Contents

1 INTRODUCTION **2**
- 1.1 Motivation . . . 2
- 1.2 The Pipeline . . . 3
- 1.3 Comment . . . 4

2 HANDLING REFERENCES TO CODE FRAGMENTS: makeindex.tex **5**
- 2.1 The Input . . . 5
- 2.2 The Output . . . 6
- 2.3 The Input Records . . . 6
- 2.4 Writing the Output Records . . . 8

3 HANDLING REFERENCES TO IDENTIFIERS: makeindex.tex **10**
- 3.1 The Input . . . 10
- 3.2 The Output . . . 10
- 3.3 The Input Records . . . 10

4 FINDING REFERENCES TO IDENTIFIERS: getids **13**
- 4.1 Information Base . . . 13
- 4.2 The Output of getids . . . 13
- 4.3 The Lex File . . . 14

5 THE FILE Ccode.sty **16**
- 5.1 Boxed Code Fragments . . . 16
- 5.2 Trace Sections . . . 17
- 5.3 Cross References into Code Fragments . . . 17
- 5.4 Import Index . . . 18

A CODE FILES **20**
- A.1 puzzle.w . . . 20
- A.2 puzzle.tex . . . 23
- A.3 puzzle1.tex . . . 27
- A.4 puzzle2.tex . . . 28

Chapter 1

INTRODUCTION

1.1 Motivation

See the next note that I posted in 'comp.programming.lit' following an earlier posting of Joachim Schrod.

```
<>
<> This file ... presents the *WEB systems and tools available, in
<> an alphabetic order, and names the directory where you can find it.
<> .......
<>  ProTex       independent         [really a LitProg system?]
                                     ^^^^^^^^^^^^^^^^^^^^^^^^^

Definitely yes.

From a user point of view, ProTeX is an integral part of TeX. It adds
to TeX the ability to process code side by side with prose. Like in
other xxWEB systems, the code can be fragmented and reordered.

Moreover, ProTex like TeX tries to offer a style free environment, yet
be flexible enough to adjust itself to different imported styles.

ProTeX does very little to integrate automated indexing and cross
referencing capabilities into TeX.  However, it can be easily
augmented with preprocessors that take care of these and other types of
reverse engineering activities.  (In this regard, it should be noticed
that many of the xxWEB systems are in essence preprocessors of TeX.)

To illustrate the above claims, I placed in the public domain two
literate programs 'puzzle1' and 'puzzle2' in ProTeX.  The first is a
very basic program with a CWEB-oriented style but without indexing and
cross referencing.  The second is little fancier in appearance and it
includes an index and cross references.

The source programs differ only in the style files that they employ.
Otherwise, they are the CWEB 'puzzle' program of Lee Wittenberg with
minor adjustments (and Lee's permission to reuse his program).

The above programs, and a third one producing the utilities for
C programs (used in 'puzzle2'), can be retrieved by anonymous ftp
```

```
from ‘ftp.cis.ohio-state.edu : pub/tex/osu/gurari/puz’.

--eitan (gurari@cis.ohio-state.edu)
```

I am thankful to Joachim for constructively challenging my work, and to Lee for permitting me to use his program.

1.2 The Pipeline

ProTeX is used twice.

- To directly and indirectly generate the utilities for handling cross referencing and indexing in C programs.

```
tex  utilities.tex
lex  lexids.l
cc  -o getids lex.yy.c -ll
```

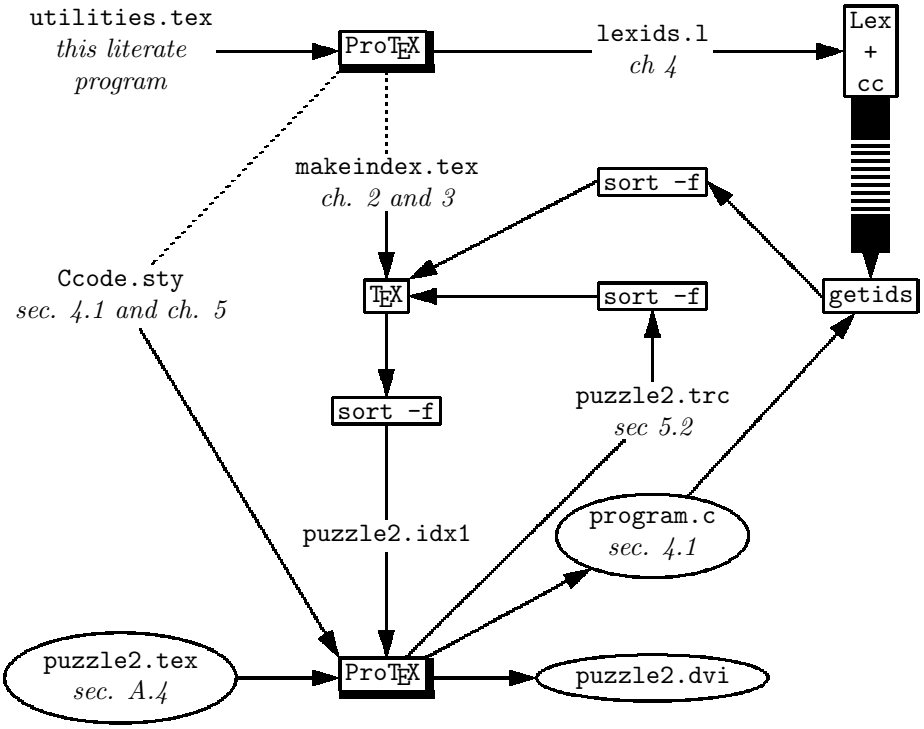
  The utilities are as follows

  - `getids`. A C program to extract the references to identifiers that consist of more than one character (references to code fragments are traced directly by ProTeX into ‘`jobname.trc`’).
  - ‘`makeindex.tex`’. A TEX file to arrange together different references to identifiers and code fragments.
  - ‘`Ccode.sty`’. A TEX file to import cross references and indexes, and to box code fragments.

- To compile the source program (`puzzle2` in this case) into a literate program.

```
tex      puzzle2.tex
sort -f  puzzle2.trc -o temp.idx1
getids < program.c    > temp.idx2
sort -f  temp.idx2   -o temp.idx2
tex      makeindex.tex
sort -f  temp.idx3   -o puzzle2.idx1
```

  The index and cross references are one generation old.



1.3 Comment

This is not a well documented program, and I currently have no time and/or motivation to polish it. It is presented with the hope that it might still provide some additional insight into the nature of ProTeX. It is also not a complete program, in the sense that it takes some short cuts because `puzzle2` is its only user. Making general C (or Pascal, TeX, ...) utilities from this program shouldn't require too much work.

As a side comment, most of the literate programs that I write are not better than this one. Yet, I find them very useful for my personal needs.

Chapter 2

HANDLING REFERENCES TO CODE FRAGMENTS: makeindex.tex

The utility '`makeindex.tex`' is a TEX program. It extracts the references to the code fragments from the trace file that ProTeX generates during the compilation of the source program.

2.1 The Input

Sorting of a trace file is implicitly assumed to be into file '`temp.idx1`'.

```
                                    make index
.....................................................................................
     set environment for importing references to code fragments and ids
     \def\ExtIndex{\openin15=temp.idx1   {
        \ifeof15    \else
           \catcode`\^^M=13
              \input temp.idx1
           \catcode`\^^M=5
           do after importing input file
        \fi   }
        collect references to identifiers }
     \ExtIndex  \bye
```

The group within the last macro restores `\Out` to its original definition (for the code in the next chapter). Sorting of the trace file '`puzzle2.trc`' gives the following outcome.

```
\<define\> {DefineTitle}{{define};;35}{}
\<define\> {ShowTitle}{{2}{1001}{35}}{2}
\<define\>.1002 {DefineCode}{{3}{1002}{56}}{2}
\<define\>.1002 {ShowCode}{{3}{1002}{56}}{2}
\<define\>.1007 {DefineCode}{{7}{1007}{129}}{4}
\<define\>.1007 {ShowCode}{{7}{1007}{129}}{4}
\<Functions\> {DefineTitle}{{Functions};;35}{}
\<Functions\> {ShowTitle}{{2}{1001}{35}}{2}
\<Functions\>.1006 {DefineCode}{{6}{1006}{108}}{3}
\<Functions\>.1006 {ShowCode}{{6}{1006}{108}}{3}
```

```
\<Functions\>.1008 {DefineCode}{{7}{1008}{145}}{4}
\<Functions\>.1008 {ShowCode}{{7}{1008}{145}}{4}
\<Global variables\> {DefineTitle}{{Global variables};;35}{}
.....................................................
```

To enable easy sorting, '`Ccode.sty`' instructs ProTeX to uniformly assign four digits to the code fragments (by starting counting at 1000 instead of 1).

```
\< title \> {DefineTitle}{{ title };;35}{}
\< title \> {ShowTitle}{ section number }{ fragment number }{35}}{2}
\< title \>.1002 {DefineCode}{{ section number }{ fragment number }{56}}{2}
\< title \>.1002 {ShowCode}{{ section number }{ fragment number }{56}}{3}
.....................................................
```

2.2 The Output

For each code segment, we want to introduce an entry of the following form into the index.

title `\dotfill` *references*

In addition, for each listing of a code title, we want to introduce an entry of the following form into the code fragment in which the title is listed. Such an entry produces a cross reference for the title.

`{` *fragment number* `}` `{` *title* `\dotfill` *references* `}`

For '`puzzle2`' we get an output of the following form.

```
 {1007} {define\dotfill \underbar{2} 3 7}
 {1002} {define\dotfill \underbar{2} 3 7}
 {1001} {define\dotfill \underbar{2} 3 7}
define\dotfill \underbar{2} 3 7
 {1008} {Functions\dotfill \underbar{2} 6 7}
.....................................................
```

```
          set environment for importing references to code fragments and ids
.....................................................................................................
    \newwrite\idxI
    \immediate\openout\idxI=temp.idx3
```

2.3 The Input Records

The input records are interpreted and processed by the following command.

```
          set environment for importing references to code fragments and ids
.....................................................................................................+
    \catcode`\^^M=13
    \def\<#1\>#2 #3#4#5^^M{%
       \def\temp{#3}%
       \ifx      \temp\dt Define Title ...
       \else\ifx \temp\st Show Title ...
       \else\ifx \temp\sc Show Code ...
       \fi\fi \fi }
    \catcode`\^^M=5
    \def\dt{DefineTitle}
    \def\st{ShowTitle}
    \def\sc{ShowCode}
```

The records tracing the construction of a code segment appear in a sequence that starts with the definition of the title. The records are processed one at a time according to the case.

- '`\<_title_\> {DefineTitle}{{_title_};;_input_line_no_}{}`'

  This record starts a sequence of records that is associated with the construction of a new code segment. At this point we want to do two things.

  - Output the entries '*title* `\dotfill` *references*' and '`{` *fragment number* `} {` *title* `\dotfill` *references* `}` ' that have been accumulated for the previous code segment (or, equivalently, for the previous title).

```
                                Define Title ...
.....................................................................................
     \Out
```

    The macro `\Out` is described in Section 2.4.

  - Start accumulating the entries for the new code segment. The macro `\entry` accumulates the information that needs to be listed, and the macro `\fragments` accumulates the locations where cross references should appear.

```
                                Define Title ...
.....................................................................................+
     \def\entry{#1\string\dotfill}
     \def\fragments{}
```

- '`\<_title_\> {ShowTitle}{{_section_no_}{_fragment_no_}{_input_line_no_}}{_page_no_}`'

  This entry shows a location where the code segment is used.

```
                                Show Title ...
.....................................................................................
     \ST #4
```

  The following macro adds '`_section_no`' to `\entry` and '`_fragment_no_`' to `\fragments`.

```
          set environment for importing references to code fragments and ids
.....................................................................................+
     \def\ST#1#2#3{
        \edef\entry{\entry\space \string\underbar{#1}}
        \edef\fragments{{\fragments}{#2}}  }
```

- '`\<_title_\>._fragment_no_ {ShowCode}{{_section_no_}{_fragment_no_}{_input_line_no_}}{_page_no_}`'

  Entries of this form show where fragments of the code segment are listed.

```
                                Show Code ...
.....................................................................................
     \SC #4
```

```
          set environment for importing references to code fragments and ids
.....................................................................................+
     \def\SC#1#2#3{
        \edef\entry{\entry\space#1}
        \edef\fragments{{\fragments}{#2}}  }
```

2.4 Writing the Output Records

The macro recursive `\Out` retrieves the information stored in `\entry` and `\fragments`.

```
          set environment for importing references to code fragments and ids
.....................................................................................................+
     \def\Out{
     \def\Out{
        \ifx \fragments\empty
           write index entry
        \else
           write cross reference entry
           \expandafter\Out
        \fi}}
```

To handle the boundary case that the first code segment has no predecessor, `\Out` is defined to do nothing at the first invocation. To handle the case that the last code fragment has no successor, `\Out` is also invoked after all the input is read.

```
                              do after importing input file
.....................................................................................................
     \Out
```

Add an Entry for the Index

The macro `\entry` holds the information '*title* `\dotfill` *references*' that should be inserted into the index for the current code title.

```
                                    write index entry
.....................................................................................................
     \immediate\write\idxI{\entry}
```

Add an Entry for a Cross Reference

The macro `\fragments` holds pointers to the code fragments that require cross references for the current title. For each pointer the command `\FragEntry` produces a record of the form '`{` *pointer* `} {` *content of* `\entry}` '

```
                              write cross reference entry
.....................................................................................................
     \expandafter\FragEntry\fragments
```

The macro `\fragments` carries a definition of the following form, using a recursive representation '`pointers → { pointers } { pointer }`' for its body.

```
     \def\fragments{
        pointers }
```

The macro `\FragmentEntry` receives a body '`{pointers} { pointer }`' of `\fragments` for actual parameters.

```
 {1012}
{1001} {define\dotfill \underbar{2} 3 7}
{1001} {Functions\dotfill \underbar{2} 6 7}
{1001} {Global variables\dotfill \underbar{2} 3 10}
.....  ........................................
{1011} {Global variables\dotfill \underbar{2} 3 10}
{1012} {  {1012}}
define\dotfill \underbar{2} 3 7
Functions\dotfill \underbar{2} 6 7
Global variables\dotfill \underbar{2} 3 10
.......................................
```

Figure 2.1 The file ‘puzzle2.2.idx1’.

```
          set environment for importing references to code fragments and ids
.....................................................................................+
    \def\FragEntry#1#2{\def\fragments{#1}
      \immediate\write\idxI{\space
         {#2} {\entry}}
      \ifnum #2>\FrgCount \FrgCount=#2 \fi}
    \newcount\FrgCount
```

The records that are prepared for the cross references start with a space character. Therefore, after the output file ‘temp.idx3’ is being sorted the records that belong to the index are pushed into the end of the file.

The following code introduces two additional records into the output file, to help separating the records designated for cross references from records designated for the index (see Figure 2.1).

- ‘{max pointer + 1}’. This record starts with two space characters and so after sorting it becomes the leading record in ‘temp.idx3’.
- ‘ {max pointer + 1} {...}’. After sorting ‘temp.idx3’, this record appears between the records that are used for cross references and the records that are used for the index.

```
                            do after importing input file
.....................................................................................+
    \advance\FrgCount by 1
    \def\entry{\space\space{\the\FrgCount}}
    \def\fragments{{}{\the\FrgCount}}
    \Out
```

Chapter 3

HANDLING REFERENCES TO IDENTIFIERS: makeindex.tex

References to identifiers are treated in a similar way as references to code titles.

3.1 The Input

The entries come in a sorted manner from `getids` (indirectly through '`temp.idx2`') .

```
                          collect references to identifiers
.....................................................................................
     \openin15=temp.idx2
     \ifeof15    \else
        \catcode`\^^M=13
           \input temp.idx2
        \catcode`\^^M=5
     \fi  \Out
```

To handle the boundary case that the first identifier that has no predecessor, `\Out` is defined to do nothing at the first invocation. To handle the case that the last identifier has no successor, `\Out` is also invoked after all the input is read.

3.2 The Output

For each identifier, we want to introduce an entry of the following form into the index (see also Figure ???(b)).

id `\dotfill` *references*

In addition, for each listing of an identifier, we want to introduce an entry of the following form into the code fragment in which the identifier is listed. Such an entry produces a cross reference for the identifier.

`{` *fragment number* `}` `{` *identifier* `\dotfill` *references* `}`

3.3 The Input Records

The input records are interpreted and processed by the following command. Duplicated records, resulting from identifiers that appear more than once in a code fragment, are ignored.

```
~{fail}{1007}{7}
~{fail}{1008}{7}
~{fail}{1008}{7}
~{fail}{1009}{8}
~{has_been_visited}{1002}{3}
~{has_been_visited}{1008}{7}
~{number_of_cells_visited}{1008}{7}
.............................................................



 {1009} {fail\dotfill7 7 8}
 {1008} {fail\dotfill7 7 8}
 {1007} {fail\dotfill7 7 8}
fail\dotfill7 7 8
 {1008} {has_been_visited\dotfill3 7}
 {1002} {has_been_visited\dotfill3 7}
has_been_visited\dotfill3 7
 {1011} {number_of_cells_visited\dotfill7 8 9 10}
.............................................................
```

Figure 3.1 Input/output of '`makeindex.tex`' for ids.

```
          set environment for importing references to code fragments and ids
.....................................................................................................+
     \def~#1#2#3{
        \def\temp{#1#2#3}
        \ifx\temp\lastrec \else
           \def\temp{#1}
           \ifx      \temp\lastid old id
           \else                  new id
           \fi
           \def\lastrec{#1#2#3}
        \fi}

     \def\lastid{}
```

All the records are of the form '`~{_id_} {_fragment_no_} {_section_no_}`' they are processed one at a time according to the case.

- A new identifier is encountered. At this point we want to do two things.
  - Output the entries '*id* `\dotfill` *references*' and '`{` *fragment number* `} {` *id* `\dotfill` *references* `}`' that have been accumulated for the previous identifier.

    ```
                                          new id
    ..........................................................................................
         \Out
    ```

  - Start accumulating the entries for the identifier. The macro `\entry` accumulates the information that needs to be listed, and the macro `\fragments` accumulates the locations where cross references should appear.

```
                                    new id
.....................................................................................+
     \def\lastid{#1}
     \def\entry{#1\string\dotfill#3}
     \def\fragments{{}{#2}}
```

- An old identifier is encountered.

  The following code adds '`_section_no`' to `\entry` and '`_fragment_no_`' to `\fragments`.

```
                                    old id
.....................................................................................
     \edef\entry{\entry\space #3}
     \edef\fragments{{\fragments}{#2}}
```

Chapter 4

FINDING REFERENCES TO IDENTIFIERS: getids

4.1 Information Base

The following code in the style file of '`Ccode.sty`' inserts delimiters around each code fragment.

Ccode

```
\Comment{/*}{*/
}
\def\CodeId#1#2{#1.#2.\the\secount}
```

The delimited fragments assume the following format.

```
/*\<title\>.fragment_no.section_no...*/
code_fragment /*...\<title\>.fragment_no.section_no*/
```

4.2 The Output of getids

The idea it to keep track of the section and fragment numbers for each code fragment. These numbers are recorded in the following variables.

variables

```
int fragno, secount;
char title[80];
```

These numbers are stored in the following pushdown memory when new code fragments are encountered, and they are retrieved from there when the ends of the fragments are reached.

variables

```
int active_frag[20][2],
    level=0;
```

The pattern for `sscanf` below cheats: it assumes that the character '`>`' instead of the string '`\>`' is the right delimiter. I don't know how to ask for a string delimiter in the pattern. If such is impossible, then some string manipulation on `yytext` can be done to overcome this problem.

start fragment

```
{
  sscanf(yytext,"/*\\<%[^>]>.%d.%d...*/",title,&fragno,&secount);
  level++; active_frag[level][0]=fragno;
         active_frag[level][1]=secount; }
```

end fragment

```
{
 level--; fragno =active_frag[level][0];
          secount=active_frag[level][1]; }
```

The identifiers to be recorded are printed out together with their corresponding section and fragment numbers. Duplications are allowed here, and they are eliminated latter on by `makeindex` (Section 3.3).

print ids that are not keywords

```
{ printf("~{%s}{%d}{%d}\n",yytext,fragno,secount);}
```

4.3 The Lex File

The application '`getids`' is defined in Lex with the following code.

lex ids

```
%{
variables
%}
%%
[ ]*    ;
[\n]    ;
keywords and statements to be ignored
["][^"]*["]    ;                                  strings between double quotes
"/*\\<".*"\\>."[0-9][0-9][0-9][0-9]"."[0-9]*"...*/" start fragment
"/*...\\<".*"\\>."[0-9][0-9][0-9][0-9]"."[0-9]*"*/" end fragment
"/*".*/"*/"     ;                                  comments not on fragments
[a-zA-Z][_a-zA-Z0-9]+  print ids that are not keywords
.    ;
%%
main()
{ yylex(); }
```

The following list is incomplete. It deals only with the needs of `puzzle2`.

```
                        keywords and statements to be ignored
.....................................................................................................
    else    |
    for     |
    fprintf |
    if      |
    int     |
    main    |
    printf  |
    putc    |
    register |
    return  |
    stderr  |
    stdout  |
    void    |
    "#include"[^\n]*    |
    "#define"  ;
```

Chapter 5

THE FILE Ccode.sty

A style file that deals mainly with (Al)ProTeX. The definition of the `\section` command in '`puzzle2.tex`' should have bben moved here, but it has not been moved to highlight the differences from '`puzzle1.tex`'.

```
                                    Ccode
.....................................................................................+
     \catcode`\:=11
        box code fragments
        put section numbers into trace file
        import cross references
        import index
     \catcode`\:=12
```

5.1 Boxed Code Fragments

The framing of code fragments is done below. Other shapes can be employed, and multiple shapes might also be used simultaneously (to get flowchart effects).

```
                                box code fragments
.....................................................................................
     \let\oldShowCode=\:ShowCode
     \edef\recallcat{\catcode`_=\the\catcode`_}
     \catcode`\_=13

      \def\:ShowCode#1{\nobreak
         \:FrameCode{\oldShowCode{#1}%
                     get cross references of fragment}}

     \recallcat
      \def\:FrameCode#1{\vtop{\noindent\vrule
                  \vtop{\hrule#1\hrule}\vrule}}
```

The cross references are introduced with the command `\cref` whose meaning is defined in Section 5.3.

```
                          get cross references of fragment
.....................................................................................
     \setbox\bx=\vtop{\hsize=0.45\hsize
        \leftskip=2em   \parindent=-2em
        \fiverm \baselineskip=7pt
        \catcode`\_=13 \def_{{\tt\string_}}%
        \cref }%
     \ifvoid\bx \else
        \noindent \  \vsplit\bx  to 0.55\dp\bx
        \hfill\box\bx\hbox{ }\smallskip
     \fi
```

The widths set for the boxes do not compensate for changes in `\leftskip` (a simple case that is considered for this literate program, i.e., see Section 3.3).

```
                                box code fragments
.....................................................................................+
     \newbox\bx
```

The design for the code titles, or the lack of it in this case, is determined in the following code.

```
                                box code fragments
.....................................................................................+
      \let\oldModifyShowCode=\ModifyShowCode
      \def\ModifyShowCode#1{\oldModifyShowCode{#1}%
                 \def\PortTitle##1{{\parindent=0pt
                     \leftskip=0pt plus 0.5\hsize
                     \rightskip=\leftskip ##1\par
                     \noindent\dotfill \null}}%
                 \def\:BottomTitle{}}
```

5.2 Trace Sections

Code fragments ids serve as major and secondary keys for sorting. Initializing their counter to 1000 gives to all the keys a uniform length of 4 digits (larger values can be used if more digits are desired). Such a value assumes that a program contains no more than 8999 code fragments.

```
                          put section numbers into trace file
.....................................................................................
     \def\:trc#1#2{\Tag{#1}{{\the\secount}{\the\:CodeNumber}{#2}}}
     \:CodeNumber=1000
```

5.3 Cross References into Code Fragments

The entries in the following file are sorted with fragment numbers as major keys (see Figure 2.1).

```
                                import cross references
.....................................................................................
     \newread\idxI
     \openin\idxI=\jobname.idx1
```

The first record in the input equals one plus the number of code fragments. Each record points to the code fragment that uses it, and the variable `\curr` holds the last pointer that has been encountered so far.

The command `\cref` nullifies itself when it exhausts all the cross referencing records, leaving the remaining record to the index.

import cross references

```
\newcount\curr
\def\cref#1{
  \def\cref{\ifnum #1=\curr
                \let\cref=\relax
           \else\expandafter\getcref
           \fi }}

\ifeof\idxI \let\cref=\relax
\else  \read\idxI to\record
       \expandafter\cref\record
\fi
```

When a new code fragment is listed, the records containing the related cross references are imported from '`\jobname.idx1`'. The leading field in each record identifies the code fragment to which the record should be associated, and the records appear in the file in the order in which they should be incorporated into the document.

import cross references

```
\def\getcref{%
   \ifnum    \curr>\:CodeNumber
   \else
      \ifnum \curr=\:CodeNumber  \NextRec\par  \fi
      \let\temp=\record
      \read\idxI to\record
      \ifx\temp\record
         \let\NextRec=\relax
      \else
         \expandafter\ReadRecord\record
      \fi
      \expandafter\getcref
   \fi}
\def\ReadRecord#1#2{\global\curr=#1
   \gdef\NextRec{#2}}
\def\NextRec{}
```

5.4 Import Index

The entries of the index appear after the entries for the cross references.

import index

```
\edef\recallcat{\catcode`_=\the\catcode`_}
\catcode`\_=13

\let\oldbye=\bye
\def\bye{{\vfill\break
   \leftskip=2em
   \parindent=-2em
   \catcode`\_=13 \def_{{\tt\string_}}%
   \InsertIndex }
   \csname oldbye\endcsname}

\recallcat

\def\InsertIndex{%
   \ifeof\idxI\else
       \read\idxI to\record
       \record\par
       \expandafter\InsertIndex
   \fi    }
```

The character '_' is considered to be standard within identifiers.

Appendix A

CODE FILES

The source files for the different versions of '`puzzle`' are listed here for completeness.

A.1 puzzle.w

The original CWEB source program.

```
\def\title{A Program to Solve ``The Convict Problem''}
@*The Problem.
Our task is to help a convict escape from prison.  The prison
is 4 cells square and the convict (represented by `C' in the diagram,
below) is in the northwesternmost cell.  He can move horizontally or
vertically (not diagonally) from cell to cell.  The only exit is in the
southeasternmost cell.
$$\vbox{\offinterlineskip
\def\tablerule{\noalign{\hrule}}
\def\tablespace{height2em&&&&&&&&\cr}
\halign{&\vrule#&\qquad#\qquad\cr
\tablerule\tablespace
&C&&P&&P&&P&\cr
\tablespace\tablerule\tablespace
&P&&P&&P&&P&\cr
\tablespace\tablerule\tablespace
&P&&P&&P&&P&\cr
\tablespace\tablerule\tablespace
&P&&P&&P&& &\cr
\tablespace\tablerule}}$$
Complicating the problem are the 14 policemen (represented by `P's in the
diagram) blocking the convict's way.  The convict must kill all the
policemen before he can leave, but cannot return to any cell he has
already been in (that would be too easy).

@*The Solution.
The program that solves the problem will be laid out like most C~programs:
@c
@<Header files used by the program@>@;
@<Global variables@>@;
@<Functions@>@;
```

```
@<The main program@>@;

@ We will need to represent the prison---a 2-dimensional array seems the
logical choice.  As the convict ``visits'' each cell, we will put a number denoting
the order in which the cell was visited into the appropriate array
element (we will use zero to represent a cell that has not yet been
visited).  Thus we can use the zero-ness of a
cell to determine whether or not it has been visited.

We can also use this property to avoid making special cases out of the
outer cells (which have fewer than~4 exits to other cells).  We create
an array 2~elements wider than the prison and initialize all the array
elements that do not correspond to actual cells to $-1$ (marking them
as ``already visited''---a convenient fiction).
All cells in the prison now have 4~neighbors and can be treated exactly
alike.

@dhas_been_visited(m,n) (prison[m][n]!=0)
@<Global...@>=
int prison[6][6] = {@t\1@>@/
  {-1,-1,-1,-1,-1,-1},@/
  {-1, 0, 0, 0, 0,-1},@/
  {-1, 0, 0, 0, 0,-1},@/
  {-1, 0, 0, 0, 0,-1},@/
  {-1, 0, 0, 0, 0,-1},@/
  {-1,-1,-1,-1,-1,-1} @t\2@>@/
  };

@ The main program is simple.  We let the |solve| function do all the
work.  If |solve| succeeds for the initial
cell~$(1,1)$, we print the configuration
of the prison (showing the order in which the cells were visited); if
not, we print an error message.  The latter case should never happen---it
will occur only if there is a bug in the program or if we have
misunderstood the problem.

@<The main...@>=
void
main(void)
{
  if (solve(1,1))@/print_prison(); else@/fprintf(stderr, "Impossible\n");
}

@ In order to produce the necessary output we need to include the
``Standard I/O'' library:
@<Header...@>=
#include <stdio.h>

@ Printing the prison configuration
is trivial, so we might as well get it out of the way.
@<Functions...@>=
void print_prison(void)
{
```

```
  register int i,j;
  for (i=1; i<=4;i++)
    for (j=1; j<=4;j++)
      @/printf("%2d%c", prison[i][j], j==4?'\n':'\t');
  putc('\n',stdout);
}

@ The |solve| function is fairly straightforward, but it has a few
special cases to deal with.  The parameters represent the row and column
numbers of the next cell to visit.  If the cell has already
been visited, it can't be visited again, so the current attempt at a
solution fails.  The solution can also fail prematurely if the convict
attempts to visit the exit cell~$(4,4)$ without having visited all the
other cells (and killing the policemen therein).

If the cell has not been visited before, we mark it as having been
visited and recursively check each of the 4 neighboring cells to see if a
solution exists starting from that cell (remember, all cells previously
visited have been marked and cannot be visited again).  If a solution
exists, we report success.  Otherwise, we report failure and pretend that
the convict hasn't actually visited this cell yet.
@d succeed return 1
@d fail return 0
@f succeed   return
@f fail   return

@<Functions...@>=
int
solve(register int m, register int n)
{
    if(has_been_visited(m,n))@/fail;
    @<If this is the exit cell, |succeed| if we have visited all the
      other cells, |fail| otherwise@>;
    prison[m][n]=++number_of_cells_visited;   /* mark cell as ``visited'' */
  if(solve(m+1,n)||solve(m,n+1)||solve(m-1,n)||solve(m,n-1))@/
    succeed;
  @<Pretend that the convict hasn't actually visited this cell yet@>;
  fail;
}

@ @<If...@>=
if (m==4&&n==4) {
  if (number_of_cells_visited==15)   /* all other cells have been visited */
    @/succeed;
  else@/ fail;
}

@ @<Pretend...@>=
--number_of_cells_visited;prison[m][n]=0;

@ We start out with no cells having been visited.
@<Global...@>=int number_of_cells_visited=0;
@*Epilogue.
```

```
When we ran the program, it produced the ‘Impossible’
message.
After checking out the program extensively, we found no bugs---we
did not completely understand the problem.  It turns out
that the convict, while not allowed to return
to a cell in which he has killed a policeman, {\it is\/} allowed to
return to his original cell.  The following is a valid
solution:
$$\eqalign{(1,1)&\to(1,2)\to(1,1)\to(2,1)\to(3,1)\to(4,1)\to(4,2)
\to(3,2)\to(2,2)\cr
&\to(2,3)\to(1,3)\to(1,4)\to(2,4)\to(3,4)\to(3,3)\to(4,3)\to(4,4)}$$

@*Index.
```

A.2 puzzle.tex

The following is a ProTeX variant of the original CWEB source program. The main differences are:

a. ‘`@ `’ replaced with ‘`\section{}`’

b. ‘`@*---.`’ replaced with ‘`\section{---}`’

c. ‘`@<full/partial title@>.  code`’ replaced with ‘`@<full title@><<< code >>>`’

```
\def\title#1{\noindent\hfil{\bf \uppercase{#1}}\bigskip
  \begingroup
     \parindent=0pt
     \openin15=\jobname.toc
     \ifeof15 \else \input \jobname.toc\fi
     \closein15
     \vfill
This is a ProTeX variant of the CWEB puzzle program
of Lee Wittenberg.
\par\break
  \endgroup}


\title{A Program to Solve ‘‘The Convict Problem’’}

\section{The Problem}
Our task is to help a convict escape from prison.  The prison
is 4 cells square and the convict (represented by ‘C’ in the diagram,
below) is in the northwesternmost cell.  He can move horizontally or
vertically (not diagonally) from cell to cell.  The only exit is in the
southeasternmost cell.
$$\vbox{\offinterlineskip
\def\tablerule{\noalign{\hrule}}
\def\tablespace{height2em&&&&&&&&&\cr}
\halign{&\vrule#&\qquad#\qquad\cr
\tablerule\tablespace
&C&&P&&P&&P&\cr
\tablespace\tablerule\tablespace
&P&&P&&P&&P&\cr
\tablespace\tablerule\tablespace
&P&&P&&P&&P&\cr
```

```
\tablespace\tablerule\tablespace
&P&&P&&P&& &\cr
\tablespace\tablerule}}$$
Complicating the problem are the 14 policemen (represented by ‘P’s in the
diagram) blocking the convict’s way.  The convict must kill all the
policemen before he can leave, but cannot return to any cell he has
already been in (that would be too easy).

\section{The Solution}
The program that solves the problem will be laid out like most C~programs:

@<program@><<<
@<Header files used by the program@>
@<define@>
@<Global variables@>
@<Functions@>
@<The main program@> >>>


\section{} We will need to represent the prison---a 2-dimensional array seems the
logical choice.  As the convict ‘‘visits’’ each cell, we will put a number denoting
the order in which the cell was visited into the appropriate array
element (we will use zero to represent a cell that has not yet been
visited).  Thus we can use the zero-ness of a
cell to determine whether or not it has been visited.

We can also use this property to avoid making special cases out of the
outer cells (which have fewer than~4 exits to other cells).  We create
an array 2~elements wider than the prison and initialize all the array
elements that do not correspond to actual cells to $-1$ (marking them
as ‘‘already visited’’---a convenient fiction).
All cells in the prison now have 4~neighbors and can be treated exactly
alike.


@<define@><<<
#define has_been_visited(m,n) (prison[m][n]!=0)
>>>

@<Global variables@><<<
int prison[6][6] = {
  {-1,-1,-1,-1,-1,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1,-1,-1,-1,-1,-1}
  };>>>

\section{} The main program is simple.  We let the |solve| function do all the
work.  If |solve| succeeds for the initial
cell~$(1,1)$, we print the configuration
of the prison (showing the order in which the cells were visited); if
```

```
not, we print an error message.  The latter case should never happen---it
will occur only if there is a bug in the program or if we have
misunderstood the problem.

@<The main program@>===
void
main(void)
{
  if (solve(1,1))
          print_prison();
        else
          fprintf(stderr, "Impossible\n");
}
>>>

\section{} In order to produce the necessary output we need to include the
``Standard I/O'' library:

@<Header files used by the program@>===
#include <stdio.h>
>>>


\section{}Printing the prison configuration
is trivial, so we might as well get it out of the way.


@<Functions@>===
void print_prison(void)
{
  register int i,j;
  for (i=1; i<=4;i++)
    for (j=1; j<=4;j++)
        printf("%2d%c", prison[i][j], j==4?'\n':'\t');
  putc('\n',stdout);
}
>>>


\section{}The |solve| function is fairly straightforward, but it has a few
special cases to deal with.  The parameters represent the row and column
numbers of the next cell to visit.  If the cell has already
been visited, it can't be visited again, so the current attempt at a
solution fails.  The solution can also fail prematurely if the convict
attempts to visit the exit cell~$(4,4)$ without having visited all the
other cells (and killing the policemen therein).

If the cell has not been visited before, we mark it as having been
visited and recursively check each of the 4 neighboring cells to see if a
solution exists starting from that cell (remember, all cells previously
visited have been marked and cannot be visited again).  If a solution
exists, we report success.  Otherwise, we report failure and pretend that
the convict hasn't actually visited this cell yet.
```

```
@<define@><<<
#define  succeed return 1
#define  fail return 0
>>>


@<Functions@><<<
int
solve(register int m, register int n)
{
    if(has_been_visited(m,n))
                        fail;
    @<If this is the exit cell, |succeed| if we have visited all the other cells, |fail| otherwise@>
    prison[m][n]=++number_of_cells_visited;   /* mark cell as ``visited'' */
  if(solve(m+1,n)||solve(m,n+1)||solve(m-1,n)||solve(m,n-1))
    succeed;
  @<Pretend that the convict hasn't actually visited this cell yet@>;
  fail;
}
>>>

\section{}

@<If this is the exit cell, |succeed| if we have visited all the other cells, |fail| otherwise@><<<
if (m==4&&n==4) {
  if (number_of_cells_visited==15)   /* all other cells have been visited */
    succeed;
  else fail;
}>>>

\section{}

@<Pretend that the convict hasn't actually visited this cell yet@><<<
--number_of_cells_visited;prison[m][n]=0;
>>>

\section{}We start out with no cells having been visited.


@<Global variables@><<<
int number_of_cells_visited=0;
>>>

\section{Epilogue}
When we ran the program, it produced the `Impossible'
message.
After checking out the program extensively, we found no bugs---we
did not completely understand the problem.  It turns out
that the convict, while not allowed to return
to a cell in which he has killed a policeman, {\it is\/} allowed to
return to his original cell.  The following is a valid
solution:
```

```
$$\eqalign{(1,1)&\to(1,2)\to(1,1)\to(2,1)\to(3,1)\to(4,1)\to(4,2)
\to(3,2)\to(2,2)\cr
&\to(2,3)\to(1,3)\to(1,4)\to(2,4)\to(3,4)\to(3,3)\to(4,3)\to(4,4)}$$




\OutputCode@<program@>

\bye
```

A.3 puzzle1.tex

Calls '`puzzle.tex`' with `CWEB` oriented style of prose. No index and cross references are provided.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%  setting of ProTex
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\input ProTex.sty
\AlProTex{c,<<<>>>,list,title,@}
\catcode`\@=0

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%  set web style for document
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


\newwrite\toc
\openout\toc=\jobname.toc
\newcount\secount

\def\section#1{\bigskip \advance\secount by 1
   \def\temp{#1}%
   \ifx\temp\empty \else
      \edef\temp{\write\toc{#1\string\dotfill\the \secount
         \hbox to 10mm{\string\hfil\noexpand
                       \the\pageno}\string\par}%
       }\temp \vfill\break
   \fi
   {\noindent\bf\the\secount. #1\quad}\ignorespaces}

\hbadness=10000     \vbadness=10000  \hfuzz=99in \vfuzz=99in

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% end style file, start program
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\input puzzle.tex
```

A.4 puzzle2.tex

Calls ‘puzzle.tex’ with modified style for prose. Index and cross references are provided.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%  setting of ProTex
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\input ProTex.sty
\AlProTex{c,<<<>>>,list,title,@,[]}
\catcode`\@=0

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%  set web style for document
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


\newwrite\toc
\openout\toc=\jobname.toc
\newcount\secount

\font\Bf=cmbx10  scaled \magstep 3

\def\section#1{\bigskip \advance\secount by 1
   \def\temp{#1}%
   \ifx\temp\empty {\noindent\bf\the\secount. #1\quad}%
   \else
      {\bigskip\noindent\Bf\the\secount. #1\par\nobreak\medskip}%
      \edef\temp{\write\toc{#1\string\dotfill\the \secount
         \hbox to 10mm{\string\hfil\noexpand
                       \the\pageno}\string\par}%
       }\temp
   \fi \nobreak}

\hbadness=10000     \vbadness=10000  \hfuzz=99in \vfuzz=99in


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% extras for ProTeX
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\input Ccode.sty

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% end style file, start program
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\input puzzle.tex
```