the-trie());
  otx := [];
  for var i: nat in {1..k} (
    let w == word-from-trie(q);
    let n == cnt(q);
    otx := otx | w | [blank] | itoa(n) | [newline]
    q := foq[q];
  )
)
```

7.4.2 Preparing to dispense with `foq`

The function `word-from-trie()` depends on `nxt` fields only because `header()` uses `nxt` fields. If the header can be computed in some other manner, the contents of all `nxt` fields is irrelevant to the `for`-loop of `find-frequent-words()`. This issue will be dealt with in Section 8.2.

The procedure `rsort-the-trie` of Section 7.4.1 uses the loop `for p: (cids - nilid, rootid)`. Note that the order in which different values for `p` are chosen is unspecified. So, we exercise our design freedom and start with the alphabetically last word and end after the first.

```
procedure last-word-from(p: cellid) := var p (
  while (hic(p) != nilid => (
     p := nxt(hic(p))
  )
);

let last-word() == last-word-from(rootid);
let end-of-words() == rootid;

procedure next-word(p: cellid) := var p (
  p := nxt(p);
  p := if (
    ltr(p) = hdr => hic(p);
    else => last-word-from(p);
  )
);

procedure rsort-the-trie()  := var q: cellid (
  q := rootid;
  foq(q) := nilid;
  p := last-word-from(rootid);
  while (
    if (cnt(p) > 0 => insert-into-foq(p));
    p := next-word(p);
    if (p = end-of-words() => break);
```

```
 )
);
```

As of now, `next-word(p)` does not use the `fop` fields, and `insert-....` does not use `nxt` fields. Hence, while preserving semantics, we can rewrite the above:

```
procedure rsort-the-trie()  := var q: cellid (
  q := rootid;
  foq(q) := nilid;
  p := last-word-from(rootid);
  while (
    q := next-word(p);
    if (cnt(p) > 0 => insert-into-foq(p));
    if (q = end-of-words() => break);
    p = q;
 )
);
```

This moved `next-word(p)` to a position above the `insert-into-foq(p)`. It can be seen readily that after the `nxt` field of a node `p` are used (eother via `last-word-from` or via `next-word`, it will never be needed again except in `word-from-trie()`. Thus, we can set such `nxt` fields to whatever. We let the `foq` values time-share the "residences" of `nxt` fields: that is:
We declare `foq` as an alias for `nxt` field.

7.5 Design D6

8 Hash Tries

In defining regular tries, `cellid` was simply a set of cellids whose details were left unspecified. The search and insertion speed is, on the other hand, influenced by what values these cellids are. We now superimpose these considerations on the tries.

p	*link[p]*	*ch[p]*	*sibling[p]*	*count[p]*	*Word*
0	0	hdr	26		
1	2014	1	0		a
2	1000	2	1		b
3		3	2		c
1000	2	hdr	1005		
1001					
1002					
1003					
1004					
1005	2000	5	1000		be
2000	1005	hdr	2021		
2014	1	hdr	2020		
2015	3000	15	2000		ben
2016					
2017					
2018					
2019					
2020	4000	6	2014		af
2021		20	2015		bet
3000	2015	hdr	3021		
3021	0	21	3000		bent

Figure 4: Example hash trie [Bentley 86](p 479)

8.1 Cell-Ids Refined

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

```
acn(v) - acn(u) = ltr(v) - ltr(u).

type hash-trie := ringed-trie such that (
  value trie-size := ...;
  value alpha := ((sqrt(5) - 1)/2) * trie_size;
  rootid = 0;
  nilid = trie-size + 1;
  cellid = rootid .. nilid;
  for t: hash-trie (
    for u: t.cids (
      let q == childrenq(u),
      for i: 1..#q-1 (
        q[i+1] - q[i] = ltr(q[i+1]) - ltr(q[i]) ))));
```

Note that we can now satisfy `hic(hic(rootid)) = rootid` by making `hic(rootid)` also be a `0`. Note also that the rings of different parents can be interleaved.

8.2 Word from the Hash Trie

The `header` of a node `u` is readily computed, without any traversal.

```
function hash-trie.header(u: cids) := value v: cids (
  pre (u != rootid, u != nilid);
  v := u - ltr(u)
);
```

Reading the `word-from-trie` is the same as in the preceding section but it now uses the above `header()`.

8.3 Initial Hash Trie

The following builds an initial hash trie. The `emt` and `hdr` values were defined in Section 7.1.

```
hash-trie.init := (
  var t: hash-trie;
```

```
  ltr(0) := hdr;
  cnt(0) := 0;
  hic(0) := #letter;

  for i in {1 .. #letter} (
    ltr(i) := i;
    cnt(i) := 0;
    hic(i) := nilid;
    nxt(i) := i - 1;
  );

  for i in {#letter + 1 .. trie-size} (
    ltr(i) := emt; // empty-slot
  )
)
```

8.4 Search of a Word

The procedure `search` of hash-tries is exactly the same as for tries (see Section 7.1.4). The definitions of `old-word(w)` and `incr-count(w)` remain as before.

8.5 Insertion of a Word

As we enter the words into the hash trie, the family rings grow larger and larger. As a result, at some point the interleaving of rings does not permit an insertion. To resolve such a collision, we need to relocate one of the rings.

```
procedure hash-trie.relocate-children
  (oh: cellid, nh: cellid) := (
  var r: cellid := oh;
  var d: integer:= nh - r;
  while ltr(r) != emt => (
    nxt(r + d) := nxt(r) + d;
    ltr(r + d) := ltr(r);
    cnt(r + d) := cnt(r);
```

```
    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;
```

```
  nxt(an) := nxt(pn);
  nxt(pn) := an;
)
```

8.6 Sorting the Words by Frequency

Recall that the sorting by frequency of all the words is begun only after all the input has been read, and the hash trie is fully constructed.

```
procedure hash-trie.link-p-into-sorted(p: cellid) := (
  let m == sorted(large-count);
  if (
    f < large-count => insert-into-list(p, f);
    cnt(p) >= cnt(m) => insert-into-list(p, large-count);
    else => insert-into-sib(p, m)
  );
);

procedure hash-trie.move-to-last-suffix(var p: cellid) (
  while (chi[p] != nilid => p := nxt[chi[p]]);
)

procedure hash-trie.rsort() := ( // trie-sort()
  var p, q: cellid;

  for p in 1 .. large-count (sorted[p] := nilid);
  p := nxt[rootid];
  move-to-last-suffix(p);
  while (p /= nilid => (
    q := nxt[p];
    if (count[p] != 0 => link-p-into-sorted(p));
    if (
      ltr[q] != hdr => move-to-last-suffix(p);
      else => p := hic[q]
    )
```

```
 )
)
```

The procedure `find-frequent-words` is the same as before except `fop` is replaced by `nxt`. "After `trie-sort` has done its thing, the linked lists `sorted[largecount]`, `...`, `sorted[1]` collectively contain all the words of the input file, in decreasing order of frequency. Words of equal frequency appear in alphabetic order." [Bentley 86]

8.7 Design D7

```
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;
```

```
);
```

9 Example Evaluation

This section is intended to give an objective evaluation of the experience we have had in doing the exercise reported in this paper. We are concerned with three factors in this evaluation: a retrospect of ÔM, the complexity of the example, and our stylistic use of ÔM.

9.1 Critique of Knuth's Solution

As we have mentioned earlier, the complexity of this paper as an example of a design document is due to the intricate structure of the hash trie. The hash trie has some nice properties such as storage efficiency and alphabetical orderedness of its content. Unfortunately, these properties do not come for free. McIlroy comments that "Knuth has shown us here how to program intelligently, but not wisely. ... He has fashioned a sort of industrial strength Fabergé egg — intricate, wonderfully worked, refined beyond all ordinary desires, a museum piece from the start."

We ought to describe this complex structure and the operations to access it in such a way as to exhibit the reasons why it has these properties and how they are preserved. In addition to that, we are very much concerned with precision throughout the descriptions.

However, the complexity inherent to Knuth's solution motivated us in choosing it as the appropriate material to test the suitability of ÔM to the design of real software systems.

9.2 ÔM

The following criteria are important in evaluating a software design specification lanuguage: availability of versatile high level data structures, expressibility of algorithmic descriptions and design decisions, degree of precision realizable in design sepecifications, executability of designs, and support to design methodologies and principles. These data structures together should allow a "good"

designer to compose an abstract representation of any data object necessary to express designs. In addition, a design language must offer an adequate and efficient syntax if it is going to be used by human designers. The syntax of the language must allow the designer to express designs in a natural manner, without having to cope with overwhelming syntactic details. Short hand builtin notations help avoid unnecessarily lengthy description. With respect to these criteria ÔM appears to be fairly satisfactory.

Due to its nature, ÔM can be used in various styles [Diby et al. 1988]: functional style, imperative style, sometimes logic style, and very often a mixture of the above. However, the functional and logic styles are more appropriate for high level design specifications whereas the imperative style is used for low level design specifications.

The current release is just a subset of the full language, constructs to support expression of design decisions such as those concerning resource usage and choices among alternative designs are still under investigation. In addition, as related to natural design expressibilty, the language will have some mechanism for importing notations from the problem domains of the designs. For instance, if we were to design a satellite control program, we would like to be able to express its design in terms of the vector notation that the typical physicist would use in this case. Finally, because ÔM is primarily directed towards an engineering contribution to software design, it does not yet provide any particular documentation support beyond the usual commentaries.

10 Conclusion

Our example is the result of a reverse engineering process[11]. The specifications and designs were written after studying the implementation. The material presented in this paper turned out to be longer (in terms of text size) than Knuth's implementation for instance. However, one must note that formal specifications and designs, as presented above, convey more information than does their corresponding implementations. Through the example presented above, we have demonstrated some features of ÔM and have shown that indeed the language is powerful enough to support the design complex software systems. With its con-

11. Discuss why this is so long.

structs, we have been able to precisely "specify a design solution" to the common words problem.[Diby 90]

References

[Bentley 86] J. Bentley, D. E. Knuth and M. D. McIlroy, "Programming Pearls," a column in *Communications of the ACM*, Vol. 29, No. 6, 471-483.

[Cohen et al. 86] Bernard Cohen, W.T. Harwood, and M.I. Jackson, The specification of complex systems, Addison-Wesley, 1986, ISBN: 0-201-14400-X.

[Comer 84] Douglas Comer, Operating System Design, Prentice-Hall, c1984-c1987, ISBN: 0-13-637539-1 (v. 1), ISBN: 0-13-637414-X (v. 2).

[Diby 90] Kouakau Diby, *Foundations of Hierarchical Design Methods for Software*, Ph. D. Dissertation, Wright State U, June 90.

[Fraser and Hanson 95] Christopher W. Fraser and David R. Hanson, *A retargetable C compiler : design and implementation*, Benjamin/Cummings Pub. Co. ISBN: 0-8053-1670-1.

[Hayes 93] Ian Hayes (Editor), Specification case studies, Prentice Hall, 1993, 2nd ed. ISBN: 0-13-832544-8.

[Kernighan and Plauger 76] Brian Kernighan, and Plauger, *Software Tools,* Prentice-Hall.

[Knuth 97] Donald Knuth, *The Art of Computer Programming Vol. 1: Fundamental Algorithms,* Addison-Wesley, 1997, 3rd ed, ISBN: 0-201-89683-4 (v. 1).

[Knuth 73] Donald Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley, Reading, Mass., Section 6.3.

[Knuth 84] Donald E. Knuth, "Literate Programming," *Computer Journal*, Vol. 27, No. 2, 97–111. Reprinted in Literate Programming (book), Center for the Study of Language and Information, c1992, ISBN: 0-937073-80-6.

[Knuth 86a] Donald E. Knuth, TeX: The Program, Addison-Wesley, Reading, Mass.

[Knuth 86b] Donald E. Knuth, MetaFont: The Program, Addison-Wesley, Reading, Mass. ISBN: 0-201-13438-1.

[Knuth and Levy 94] Donald E. Knuth, and Silvio Levy, The CWEB system of structured documentation : version 3.0, Addison-Wesley, c1994, ISBN: 0-201-57569-8.

[Mateti 90] Prabhaker Mateti, "ÔM: A Design Specification Language", Technical Report WSU-CS-90-07, 45 pp., Jan 1990, Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435.

[Meyer 85] Bertrand Meyer, "On Formalism in Specifications," *IEEE Software,* Jan 1985, 6–26.

[Tanenbaum 87] Andrew Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, ISBN: 0-13-637406-9.

[VDM] Vienna Development Method, http://www.ifad.dk/vdm/vdm.html

[Van Wyck 87] Christopher J. Van Wyck, "Literate Programming," a column in *Communications of the ACM*, Vol. 30, No. 7, pp. 594-599.