Mini-Indexes for Literate Programs

Donald E. Knuth

Computer Science Department, Stanford University, Stanford, CA 94305-2140 USA

Abstract. This paper describes how to implement a documentation technique that helps readers to understand large programs or collections of programs, by providing local indexes to all identifiers that are visible on every two-page spread. A detailed example is given for a program that finds all Hamiltonian circuits in an undirected graph.

Keywords: Literate programming, WEB, CWEB, C++, CTWILL, TEX, indexes, hypertext, Hamiltonian circuits

1. Introduction

Users of systems like WEB [2], which provide support for structured documentation and literate programming [5], automatically get a printed index at the end of their programs, showing where each identifier is defined and used. Such indexes can be extremely helpful, but they can also be cumbersome, especially when the program is long. An extreme example is provided by the listing of TEX [3], where the index contains 32 pages of detailed entries in small print.

Readers of [3] can still find their way around the program quickly, however, because

> . . . the right-hand pages of this book contain mini-indexes that will make it unnecessary for you to look at the big index very often. Every identifier that is used somewhere on a pair of facing pages is listed in a footnote on the right-hand page, unless it is explicitly defined or declared somewhere on the left-hand or right-hand page you are reading. These footnote entries tell you whether the identifier is a procedure or a macro or a boolean, etc. [3]

A similar idea is sometimes used in editions of literary texts for foreign language students, where mini-dictionaries of unusual words appear on each page [10];

this saves the student from spending a lot of time searching big dictionaries.

The idea of mini-indexes was first suggested to the author by Joe Weening, who prepared a brief mockup of what he thought might be possible [11]. His proposal was immediately appealing, so the author decided to implement it in a personal program called TWILL—a name suggested by the fact that it was a two-pass variant of the standard program called WEAVE. TWILL was used in September, 1985, to produce [3] and a companion book [4].

The original WEB system was a combination of TEX and Pascal. But the author's favorite programming language nowadays is CWEB [6], which combines TEX with C. (In fact, CWEB version 3.0 is fully compatible with C++, although the author usually restricts himself to a personal subset that might be called C--.) One of the advantages of CWEB is that it supports collections of small program modules and libraries that can be combined in many ways. A single CWEB source file foo.w can generate several output files in addition to the C program foo.c; for example, foo.w might generate a header file foo.h for use by other modules that will be loaded with the object code foo.o, and it might generate a test program testfoo.c that helps verify portability.

CWEB was used to create the Stanford GraphBase, a collection of about three dozen public-domain programs useful for the study of combinatorial algorithms [9]. These programs have recently been published in book form, again with mini-indexes [7]. The mini-indexes in this case were prepared with CTWILL [8], a two-pass variant of CWEAVE.

The purpose of this paper is to explain the operations of TWILL and of its descendant, CTWILL. The concepts are easiest to understand when they are related to a detailed example, so a complete CWEB program has been prepared for illustrative purposes. Section 2 of this paper explains the example program; Sections 3 and 4 explain how CTWILL and TEX process it; and Section 5 contains concluding comments.

2. An example

The CWEB program for which sample mini-indexes have been prepared especially for this paper is called HAM. It enumerates all Hamiltonian circuits of a graph, that is, all undirected cycles that include each vertex exactly once. For example, the program can determine that there are exactly 9862 knight's tours on a 6×6 chessboard, ignoring symmetries of the board, in about 2.3 seconds on a SPARCstation 2. Since HAM may be interesting in its own right, it is presented in its entirety as sort of a "sideshow" in the right-hand columns of the pages of this article and on the final (left-hand) page.

Please take a quick look at HAM now, before reading further. The program appears in five columns, each of which will be called a *spread* because it is analogous to the two-page spreads in [3] and [7]. This arrangement gives us five mini-indexes to look at instead of just two, so it makes HAM a decent example in spite of its relatively small size. A shorter program wouldn't need much of an index at all; a longer program would take too long to read.

HAM is intended for use with the library of routines that comes with the Stanford GraphBase, so §1 of the program tells the C preprocessor to include header files `gb_graph.h` and `gb_save.h`. These header files define the external functions and data types needed from the GraphBase library.

A brief introduction to GraphBase data structures will suffice for the interested reader to understand the full details of HAM. A graph is represented by combining three kinds of **struct** records called **Graph**, **Vertex**, and **Arc**. If v points to a **Vertex** record, $v{\rightarrow}name$ is a string that names the vertex represented by v, and $v{\rightarrow}arcs$ points to the representation of the first arc emanating from that vertex.[1] If a points to an **Arc** record that represents an arc from some vertex v to another vertex u, then $a{\rightarrow}tip$ points to the **Vertex** record that represents u; also $a{\rightarrow}next$ points to the representation of the next arc from v, or $a{\rightarrow}next = \Lambda$ (i.e., `NULL`) if a is the last arc from v. Thus the following loop will print the names of all vertices adjacent to v:

```
for (a = v→arcs; a; a = a→next)
  printf("%s\n", a→tip→name);
```

An undirected edge between vertices u and v is represented by two arcs, one from u to v and one from v to u. Finally, if g points to a **Graph** record, then $g{\rightarrow}n$ is the number of vertices in the associated graph, and the **Vertex** records representing those vertices are in locations $g{\rightarrow}vertices + k$, for $0 \le k < g{\rightarrow}n$.

[1] '$v{\rightarrow}name$' is actually typed '`v->name`' in a C or CWEB program; typographic sugar makes the program easier to read in print.

A **Vertex** record also contains "utility fields" that can be exploited in different ways by different algorithms. The actual C declarations of these fields, quoted from §8 and §9 of the program GB_GRAPH [7], are as follows:

```
typedef union {
  struct vertex_struct *V;
    /* pointer to Vertex */
  struct arc_struct *A;
    /* pointer to Arc */
  struct graph_struct *G;
    /* pointer to Graph */
  char *S;
    /* pointer to string */
  long I;
    /* integer */
} util;
typedef struct vertex_struct {
  struct arc_struct *arcs;
    /* linked list of arcs out of this vertex */
  char *name;
    /* string identifying this vertex symbolically */
  util u, v, w, x, y, z;
    /* multipurpose fields */
} Vertex;
```

Program HAM uses the first four utility fields in order to do its word efficiently. Field u, for example, is treated as a **long** integer representing the degree of the vertex. Notice the definition of deg as a macro in §2; this makes it possible to refer to the degree of v as $v\neg deg$ instead of the more cryptic '$v\neg u.I$' actually seen by the C compiler. Similar macros for utility fields v, w, and x can be found in §4 and §6.

The first mini-index of HAM, which can be seen below §2 in the first column of the program, gives cross-references to all identifiers that appear in §1 or §2 but are not defined there. For example, $restore\_graph$ is mentioned in one of the comments of §1; the mini-index tells us that it is a function, that it returns a value of type **Graph** $*$, and that it is defined in §4 of another CWEB program called GB_SAVE. The mini-index also mentions that **Vertex** and $arcs$ are defined in §9 of GB_GRAPH (from which we quoted the relevant definitions above), and that fields $next$ and tip of **Arc** records are defined in GB_GRAPH §10, etc.

One subtlety of this first mini-index is the entry for u, which tells us that u is a utility field defined in GB_GRAPH §9. The identifier u actually appears twice in §2, once in the definition of deg and once as a variable of type **Vertex** $*$. The mini-index refers only to the former, because the latter usage is defined in §2. Mini-indexes don't mention identifiers defined within their own spread.

The second mini-index, below §5 of HAM, is similar to the first. Notice that it contains two separate entries for v, because the identifier v is used in two senses—both as a utility field (in the definition of $taken$) and as a variable (elsewhere). The C compiler will understand how to deal with constructions like '$v\neg v.I$ = 0', which the C preprocessor expands from '$v\neg taken$ = 0', but human readers are spared such trouble.

Notice the entry for deg in this second mini-index: It uses an equals sign instead of a colon, indicating that deg is a macro rather than a variable. A similar notation was used in the first mini-index for cross-references to typedef'd identifiers like **Vertex**. See also the entry for not_taken in the fourth mini-index: Here 'not_taken = macro ()' indicates that not_taken is a macro with arguments.

3. The operation of CTWILL

It would be nice to report that the program CTWILL produces the mini-indexes for HAM in a completely automatic fashion, just as CWEAVE automatically produces ordinary indexes. But that would be a lie. The truth is that CTWILL only does about 99% of the work automatically; the user has to help it with the hard parts.

Why is this so? Well, in the first place, CTWILL isn't smart enough to figure out that the 'u' in the definition of deg in §2 is not the same as the 'u' declared to be **register Vertex** $*$ in that same section. Indeed, a high degree of artificial intelligence would be required before CTWILL could deduce that.

In the second place, CTWILL has no idea what mini-index entry to make for the identifier k that appears in §6. No variable k is declared anywhere! Indeed, users who write comments involving expressions like '$f(x)$' might or might not be referring to identifiers f and/or x in their programs; they must tell CTWILL when they are making "throwaway" references that should not be indexed. CWEAVE doesn't have this problem because it indexes only the definitions, not the uses, of single-letter identifiers.

In the third place, CTWILL will not recognize automatically that the $vert$ parameter in the definition of not_taken, §4, has no connection with the $vert$ macro defined in §6.

A fourth complication, which does not arise in HAM but does occur in [3] and [7], is that sections of a WEB or CWEB program can be used more than once. Therefore a single identifier might actually refer to several different variables simultaneously. (See, for example, §652 in [3].)

In general, when an identifier is defined or declared exactly once, and used only in connection with its

unique definition, CTWILL will have no problems with it. But when an identifier has more than one implicit or explicit definition, CTWILL can only guess which definition was meant. Some identifiers—especially single-letter ones like x and y—are too useful to be confined to a single significance throughout a large collection of programs. Therefore CTWILL was designed to let users provide hints easily when choices need to be made.

The most important aspect of this design was to make CTWILL's default actions easily predictable. The more "intelligence" we try to build into a system, the harder it is for us to control it. Therefore CTWILL has very simple rules for deciding what to put in mini-indexes.

Each identifier has a unique *current meaning*, which consists of three parts: its type, and the program name and section number where it was defined. At the beginning of a run, CTWILL reads a number of files that define the initial current meanings. Then, whenever CTWILL sees a C construction that implies a change of meaning—a macro definition, a variable declaration, a typedef, a function declaration, or the appearance of a label followed by a colon—it assigns a new current meaning as specified by the semantics of C. For example, when CTWILL sees '**Graph** $*g$' in §2 of HAM, it changes the current meaning of g to '**Graph** $*$, §2'. These changes occur in the order of the CWEB source file, not in the "tangled" order that is actually presented to the C compiler. Therefore CTWILL makes no attempt to nest definitions according to block structure; everything it does is purely sequential. A variable declared in §5 and §10 will be assumed to have the meaning of §5 in §6, §7, §8, and §9.

Whenever CTWILL changes the current meaning of a variable, it outputs a record of that current meaning to an auxiliary file. For the CWEB program `ham.w`, this auxiliary file is called `ham.aux`. The first few entries of `ham.aux` are

```
@$deg {ham}2 =macro@>
@$argc {ham}2 \&{int}@>
@$argv {ham}2 \&{char} ${*}[\,]$@>
```

and the last entry is

```
@$d {ham}8 \&{register} \&{int}@>
```

In general these entries have the form

$$\texttt{@\$}\mathit{ident}\ \texttt{\{}\mathit{name}\texttt{\}}\mathit{nn}\ \ \mathit{type}\texttt{@>}$$

where $ident$ is an identifier, $name$ and nn are the program name and section number where $ident$ is defined, and $type$ is a string of TEX commands to indicate its type. In place of '{$name$}nn' the entry might have the

form "string" instead; then the program name and section number are replaced by the string. (This mechanism leads, for example, to the appearance of <stdio.h> in HAM's mini-index entries for *printf*.) Sometimes the *type* field says '\zip'. This situation doesn't arise in HAM, nor does it arise very often in [7]; but it occurs, for example, when a preprocessor macro name has been defined externally as in a Makefile, or when a type is very complicated, like FILE in <stdio.h>. In such cases the mini-index will simply say 'FILE, <stdio.h>', with no colon or equals sign.

The user can explicitly change the current meaning by specifying @$*ident* {*name*}*nn type*@> anywhere in a CWEB program. This means that CTWILL's default mechanism is easily overridden.

When CTWILL starts processing a program foo.w, it looks first for a file named foo.aux that might have been produced on a previous run. If foo.aux is present, it is read in, and the @$...@> commands of foo.aux give current meanings to all identifiers defined in foo.w. Therefore CTWILL is able to know the meaning of an identifier before that identifier has been declared—assuming that CTWILL has been run successfully on foo.w at least once before, and assuming that the final definition of the identifier is the one intended at the beginning of the program.

CTWILL also looks for another auxiliary file called foo.bux. This one is not overwritten on each run, so it can be modified by the user. The purpose of foo.bux is to give initial meanings to identifiers that are not defined in foo.aux. For example, ham.bux is a file containing the two lines

```
@i gb_graph.hux
@i gb_save.hux
```

which tell CTWILL to input the files gb_graph.hux and gb_save.hux. The latter files contain definitions of identifiers that appear in the header files gb_graph.h and gb_save.h, which HAM includes in §1. For example, one of the lines of gb_graph.hux is

```
@$Vertex {GB\_GRAPH}9 =\&{struct}@>
```

This line appears also in gb_graph.aux; it was copied by hand, using a text editor, into gb_graph.hux, because **Vertex** is one of the identifiers defined in gb_graph.h.

CTWILL also reads a file called system.bux, if it is present; that file contains global information that is always assumed to be in the background as part of the current environment. One of the lines in system.bux is, for example,

```
@$printf "<stdio.h>" \&{int} (\,)@>
```

After system.bux, ham.aux, and ham.bux have been input, CTWILL will know initial current meanings of almost all identifiers that appear in HAM. The

only exception is k, found in §6; its current meaning is `\uninitialized`, and if the user does not take corrective action its mini-index entry will come out as

k: ???, §0.

Notice that d is declared in §4 of HAM and also in §8. Both of these declarations produce entries in `ham.aux`. Since `CTWILL` reads `ham.aux` before looking at the source file `ham.w`, and since `ham.aux` is read sequentially, the current meaning of d will refer to §8 at the beginning of `ham.w`. This causes no problem, because d is never used in HAM except in the sections where it is declared, hence it never appears in a mini-index.

When `CTWILL` processes each section of a program, it makes a list of all identifiers used in that section, except for reserved words. At the end of the section, it mini-outputs (that is, it outputs to the mini-index) the current meaning of each identifier on the list, unless that current meaning refers to the current section of the program, or unless the user intervenes.

The user has two ways to change the mini-outputs, either by suppressing the default entries or by inserting replacement entries. First, the explicit command

`@-`*ident*`@>`

tells `CTWILL` not to produce the standard mini-output for *ident* in the current section. Second, the user can specify one or more *temporary meanings* for an identifier, all of which will be mini-output at the end of the section. Temporary meanings do not affect an identifier's current meaning. Whenever at least one temporary meaning is mini-output, the current meaning will be suppressed just as if the `@-...@>` command had been given. Temporary meanings are specified by means of the operation `@%`, which toggles a state switch affecting the `@$...@>` command: At the beginning of a section, the switch is in "permanent" state, and `@$...@>` will change an identifier's current meaning as described earlier. Each occurrence of `@%` changes the state from "permanent" to "temporary" or back again; in "temporary" state the `@$...@>` command specifies a temporary meaning that will be mini-output with no effect on the identifier's permanent (current) meaning.

Examples of these conventions will be given momentarily, but first we should note one further interaction between `CTWILL`'s `@-` and `@$` commands: If `CTWILL` would normally assign a new current meaning to *ident* because of the semantics of C, and if the command `@-`*ident*`@>` has already appeared in the current section, `CTWILL` will not override the present meaning, but `CTWILL` will output the present meaning to the `.aux` file. In particular, the user may have specified the present meaning with `@$`*ident*`...@>`; this allows user control over what gets into the `.aux` file.

For example, here is a complete list of all commands inserted by the author in order to correct or enhance CTWILL's default mini-indexes for HAM:

- At the beginning of §2,

```
@-deg@>
@$deg {ham}2 =\|u.\|I@>
@%@$u {GB\_GRAPH}9 \&{util}@>
```

  to make the definition of deg read '$u.I$' instead of just 'macro' and to make the mini-index refer to u as a utility field.
- At the beginning of §4,

```
@-taken@> @-vert@>
@$taken {ham}4 =\|v.\|I@>
@%@$v {GB\_GRAPH}9 \&{util}@>
 @$v {ham}2 \&{register} \&{Vertex} $*$@>
```

  for similar reasons, and to suppress indexing of $vert$. Here the mini-index gets two "temporary" meanings for v, one of which happens to coincide with its permanent meaning.
- At the beginning of §6,

```
@-k@> @-t@> @-vert@> @-ark@>
@$vert {ham}6 =\|w.\|V@>
@$ark {ham}6 =\|x.\|A@>
@%@$w {GB\_GRAPH}9 \&{util}@>
  @$x {GB\_GRAPH}9 \&{util}@>
```

  for similar reasons. That's all.

These commands were not inserted into the program file `ham.w`; they were put into another file called `ham.ch` and introduced via CWEB's "change file" feature [6]. Change files make it easy to modify the effective contents of a master file without tampering with that file directly.

4. Processing by TEX

CTWILL writes a TEX file that includes mini-output at the end of each section. For example, the mini-output after §10 of HAM is

```
\]{GB\_GRAPH}10 \\{next} \&{Arc} $*$
\[7 \\{advance} label
\[6 \\{ark} =\|x.\|A
\[2 \|{t} \&{register} \&{Vertex} $*$
\[4 \\{not\_taken} =macro (\,)
\]{GB\_GRAPH}10 \\{tip} \&{Vertex} $*$
\[2 \|{v} \&{register} \&{Vertex} $*$
\[2 \|{a} \&{register} \&{Arc} $*$
```

Here `\[` introduces an internal reference to another section of HAM; `\]` introduces an external reference to some other program; `\\` typesets an identifier in text italics; `\|` typesets an identifier in math italics; `\&` typesets in boldface.

A special debugging mode is available in which TEX will simply typeset all the mini-output at the end of each section, instead of making actual mini-indexes. This makes it easy for users to check that `CTWILL` is in fact producing the information they really want. Notice that mistakes in `CTWILL`'s output need not necessarily lead to mistakes in mini-indexes; for example, a spurious reference in §6 to an identifier defined in §5 will not appear in a mini-index for a spread that includes §5. It is best to make sure that `CTWILL`'s output is correct before looking at actual mini-indexes. Then unpleasant surprises won't occur when sections of the program are moved from one spread to another.

When TEX is finally asked to typeset the real mini-indexes, however, it has plenty of work to do. That's when the fun begins. TEX's main task, after formatting the commentary and C code of each section, is to figure out whether the current section fits into the current spread, and (if it does) to update the mini-index by merging together all entries for that spread.

Consider, for example, what happens when TEX typesets §10 of HAM. This spread begins with §8, and TEX has already determined that §8 and §9 will fit together in a single column. After typesetting the body of §10, TEX looks at the mini-index entries. If any of them refer to §8 or §9, TEX will tentatively ignore them, because those sections are already part of the current spread. (In this case that situation doesn't arise; but when TEX processed §7, it did suppress entries for $vert$ and ark, since they referred to §6.) TEX also tentatively discards mini-index entries that match other entries already scheduled for the current spread. (In this case, everything is discarded except the entries for $advance$ and ark; the others—$next$, t, not_taken, v, and a—are duplicates of entries in the mini-output of §8 or §9.) Finally, TEX tentatively discards previously scheduled entries that refer to the current section. (In this case nothing happens, because no entries from §8 or §9 refer to §10.)

After this calculation, TEX knows the number n of mini-index entries that would be needed if §10 were to join the spread with §8 and §9. TEX divides n by the number of columns in the mini-index (here 2, but 3 in [3] and [7]), multiplies by the distance between mini-baselines (here 9 points), and adds the result to the total height of the typeset text for the current spread (here the height of §8 + §9 + §10). With a few minor refinements for spacing between sections and for the ruled line that separates the mini-index from the rest of the text, TEX

is able to estimate the total space requirement. In our example, everything fits in a single column, so TEX appends §10 to the spread containing §8 and §9. Then, after §11 has been processed in the same fashion, TEX sees that there isn't room for §§8–11 all together; so it decides to begin a new spread with §11.

The processing just described is not built in to TEX, of course. It is all under the control of a set of macros called `ctwimac.tex` [8]. The first thing `CTWILL` tells TEX is to input those macros.

TEX was designed for typesetting, not for programming; so it is at best "weird" when considered as a programming language. But the job of mini-indexing does turn out to be programmable. The full details of `ctwimac` are too complex to exhibit here, but TEX hackers will appreciate some of the less obvious ideas that are used. (Non-TEXnicians, please skip the rest of this long paragraph.) TEX reads the mini-outputs of `CTWILL` twice, with different definitions of `\[` and `\]` each time. Suppose we are processing section s, and suppose that the current spread begins with section r. Then TEX's token registers 200, 201, ..., 219 contain all mini-index entries from sections r, $r+1$, ..., $s-1$ for identifiers defined respectively in sections r, $r+1$, ..., $r+19$ of the `CWEB` program. (We need not keep separate tables for more than 20 consecutive sections starting with the base r of the current spread, because no spread can contain more than 20 sections.) Token register 199 contains, similarly, entries that refer to sections preceding r, and token register 220 contains entries that refer to sections $r+20$ and higher. Token register 221 contains entries for identifiers defined in other programs. Count register k contains the number of entries in token register k, for $199 \le k \le 221$. When count register k equals j, the actual content of token register k is a sequence of $2j$ tokens,

$$\texttt{\lmda\cs}_1\texttt{\lmda\cs}_2 \ldots \texttt{\lmda\cs}_j$$

where each $\texttt{\cs}_i$ is a control sequence defined via `\csname...\endcsname` that uniquely characterizes a mini-index entry. TEX can tell if a new mini-index entry agrees with another already in the current spread by simply testing if the corresponding control sequence is defined. The replacement text for $\texttt{\cs}_i$ is the associated mini-index entry, while the definition of `\lmda` is

```
\def\lmda#1{#1\global\let#1\relax}
```

Therefore when TEX "executes" the contents of a token register, it typesets all the associated mini-index entries and undefines all the associated control sequences. Alternatively, we can say

```
\def\lmda#1{\global\let#1\relax}
```

if we merely want to erase all entries represented in a token register. At the end of a spread containing p sections, we generate the mini-index by executing token registers 199 and $200+p$ thru 221 using the former definition of `\lmda`, and we also execute token registers 200 thru $200+p-1$ using the latter definition. Everything works like magic.

A bug in the original TEX macros for `TWILL` led to an embarrassing error in the first (1986) printings of [3] and [4]: Control sequences in token registers corresponding to sections of the current spread were not erased; in other words, the contents of those token registers were simply discarded, not executed with the second definition of `\lmda`. The effect was to make TEX think that certain control sequences were still defined, hence the macros would think that the mini-index entries were still present; such entries were therefore omitted by mistake. Only about 3% of the entries were actually affected, so this error was not outrageous enough to be noticed until after the books were printed and people started to read them. The only bright spot in this part of the story was the fact that it proved how effective mini-indexes are: The missing entries were sorely missed, because their presence would have been really helpful.

The longest-fit method by which `CTWILL`'s TEX macros allocate sections to pages tends to minimize the total number of pages, but this is not guaranteed. For example, it's possible to imagine unusual scenarios in which sections §100 and §101, say, do not fit on a single spread, while the three sections §100, §101, §102 actually do fit. This might happen if §100 and §101 have lots of references to variables declared in §102. Similarly, we might be able to fit §100 with §101 if §99 had been held over from the previous spread. But such situations are extremely unlikely, and there is no reason to worry about them. The one-spread-at-a-time strategy adopted by `ctwimac` is optimum, spacewise, for all practical purposes.

On the other hand, experience shows that unfortunate page breaks between spreads do sometimes occur unless the user does a bit more fine tuning. For example, suppose the text of §7 in HAM had been one line longer. Then §7 would not have fit with §6, and we would have been left with a spread containing just tiny little §6 and lots of wasted white space. It would look awful. And in fact, that's the reason the three statements

$$t{\rightarrow}ark = \Lambda;\ \ v = y;\ \ \textbf{goto}\ advance;$$

now appear on a single line of the program instead of on three separate lines: A bad break between spreads was avoided by manually grouping those statements, using `CWEB`'s `@+` command.

One further problem needs to be addressed—the mini-indexes must be sorted alphabetically. TEX is essential for determining the breaks between spreads (and

consequently for determining the actual contents of the mini-indexes), but TEX is not a good vehicle for sorting. The solution to this problem is to run the output of CTWILL twice through TEX, interposing a sorting program between the two runs. When TEX processes ham.tex, the macros of ctwimac tell it to look first for a file called ham.sref. If no such file is present, a file called ham.ref will be written, containing all the (unsorted) mini-index entries for each spread. TEX will also typeset the pages as usual, with all mini-indexes in their proper places but unsorted; the user can therefore make adjustments to fix bad page breaks, if necessary. Once the page breaks are satisfactory, a separate program called REFSORT is invoked; REFSORT converts ham.ref into a sorted version, ham.sref. Then when TEX sees ham.sref, it can use the sorted data to make the glorious final copy.

For example, the file ham.ref looks like this:

```
!1
+ \]{GB\_SAVE}4 \\{restore\_graph} \&{Graph}
          $*(\,)$
+ \]{GB\_GRAPH}9 \|{u} \&{util}
  ⋮
+ \]{GB\_GRAPH}8 \|{I} \&{long}
!2
  ⋮
+ \]"<stdio.h>" \\{printf} \&{int} (\,)
```

And the file ham.sref looks like this:

```
\]{GB\_GRAPH}10 \&{Arc} =\&{struct}
  ⋮
\]{GB\_GRAPH}9 \|{u} \&{util}
\]{GB\_GRAPH}9 \&{Vertex} =\&{struct}
\donewithpage1
\[2 \|{a} \&{register} \&{Arc} $*$
  ⋮
\]{GB\_GRAPH}20 \\{vertices} \&{Vertex} $*$
\donewithpage5
```

Each file contains one line for each mini-index entry and one line to mark the beginning (in ham.ref) or end (in ham.sref) of each spread.

5. Conclusions

Although CTWILL is not fully automatic, it dramatically improves the readability of large collections of programs. Therefore an author who has spent a year writing programs for publication won't mind spending an additional week improving the indexes. Indeed, a

little extra time spent on indexing generally leads to significant improvements in the text of any book that is being indexed by its author, who has a chance to see the book in a new light.

Some manual intervention is unavoidable, because a computer cannot know the proper reference for every identifier that appears in program comments. But experience with CTWILL's change file mechanism indicates that correct mini-indexes for large and complex programs can be obtained at the rate of about 100 book pages per day. For example, the construction of change files for the 460 pages of programs in [7] took 5 days, during which time CTWILL was itself being debugged and refined.

Mini-indexes are wonderful additions to printed books, but we can expect hypertext-like objects to replace books in the long run. It's easy to imagine a system for viewing CWEB programs in which you can find the meaning of any identifier just by clicking on it. Future systems will perhaps present "fish-eye" views of programs, allowing easy navigation through complicated webs of code. (See [1] for some steps in that direction.)

Such future systems will, however, confront the same issues that are faced by CTWILL as it constructs mini-indexes today. An author who wants to create useful program hypertexts for others to read will want to give hints about the significance of identifiers whose roles are impossible or difficult to deduce mechanically. Some of the lessons taught by CTWILL will therefore most likely be relevant to everyone who tries to design literate programming systems that replace books as we now know them.

References

1. Brown M, Czejdo B (1990) A hypertext for literate programming. In: Lecture Notes in Computer Science, Vol. 468, 250–259
2. Knuth DE (1984) Literate programming. In: The Computer Journal, Vol. 27, No. 2, 97–111
3. Knuth DE (1986) Computers & Typesetting, Vol. B, TEX: The Program. Addison-Wesley, Reading, Massachusetts
4. Knuth DE (1986) Computers & Typesetting, Vol. D, METAFONT: The Program. Addison-Wesley, Reading, Massachusetts
5. Knuth DE (1992) Literate Programming. CSLI Lecture Notes, No. 27, distributed by the University of Chicago Press, Chicago
6. Knuth DE, Levy S (1990) CWEB User Manual: The CWEB System of Structured Documentation. Computer Science Department Report STAN-CS-90-1336, Stanford University, Stanford, California; revised version available on the Internet via

anonymous ftp from `labrea.stanford.edu` in file `~ftp/pub/cweb/cwebman.tex`
7. Knuth DE (1993) The Stanford GraphBase: A Platform for Combinatorial Computing. ACM Press, New York
8. Knuth DE (1993) `CTWILL`. Available via anonymous ftp from `labrea.stanford.edu` in directory `~ftp/pub/ctwill`
9. Stanford University Computer Science Department (1993) The Stanford GraphBase. Available via anonymous ftp from `labrea.stanford.edu` in directory `~ftp/pub/sgb`
10. Pharr C (1930) Virgil's Æneid, Books I–VI. D. C. Heath, Boston
11. Weening JS (1983) Personal communication. Preserved in the archives of the TEX project in Stanford University Library's Department of Special Collections, SC 97, series II, box 18, folder 7.6