assoc.web
A
General Purpose Filing System
Literate Program

by

A. J. Hurst

Department of Computer Science

Monash University

Version 2.2

Abstract

This literate program maintains a database that allows quick location of files, and cross referencing of material stored in those files. Although designed specifically for paper files, the program could be used for other applications, as it makes very few assumptions about the filing system, other than that it is arranged in a pure hierarchy.

1. Background to this Program. This program was written and documented by John Hurst, e-mail address `ajh@cs.monash.edu.au`. It was originally written to help in keeping track of my (paper) filing system, for dealing with the mountains of paper that universities seem to generate. Over the years it has been extended to include not just administrative files, but also my research files (modelled on the *ACM Computing Reviews* indexing system [**CRReviews?**], and my hobbies files kept at home.

The problem I found myself facing was

- when a piece of paper arrived on my desk, where was I to put it? (And don't say the rubbish bin!) I wanted something that would allow a quick locator of an appropriate file.
- when faced with a problem for which I knew there was a piece of paper that resolved that problem, how could I find the piece of paper?

The first problem was addressed by developing a hierarchical filing system, where the path to a given file was numbered in tree fashion, such as 3.2.8.7. Each file was not only labeled with its path, but also a more descriptive label, like "3.2.8.7 students.FCIT.faculty.administration". Thus, if the piece of paper was about a student enrolled in the Faculty of Computing and Information Technology, following the descriptive terms down the tree should identify the folder 3.2.8.7, and lead straight to the right part of the filing cabinets. This mechanism assumes that a strict hierarchy can be found to direct the structure of the database.

The second problem is more subtle, and is related to the fact that a pure hierarchy is not always easily identified. While it is quite possible that a direct descriptive path will exist, so that one can use the mechanism described above, often I found myself with cross-referenced connections. These connections I term "associations", and they were originally conceived of as rather like Unix links, but they have the significant distinction that they are not necessarily synonymous with the identified file. Associations may indeed lead to more than one file. A file has an association when there is another descriptive term that can be used to identify it, but which may itself have some other connotations.

Hence I may have a file identified as "1.4.3.8 Fred Nurk.students.graduate.teaching", but there is something relevant to Fred Nurk's file that is actually filed under "3.2.8.7 students.FCIT.faculty.administration", because Fred happens to have caught the Faculty's attention for some reason or other, and there is a document in there that mentions Fred. The connection or association is therefore handled by attaching an additional piece of text to the node of the tree that cross references the other node. The association may be uni- or bi-directional.

For example, the association given above as an example might be represented textually as "1.4.3.8 Fred Nurk(3.2.8.7).students.graduate.teaching" in one direction, or "3.2.8.7 students(Fred Nurk).FCIT.faculty.administration" in the other direction. It is not essential that the link be two-way.

2. Philosophical Note. This program is trivial. This documentation is also trivial. No particular claim to originality is made, other than that this document exists, and hundreds of other potential documents like it don't. I would draw two conclusions:

- People think that documenting their programs is not easy.
- People think that documenting their programs is a waste of time.

Having sat recently upon a promotions committee, I was struck by the fact that of a dozen or so applicants for promotion (within the Computing Faculty), not one applicant listed a piece of software as part of his/her productive output. Why is the production of high quality, readable software paid such little lip service, let alone practice? Literate Programming tools such as CWEB [**CWEB?**], used here, make this a straightforward task.

I make no claim for quality or readability here. The exercise is more one of setting a precedent. It is my hope that that precedent will lead to others that are high quality, readable, documents.

3. One important attribute of the system that I wanted was that it should have a "sensible" storage format. This meant a text file to maintain the data over the longer term, which would allow direct editing of the data in the event of something going wrong, and also allow minor modifications and updates directly. Whether you think this a good thing is probably an issue of stylistic belief!

The text format used is very straightforward. Each line of the file identifies a single file, which in turn is represented by a single node in the overall tree structure. The depth of nesting of the node is indicated by its indentation level, and the first non-blank character on the line is taken to be the selector character for the node, when traversing the tree from the parent node. Further text is interpreted as the file name, and a parenthesized list of associations, each separated by commas completes the line. If there are no parentheses, the associations are empty.

4. Here is a sample of the textual form of the data:

```
 3 administration
   1 department
     0 general
       1 business cards
       2 introduction
     ...
   2 faculty
     ...
     8 FCIT(BComp,faculty of computing,new faculty)
       0 general(telephone directory)
       1 Frankston
         1 finance and budget2
           1 travel2
             1 claim forms2
           ...
       7 students(credit transfer)
       ...
   3 university
   ...
 4 professional
   0 general
   ...
 D Software
   0 general(Wirth's 4 Software Laws)
   1 programming techniques
     5 persistent programming(persistence)
       3 hello(another test,asd)
       ...
```

5. Further Philosophical Note. This program is trivial, yet I wrote it because I needed something that works, was simple to use, was highly portable, and very robust. These attributes must apply to countless other pieces of software. If the software is other than ephemeral or 'once off', then motivation for documentation in this form surely exists.

A much more elaborate program could have been written (and may still be, if circumstances warrant it), but there comes a point at which one must draw the line. This work stands on its own (limited) merit – while there is yet more to be done, sufficient has been done to make the point, and to be usable. PhD students writing up their theses should also espouse this philosophy!

6. User Notes. To use *assoc*, you must make sure that the executable is in your path. On the Monash Computer Science machines, add `/u/people3/ajh/bin/`*machine-type* to your path, where *machine-type* is `dec`, `sun4`, or `indy` as appropriate.

You must also define where the database is to be stored. Do this by setting the environment variable `ASSOCDB` to the full path name of your database file. This file can be setup according to the format described in the previous section, if you want to create some initial data using a text editor. Alternatively, if you want to start with an empty database, an empty file should be created (for example, using the command `touch $ASSOCDB`).

Then run *assoc*. It will not normally display anything until commands are given to it. The new-line character is treated as the print command, so entering blank lines will print the current node, and the current node will also be printed after a line of 1 or more commands.

All commands are performed on the *current node*, which may be moved around the database. The current node is initialized to the *root node*, and it may be returned to the root node at any stage by the `r` command. The command to move down the tree is `d`, which must be followed by the leaf number of the node selected, and the command to move up the tree is `u`.

The other commands are described in the next paragraph, ⟨ Command Description 7 ⟩.

7. The commands that we handle at the moment, together with their alternative forms, are:

Command	Alternates	Parameter	Operation
`?`	`h`		print a help message
`q`			quit the program
`+`	`= a`	`string`	add an association *string* to the current node
`#`	`c`		clear the markings on nodes
`>`	`, g`	`char`	go to a given node '**char**'
`;`	`e`	`char string`	add/replace a child node selector **char**, name *string*
`*`	`p`		pattern match all nodes for names matching a given pattern
`^`	`m`		find a pattern in previously matched nodes
`-`	`_`	`k`	kill an association set for the current node
`@`	`l`		set the current location to the given parameter
`'`	`" n`		find the next occurrence of the current pattern
`` ` ``	`~ r`		go to the absolute path name given
`<`	`, u`		go up to the parent node
`/`	`f`	`string`	find the given string pattern

⟨ Command Description 7 ⟩ ≡

This code is cited in section 6.

8. Most of the commands are self explanatory, although a couple of remarks are in order. The commands are chosen for their mnemonic value, so that `>` means "follow this branch", while `<` means return to previous (parent) node. The `.` and `,` alternatives represent the unshifted characters as assigned to most keyboards. The mnemonic value of other commands may (or may not) be as obvious!

For finding a particular node, the "find" command (`f` or `/`) does a straightforward string search, searching over the nodes below the current node. This will generally suffice for most simple searches.

However, for more complex searches, the "pattern match" '`p`' search command is intended to allow arbitrary regular expression matching. (Unfortunately, this hasn't been completely implemented yet!) The process of matching is that an initial set of potential nodes is selected by a '`p`' command, which marks nodes as matching a given pattern. This collection can be either extended by further '`p`' commands, or narrowed by the match command '`m`', which marks only those nodes already marked and which match the given pattern. In this way, disjunctive ('or') matches can be constructed with the '`p`' command command, and conjunctive ('and') matches can be constructed with the '`m`' command.

This is all very sketchy detail, but until I sit down and write a more comprehensive specification, it will have to do. I shall leave these hooks in rather than take them out for publication, as they may prove useful in future.

9. Top Level Description.

Maintenance History

Date	Author	Vers.	Comment or Reason
1993?	John Hurst	0	Original C Version
1994	John Hurst	1.0	Eiffel Version
6 Jun 1995	John Hurst	2.0	Re-engineered C Version
15 Sep 1995	John Hurst	2.1	Modified for this CWEB document
22 Nov 1995	John Hurst	2.2	Added additional letter commands

This program provides a browsing and editing facility for a hierarchical tree structure. The tree is defined by a flattened text form, with one line of input for each node. The indentation of each line defines the node depth, and the first (non-blank) character defines the selector for each edge of the tree.

Nodes of the tree are named, and the name is given by subsequent text on the input line after the selector character. Whitespace must separate the selector character from the naming text.

Each node may also have some associated text, which is given in parentheses after the naming text. Multiple associations may be made; each separate association is given by comma separated strings within the parentheses.

The form of the code is conventional: some global variable and constant declarations, various procedures, then the main program. We define all this in the following outline.

⟨ Global Declarations 27 ⟩
⟨ Procedure Declarations 12 ⟩
⟨ Main Program 10 ⟩

10. The main program provides overall control: get the filename of the file to read, as defined by a command line parameter, get the data from that file, rename the file to provide a backup, and then browse the data. Once the browser is quit, the data is written back to the file (with possible backup implications), and then program itself quits.

⟨ Main Program 10 ⟩ ≡

```
main( )
{
  char *fn;
  int result;
  fn = getfilename( );
  getdata(fn);
  makebackup(fn);
  browse( );
  result = putdata(fn);
  if (result ≠ 0) restorebackup(fn);
  exit(result);
}
```

This code is used in section 9.

11. Let's have a look at the data structures involved.

The main data structure manipulated by this system is a tree. It is represented by nodes linked through the pointers

parent	the parent node of this node
sibling	the next sibling to this node
child	the first child of this node

Each node has a *selector* field which indicates to which child arc of the parent's subtree it belongs. This field must be unique across a given subtree.

The *primary* field defines the full name of the node. It is an arbitrary text string.

The *secondary* field defines a *list* of associations for the node. Each association is itself an arbitrary text string (but there are some constraints on the contents of these strings, imposed by the scanning mechanism).

The *list* datatype is declared to handle arbitrary lists of string values.

The flag *marked* indicates whether or not the node is marked. This is used by the search routines.

The variable *location* is used to indicate the location of a file. This is a numeric value, and no further interpretation is placed upon it by this program. It would be nice if this were a string type, and the user of the program could then add mnemonic value to this field. This might be implemented in some future version of this program.

```
⟨ Global Data Structures 11 ⟩ ≡
  typedef struct list_struct {
    char *value;
    struct list_struct *next;
  } list;
  typedef struct node_struct {
    char selector;
    struct node_struct *parent, *sibling, *child;
    char *primary;
    list *secondary;
    int marked;
    int location;
  } node;
```

This code is used in section 27.

12. The main procedural components of the system are the browsing routine, *browse*, followed by the i/o routines *makebackup*, *rmbackup*, and *restorebackup*. *browse* itself calls a number of other top-level routines, detailed in sections ⟨ top level routine declarations 37 ⟩ and following.

```
⟨ Procedure Declarations 12 ⟩ ≡
  ⟨ browse declaration 13 ⟩
  ⟨ I/O routine declarations 33 ⟩
  ⟨ top level routine declarations 37 ⟩
```

This code is used in section 9.

13. The heart of the system is the browsing routine, so let's look at that next.

The body of this procedure enters a simple command interface loop, reading characters, and performing an n-way switch upon the characters read.

```
⟨ browse declaration 13 ⟩ ≡
  void browse( )
  {
    char c;
    node *current, *foundnode, *lastnode;
    char sel;
    char *prkey;
    current = &root;
    pattern = Λ;
    while (1) {
      c = getchar( );
      switch (c) {
      case '?': case 'h': ⟨ Supply Help 14 ⟩ break;
      case 'q': return;
      case '+': case '=': case 'a': ⟨ Add an Association to the Current Node 15 ⟩ break;
      case '#': case 'c': clearnodes(root); break;
      case '>': case '.': case 'g': ⟨ Down to Given Node 16 ⟩ break;
      case ':': case ';': case 'e': ⟨ Edit a Child Node 17 ⟩ break;
      case '*': case 'p': ⟨ Find All Patterns 18 ⟩ break;
      case '^': case 'm': ⟨ Find Pattern in Previously Matched Nodes 19 ⟩ break;
      case '_': case '-': case 'k': ⟨ Kill an Association 20 ⟩ break;
      case '@': case 'l': ⟨ Set Location 21 ⟩ break;
      case '\'': case '"': case 'n': ⟨ Next Occurrence of Pattern 22 ⟩ break;
      case '\n': displaylevel(current, 1); break;
      case '`': case '~': case 'r': ⟨ Go To Absolute Path Node 23 ⟩ break;
      case '<': case ',': case 'u': ⟨ Go Up To Parent Node 24 ⟩ break;
      case '/': case 'f': ⟨ Pattern Match 25 ⟩ break;
      default: ⟨ Treat anything not recognized as Illegal 26 ⟩ break;
      }
    }
  }
```

This code is used in section 12.

14. Supplying help just means printing a list of various messages detailing the command, parameters, and operation of each top-level operation. We also read the next character, discarding it if it is a newline. Newlines by themselves cause a display of the current node, which is inappropriate in the context of the help message.

```
⟨ Supply Help 14 ⟩ ≡
  printf("␣␣␣␣+␣=␣<ssss>␣␣add␣an␣association␣\"ssss\"\n");
  printf("␣␣␣␣#␣␣␣␣␣␣␣␣␣␣clear␣marking␣of␣nodes\n");
  printf("␣␣␣␣>␣.␣<c>␣␣␣␣␣down␣to␣node␣labelled␣\'c\'\n");
  printf("␣␣␣␣:␣;␣c<ssss>␣edit␣child␣node␣\'c\'␣to␣\"ssss\"\n");
  printf("␣␣␣␣_␣-␣<ssss>␣␣kill␣association␣\"ssss\"\n");
  printf("␣␣␣␣@␣␣␣<n>␣␣␣␣␣set␣location␣to␣n\n");
  printf("␣␣␣␣\"␣'␣␣␣␣␣␣␣␣next␣occurrence␣of␣pattern\n");
  printf("␣␣␣␣␣␣␣␣/ssss/␣␣find␣pattern␣\"ssss\"\n");
  printf("␣␣␣␣␣␣␣␣*ssss␣␣␣show␣all␣patterns␣\"ssss\"\n");
  printf("␣␣␣␣`␣~␣(path)␣␣go␣to␣absolute␣node\n");
  printf("␣␣␣␣<␣,␣␣␣␣␣␣␣␣go␣up␣to␣parent\n");
  printf("␣␣␣␣q␣␣␣␣␣␣␣␣␣␣to␣quit␣the␣program\n");
    /* throw away CR if it follows immediately in input */
  if ((c = getchar()) ≠ '\n') ungetc(c, stdin);
```

This code is used in section 13.

15. To add an association to the current node, we must first collect the parameter *prkey* that defines the association text (a string), then call the routine *addassoc* which does the actual work.

```
⟨ Add an Association to the Current Node 15 ⟩ ≡
  prkey = getstr();
  addassoc(&current→secondary, prkey);
```

This code is used in section 13.

16. Move down to a given node, selected by the parameter character. Keep track of where we are, in case the selection fails. Selection failure is indicated by a returned value of Λ.

```
⟨ Down to Given Node 16 ⟩ ≡
  lastnode = current;      /* retain current node pointer */
  current = choose(current, c = getchar());      /* try to move to new node */
  if (current ≡ Λ) current = lastnode;      /* couldn't, so reset */
```

This code is used in section 13.

17. Add a new child node to the tree. The node has an edge selector and a name. The selector is given by a single parameter character, and the name by a string parameter. Both of these are read here. The called routine *addnode* does the real work.

```
⟨ Edit a Child Node 17 ⟩ ≡
  sel = getchar();
  prkey = getstr();
  foundnode = addnode(current, sel, prkey);
```

This code is used in section 13.

18. Find all nodes that match a given pattern. Currently the pattern match is a simple unanchored substring match. Matched nodes are flagged using the *marked* field of the node.

```
⟨ Find All Patterns 18 ⟩ ≡
  if (pattern ≠ Λ) free(pattern);
  pattern = readpattern( );
  current = findpatterns(current, pattern);
```

This code is cited in section 19.

This code is used in section 13.

19. Here we perform an additional search, where the nodes searched are just those selected by a previous search. This gives a way of forming conjunctive searches, that is, of finding nodes that match more than one criterion. The *marked* field of a node is used to select only those nodes that have been matched previously, either as in section ⟨ Find All Patterns 18 ⟩ or this section.

The process of searching is recursive over the data tree, and always starts with the current node, searching the entire subtree below it.

```
⟨ Find Pattern in Previously Matched Nodes 19 ⟩ ≡
  if (pattern ≠ Λ) free(pattern);      /* collect pattern */
  pattern = readpattern( );      /* apply search, starting from root, reset current */
  current = &root;
  lastnode = &root;
  while (current ≠ Λ) {
    foundnode = search(pattern, strlen(pattern), current);      /* match only previously marked nodes */
    if ((foundnode ≠ Λ) ∧ (foundnode→marked ≡ 1)) {
      displaylevel(foundnode, 0);
      current = lastnode = foundnode;
    }
    current = nextnode(current);
  }
  if (lastnode ≡ &root) msg("not␣found");
  current = lastnode;
```

This code is used in section 13.

20. Deleting an association (which is specified by entering a parameter string which must match exactly the target for deletion) is done by the procedure *delassoc*.

```
⟨ Kill an Association 20 ⟩ ≡
  prkey = getstr( );
  delassoc(&current→secondary, prkey);
```

This code is used in section 13.

21. This operation changes the location marked for the current node. Since the database is changed, we set the *modified* bit to so indicate.

```
⟨ Set Location 21 ⟩ ≡
  scanf("%d", &current→location);
  modified = 1;
```

This code is used in section 13.

22. The searching operations always print the first node that matches the given search criteria. To find other matching nodes, this operation moves around the tree using the operation *nextnode*, which returns the next node from the current node in depth first, pre-order traversal.

```
⟨ Next Occurrence of Pattern 22 ⟩ ≡
  if (pattern ≡ Λ) {
    msg("no␣current␣pattern");
    break;
  }     /* loop till either we find the pattern, or we have searched from root without finding anything */
  while (1) {
    current = nextnode(current);      /* if no nextnode, reset to root */
    if (current ≡ Λ) current = &root;
    foundnode = search(pattern, strlen(pattern), current);      /* have we found something? */
    if (foundnode ≠ Λ) {
      current = foundnode;
      break;
    }
    if (current ≡ &root) {
      msg("not␣found");
      break;
    }
  }
```

This code is used in section 13.

23. This command finds a node given its path from the root node. Each selector down the tree is separated from the previous selector by a dot '.', which is the same as the command to go down a relative path. Hence this command could be simplified to just start at the root node, and let the command line switch do the rest, but I haven't bothered. Why fix it when it's not broken?

```
⟨ Go To Absolute Path Node 23 ⟩ ≡
  current = lastnode = &root;
  while ((c = getchar( )) ≠ '\n') {
    if ((c ≠ '.') & (lastnode ≠ Λ)) lastnode = choose(current, c);
    if (lastnode ≠ Λ) current = lastnode;
  }
  if (lastnode ≡ Λ) msg("not␣a␣valid␣path");
  displaylevel(current, 1);
```

This code is used in section 13.

24. Going back to the parent node is trivial: just follow the *parent* link in the current node.

```
⟨ Go Up To Parent Node 24 ⟩ ≡
  if (current ≠ &root) current = current→parent;
```

This code is used in section 13.

25. We can search for node names that match a given pattern. This match process also matches association keys. The actual pattern match is performed by a procedure call. Here we just collect the pattern and call it, starting from the root node. If no node matches the pattern, we return to the root node.

```
⟨ Pattern Match 25 ⟩ ≡
  if (pattern ≠ Λ) free(pattern);      /* collect pattern */
  pattern = readpattern( );      /* apply search, starting from root, reset current */
  current = search(pattern, strlen(pattern), &root);      /* not found, reset to root */
  if (current ≡ Λ) {
    current = &root;
    msg("not␣found");
  }
  displaylevel(current, 1);
```

This code is used in section 13.

26. Anything else, trash it, and issue a message.

```
⟨ Treat anything not recognized as Illegal 26 ⟩ ≡
  msg("illegal␣command");
```

This code is used in section 13.

27. Global Declarations. Here are all the global data values and procedures. We restrict (rather arbitrarily) the maximum tree depth handled to be `TREE_DEPTH`, and the maximum string length for various string buffers to be `MAXSTR`.

⟨ Global Declarations 27 ⟩ ≡
```
#include <stdio.h>
#define TREE_DEPTH 10
#define MAXSTR 100
  ⟨ Global Macro Definitions 29 ⟩
  ⟨ Global Data Structures 11 ⟩
  ⟨ Global Variables 28 ⟩
  ⟨ Global Procedure Templates 30 ⟩
  ⟨ Global System Dependencies 31 ⟩
```
This code is used in section 9.

28. The global variable *root* is the root of the data tree. *pattern* is a string pointer to the pattern used for searching purposes. The variable *modified* is set to 1 if the database is modified during user interaction, but is otherwise unmodified.

⟨ Global Variables 28 ⟩ ≡
```
  node root;
  char *pattern;
  int modified = 0;
```
This code is used in section 27.

29. `LOWER` converts alphabetic characters to lower case, for pattern matching purposes.

⟨ Global Macro Definitions 29 ⟩ ≡
```
#define UPTOLOW ((int) 'A' + (int) 'a')
#define LOWER(c)(c ≥ 'A' ∧ c ≤ 'Z') ? (char) ((int) c − UPTOLOW) : c
```
This code is used in section 27.

30. This bit is boring. It is here to provide forward declarations of all procedures (not just those used recursively), for completeness sake.

⟨ Global Procedure Templates 30 ⟩ ≡

```
int displayassocs( );
void displaylevel( );
void clearnodes( );
void marknodes( );
node *choose( );
char *getstr( );
int emptystr( );
node *insert( );
node *addnode( );
void addassoc( );
void delassoc( );
int command( );
char *readpattern( );
int patmatch( );
node *search( );
node *nextnode( );
node *findpatterns( );
char *string_copy( );
void getdata( );
char *getfilename( );
void msg( );
void traverse( );
int putdata( );
extern char *malloc( );
```

This code is used in section 27.

31. Of course, there has to be something different on the two systems with which I work, and here it is.

⟨ Global System Dependencies 31 ⟩ ≡

```
#ifdef PYR
#include <strings.h>
#endif
#ifdef SUN
#include <string.h>
#endif
```

This code is used in section 27.

32. I/O Routine Declarations.

33. *makebackup* ensures that a disaster in the middle of writing the updated data back into the file will never cause data to be lost. The file from which data is initially read is renamed, by appending ". *bak*" to the filename attached to the original file. The old filename is then used to name the new file, which if corrupted, can be discarded without prejudicing the original data file.

```
⟨ I/O routine declarations 33 ⟩ ≡
  makebackup(fn)
       char *fn;
  {
    char fnb[100];
    strcpy(fnb, fn);
    strcat(fnb, ".bak");      /* make backup file name */
    unlink(fnb);
    link(fn, fnb);      /* link backup name to original file */
    unlink(fn);      /* disconnect original file name from backup file */
  }
```

See also sections 34 and 35.

This code is used in section 12.

34. Once the new file has been created and written, the backup version can be discarded.

```
⟨ I/O routine declarations 33 ⟩ +≡
  rmbackup(fn)
       char *fn;
  {
    char fnb[100];
    strcpy(fnb, fn);      /* make backup file name */
    strcat(fnb, ".bak");
    unlink(fnb);      /* new file has been written, remove backup file */
  }
```

35. In the event of not being able to write the new output file, we reset the backup file to the original file name. This ensures that all the transactions performed by this invocation of the program are all aborted.

```
⟨ I/O routine declarations 33 ⟩ +≡
  restorebackup(fn)
       char *fn;
  {
    char fnb[100];
    strcpy(fnb, fn);      /* make backup file name */
    strcat(fnb, ".bak");
    link(fnb, fn);      /* new file not written, link original file name back */
    unlink(fnb);      /* remove reference to backup file */
  }
```

36. Top-Level Routines. Now follow the top-level routines used by *browse.c*

37. *displayassocs* display associations for the node p. It will return 1 if there is a non-empty association list, 0 if it is empty.

```
⟨ top level routine declarations 37 ⟩ ≡
  int displayassocs(p)
        node *p;
  {
    char s;
    list *ls;
    s = '(';
    ls = p→secondary;
    if (ls ≡ Λ) return 0;
    while (ls ≠ Λ) {
      if (s ≡ '(') printf("␣␣␣␣␣␣␣␣");
      printf("%c%s", s, ls→value);
      s = ',';
      ls = ls→next;
    }
    if (s ≡ ',') printf(")");
    return 1;
  }
```

See also sections 38, 39, 40, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 59, 60, 61, and 62.

This code is cited in section 12.

This code is used in section 12.

38. *displaylevel* is called after any operation which repositions the current node, and can be used in general to print information about a node. The first parameter p is a node pointer (usually *current*) that indicates which node is to be printed. The second parameter *full* indicates (if non-zero) that the children of the node should also be printed.

Since the root node has no explicit number, it is treated as a special case.

```
⟨ top level routine declarations 37 ⟩ +≡
  void displaylevel(p, full)
       node *p;
       int full;
  {
    node *l;
    node *stk[TREE_DEPTH];
    int sp = 0;
    list *ls;
    char s;
    if (p ≡ &root) {      /* print root indication */
      printf("root\n");
    }
    else {      /* scan parents to find path, store in stk */
      l = p;
      while (l ≠ Λ) {
        stk[sp++] = l;
        l = l→parent;
      }
      sp--;      /* back to point at last entry */      /* print stk in shallowest first order */
      sp--;      /* remove root */
      while (sp > 0) printf("%c.", stk[sp--]→selector);
      printf("%c␣", stk[0]→selector);      /* scan parents to print path in deepest first order */
      l = p;
      while (l ≠ &root) {
        printf("%s.", l→primary);
        l = l→parent;
      }
      if (p→location ≠ 1) printf("␣{%d}", p→location);
      printf("\n");
    }
    if (displayassocs(p)) printf("\n");
    if (¬full) return;      /* scan children of current node */
    l = p→child;
    while (l ≠ Λ) {
      printf("␣␣%c␣%s", l→selector, l→primary);
      if (l→location ≠ 1) printf("␣{%d}", l→location);
      displayassocs(l);
      printf("\n");
      l = l→sibling;
    }
  }
```

39. Nodes may be marked. The *clearnodes* routine clears out any current marking, starting from the node given by the parameter *subtree*.

```
⟨ top level routine declarations 37 ⟩ +≡
  void clearnodes(subtree)
       node *subtree;
  {
    if (subtree ≡ Λ) return;
    subtree→marked = 0;
    subtree = subtree→child;
    while (subtree ≠ Λ) {      /* handle all children */
      clearnodes(subtree);
      subtree = subtree→sibling;
    }
  }
```

40. This routine marks all children of given node *subtree*. It is used in the searching routines.

```
⟨ top level routine declarations 37 ⟩ +≡
  void marknodes(subtree)
       node *subtree;
  {
    if (subtree ≡ Λ) return;
    subtree→marked = 1;
    subtree = subtree→child;
    while (subtree ≠ Λ) {      /* handle all children */
      marknodes(subtree);
      subtree = subtree→sibling;
    }
  }
```

41. I've left this in to show one way of how debugging code can be commented out (see the CWEB text for the preceeding section).

```
⟨ Debug Check for Marking 41 ⟩ ≡
#ifdef DEBUG
  printf("marking␣node:\n");
  displaylevel(subtree, 0);
#endif
```

42. *choose* selects a subnode with selector *what* from amongst the children of node *where*. The subnode selected is returned, unless there is no such sub-node, in which case Λ is returned.

```
⟨ top level routine declarations 37 ⟩ +≡
  node *choose(where, what)
       node *where;
       char what;
  {
    node *l = where→child;
    while (l ≠ Λ) {
      if (l→selector ≡ what) return l;
      l = l→sibling;
    }
    return Λ;
  }
```

43. A simple string-reading routine. We first skip blanks, then copy the remainder of the input line into a buffer. A copy of this buffer is created (using *malloc*), and a pointer to this (new) string returned.

```
⟨ top level routine declarations 37 ⟩ +≡
  char *getstr( )
  {
    char c;
    char *p;
    char str[80];
    while ((c = getchar( )) ≡ '␣') ;
    p = str;
    while (c ≠ '\n') {
      *p++ = c;
      c = getchar( );
    }
    *p = '\0';
    ungetc(c, stdin);
    return string_copy(str);
  }
```

44. A boolean procedure to respond whether a string is null or empty (return 1), otherwise return 0.

```
⟨ top level routine declarations 37 ⟩ +≡
  int emptystr(s)
      char *s;
  {
    if (s ≡ Λ) return 1;
    if (*s ≡ '\0') return 1;
    return 0;
  }
```

45. This routine inserts the new node n into the sibling list of the node pointed to by a. Ordered insertion is performed. Note that a is passed as a pointer to a pointer, i.e., it is a pointer being passed by reference, and the referenced location may be altered.

Several cases arise:

- a doesn't point anywhere. It is set to point to n.
- The selector s is less than the selector for the node pointed to by a, so n becomes the first element of the list, and a now points to it instead.
- The selectors match. If the new node has an empty name, *delete* the existing node. If the new node has a non-empty name, it replaces the existing node.
- The selector s is greater than that of the node referenced by a. Recursively call this procedure to find the right place for the new node.

⟨ top level routine declarations 37 ⟩ +≡

```
node *insert(a, n, s)
    node **a;
    node *n;
    char s;
{
  node *l, *p;
  if (*a ≡ Λ) {
    *a = n;
    return n;
  }
  if (s < (*a)→selector) {
    n→sibling = *a;
    *a = n;
    return n;
  }
  if (s ≡ (*a)→selector) {
    if (emptystr(n→primary)) {      /* discard this node */
      p = *a;
      l = p→parent;
      *a = p→sibling;
      free(n);
      free(p);
      return l;
    }
    else {
      n→sibling = (*a)→sibling;
      free(*a);
      *a = n;
      return n;
    }
  }
  return insert(&(*a)→sibling, n, s);
}
```

46. Create and add a new node to the children of node c, with selector s and primary key (name) k.

```
⟨ top level routine declarations 37 ⟩ +≡
  node *addnode(c, s, k)
       node *c;
       char s;
       char *k;
  {
    node *new;
    new = (node *) malloc(sizeof(node));
    new→selector = s;
    new→primary = k;
    new→parent = c;
    new→child = Λ;
    new→marked = 0;
    new→location = 1;
    modified = 1;      /* find position in list of children of c, and insert there */
    return insert(&c→child, new, s);
  }
```

47. Add an association k to the association list given by l.

```
⟨ top level routine declarations 37 ⟩ +≡
  void addassoc(l, k)
       list **l;
       char *k;
  {
    list *p;
    p = (list *) malloc(sizeof(list));
    p→value = k;
    p→next = *l;
    *l = p;
    modified = 1;
    return;
  }
```

48. Delete an association k from the list l. The associations in the list l are searched for a match with k to find the element to remove.

```
⟨ top level routine declarations 37 ⟩ +≡
  void delassoc(l, k)
       list **l;
       char *k;
  {
    list *p;
    p = *l;
    if (p ≡ Λ) return;
    if (patmatch(k, p→value, strlen(k))) {
      *l = p→next;
      modified = 1;
      return;
    }
    delassoc(&(p→next), k);
  }
```

49. *command* identifies whether a given character is a potential command. Returns 0 if not, 1 if it is.

```
⟨ top level routine declarations 37 ⟩ +≡
  int command(c)
      char c;
  {
    return ((c ≡ '?') ∨ (c ≡ '+') ∨ (c ≡ '=') ∨ (c ≡ '#') ∨ (c ≡ '>') ∨ (c ≡ '.') ∨ (c ≡ ':') ∨ (c ≡
        ';') ∨ (c ≡ '*') ∨ (c ≡ '^') ∨ (c ≡ '_') ∨ (c ≡ '-') ∨ (c ≡ '@') ∨ (c ≡ '\'') ∨ (c ≡ '"') ∨ (c ≡
        '\n') ∨ (c ≡ '`') ∨ (c ≡ '~') ∨ (c ≡ '<') ∨ (c ≡ ',') ∨ (c ≡ '/'));
  }
```

50. *readpattern* returns a string read from stdin, skipping leading blanks and stopping at a command character. All alphabetics are converted to lower case, to avoid case sensitivity problems. The string is null terminated.

```
⟨ top level routine declarations 37 ⟩ +≡
  char *readpattern( )
  {
    char str[80];
    char c;
    int j;
    j = 0;      /* skip leading blanks */
    while ((c = getchar( )) ≡ '␣')      /* empty */
      ;
    while (¬command(c)) {
      str[j++] = LOWER(c);
      c = getchar( );
    }
    if (c ≠ '\n') ungetc(c, stdin);
    str[j] = '\0';
    return string_copy(str);
  }
```

51. *patmatch* returns 1 if the pattern *pat* is found within the string *str*, 0 otherwise. The search is unachored, meaning that successive starting positions (up to a point such that the remaining string is still long enough) are tried in an effort to match the pattern. *len* must be set to the length of the pattern.

```
⟨ top level routine declarations 37 ⟩ +≡
  int patmatch(pat, str, len)
      char *pat, *str;
      int len;
  {
    int i;
    char c, *p;
    char s[MAXSTR];      /* first reduce given string to all lower case */
    i = 0;
    p = str;
    while (i < MAXSTR ∧ (*p ≠ '\0')) {
      c = *p++;
      s[i++] = LOWER(c);
    }
    s[i] = '\0';      /* now compare strings at all alignments (unanchored) */
    i = strlen(s);
    p = s;
    while (i ≥ len) {
      if (strncmp(p, pat, len) ≡ 0) return 1;
      i--;
      p++;
    }
    return 0;
  }
```

52. *search* looks for the pattern *pat* in the subtree given by *startpos*. Both primary keys (names of nodes) and secondary keys (associations of nodes) are searched.

```
⟨ top level routine declarations 37 ⟩ +≡
  node *search(pat, l, startpos)
      char *pat;
      int l;
      node *startpos;
  {
    node *where, *checkchild;
    list *sec;
    where = startpos;
    while (where ≠ Λ) {
      if (patmatch(pat, where→primary, l)) return where;
      sec = where→secondary;
      while (sec ≠ Λ) {
        if (patmatch(pat, sec→value, l)) return where;
        sec = sec→next;
      }     /* not this one, check all children */
      checkchild = search(pat, l, where→child);
      if (checkchild ≠ Λ) return checkchild;
      where = where→sibling;
    }
    return Λ;
  }
```

53. *nextnode* allows traversal over the tree. It does a depth first traversal, returning the child if there is one, a sibling if no child, and the next untraversed great-sibling (uncles, aunts, great-uncles, etc.) otherwise. If all this fails, return Λ.

```
⟨ top level routine declarations 37 ⟩ +≡
  node *nextnode(pos)
      node *pos;
  {
    node *where;
    where = pos;
    if (where ≡ Λ) return where;
    if (where→child ≠ Λ) return where→child;
    if (where→sibling ≠ Λ) return where→sibling;
    while (where ≠ &root) {
      where = where→parent;
      if (where→sibling ≠ Λ) return where→sibling;
    }
    return Λ;
  }
```

54. Find patterns starting at the given node. We first collect a new pattern, and reset all markings below the current node. The search starts from the current node, and traverses all nodes defined by the sequence returned by *nextnode*, until that returns NULL. All nodes found are marked and displayed, and *lastnode* is set to point to the last node found.

```
⟨ top level routine declarations 37 ⟩ +≡
  node *findpatterns(sn, pattern)
       node *sn;
       char *pattern;
  {
    node *lastnode, *thisnode, *foundnode;
    thisnode = sn;
    lastnode = Λ;
    clearnodes(sn);
    while (thisnode ≠ Λ) {
      foundnode = search(pattern, strlen(pattern), thisnode);
      if (foundnode ≠ Λ) {
        marknodes(foundnode);
        displaylevel(foundnode, 0);
        lastnode = foundnode;
      }
      thisnode = nextnode(thisnode);
    }
    if (lastnode ≡ Λ) {
      msg("not found");
      return (sn);
    }
    else return (lastnode);
  }
```

55. copy a string into a newly allocated area of memory, and return its pointer. We also define the *nextChar* macro here, which reads the next character from the *data_file*, and stores it in the variable *nc*.

```
⟨ top level routine declarations 37 ⟩ +≡
#include <ctype.h>
#define nextChar nc = fgetc (data_file)
  char *string_copy(s)
       char *s;
  {
    char *p;
    int l;
    l = strlen(s);
    p = malloc(l);
    return (char *) strcpy(p, s);
  }
```

56. *getdata* does all the work of reading the database file and parsing it.

```
⟨ top level routine declarations 37 ⟩ +≡
  void getdata(filename)
       char *filename;
  {
    int i, j, k;
    char nc;      /* next character */
    char instr[80];      /* work string */
    FILE *data_file;
    int level, prevlevel;
    char context[TREE_DEPTH];
    node *current[TREE_DEPTH];
    node *new, *temp;
    list **nextSecKey;
    list *nl;      /* initialize data structures */
    root.selector = '\0';
    root.parent = Λ;
    root.sibling = Λ;
    root.child = Λ;
    root.primary = string_copy("root");
    root.marked = 0;
    level = 0;
    prevlevel = 0;
    current[level] = &root;      /* open the character data file */
    data_file = fopen(filename, "r");
    if (data_file ≡ Λ) {
      msg("Cannot␣open␣data␣file");
      exit(1);
    }      /* initialize next character */
    nextChar;
    ⟨ Scan Data File and Process It 57 ⟩
  }
```

57. Here we read and process the input file. This routine is responsible for converting leading strings of blanks in the input file to appropriate nesting levels of the tree.

```
⟨ Scan Data File and Process It 57 ⟩ ≡
  for (i = 1; ¬feof(data_file); i++) {      /* create the new node */
    new = (node *) malloc(sizeof(node));
    new→marked = 0;      /* read blanks to pick level */
    for (j = 0; nc ≡ '␣'; j++) nextChar;
    level = j/2 + 1;
#ifdef DEBUG
    printf("level␣is␣%d␣at␣line␣%d\n", level, i);
#endif      /* read context character to pick context name */
    new→selector = nc;
    context[level] = nc;
    nextChar;
#ifdef DEBUG
    for (k = 1; k < level; k++) printf("%c.", context[k]);
    printf("%c␣", context[k]);
#endif      /* read string to pick primary key */
    while (nc ≡ '␣') nextChar;
    j = 0;
    while (nc ≠ '(' ∧ nc ≠ '{' ∧ nc ≠ '\n') {
      instr[j++] = nc;
      nextChar;
    }
    while (instr[--j] ≡ '␣')      /* do nothing */
      ;
    instr[++j] = '\0';
#ifdef DEBUG
    printf("%s\n", instr);
#endif
    new→primary = string_copy(instr);
    new→secondary = Λ;      /* read () to pick existence of secondary keys */
    if (nc ≡ '(') {      /* read secondary keys */
      nextSecKey = &new→secondary;
      while (nc ≠ ')') {
        j = 0;
        nextChar;
        nl = (list *) malloc(sizeof(list));
        while (nc ≠ ',' ∧ nc ≠ ')') {
          instr[j++] = nc;
          nextChar;
        }
        instr[j] = '\0';
        nl→value = string_copy(instr);
        nl→next = Λ;
        *nextSecKey = nl;
        nextSecKey = &nl→next;
      }
      nextChar;
    }
    new→location = 1;
    if (nc ≡ '{') {      /* read location data */
```

```
      new→location = 0;      /* explicitly specified, assume nothing */
      while (nc ≠ '}') {
        int val = 0;
        j = 0;
        nextChar;
        while (isdigit(nc)) {
          instr[j++] = nc;
          nextChar;
        }
        instr[j] = '\0';
        sscanf(instr, "%d", &val);
#ifdef DEBUG
        printf("%s␣{%d}\n", instr, val);
#endif
        new→location += val;
      }
      nextChar;
    }     /* advance to next line */
    if (nc ≡ '\n') nextChar;
    ⟨ Add New Node to Master Tree 58 ⟩
  }
```

This code is used in section 56.

58. Once the data for a node has been read and parsed, it must be added to the master data structure. Three situations arise, dependent upon the indentation used between this entry and the previous entry.

1) The entries are formatted
```
x previous entry
y curent entry
```
The "current entry" starts a new set of children of the "previous entry".

2) The entries are formatted
```
x previous entry
y current entry
```
The "current entry" is a sibling of the "previous entry".

3) The entries are formatted
```
x previous entry
y this entry (or even closer to left margin)
```
The "current entry" is a sibling of an entry appearing even earlier than the "previous entry".

⟨ Add New Node to Master Tree 58 ⟩ ≡
```
if (prevlevel < level) {      /* advance down tree */
  if (prevlevel + 1 ≠ level) msg("Multiple␣level␣jump␣in␣data␣file");
      /* insert this node on child list */
  current[prevlevel]→child = new;
  new→parent = current[prevlevel];
}
else {      /* next sibling of current at this level */
  current[level]→sibling = new;
  new→parent = current[level]→parent;
}
new→sibling = Λ;
new→child = Λ;
current[level] = new;
prevlevel = level;
```
This code is used in section 57.

59. The procedure *getfilename.c* interrogates the environment variable `ASSOCDB` to see where the database file is located. This environment variable should contain an absolute path name to the database file. If it is empty, then we assume a file called *files* in the local directory.

⟨ top level routine declarations 37 ⟩ +≡
```
char *getfilename( )
{
  char *f;
  extern char *getenv( );
  f = getenv("ASSOCDB");
  if (f ≡ Λ) f = string_copy("files");
  return f;
}
```

60. *msg* is very trivial: it just prints a message!

```
⟨ top level routine declarations 37 ⟩ +≡
  void msg(str)
       char *str;
  {
    printf("%s\n", str);
  }
```

61. *putdata* is responsible for outputting the revised tree, in flattened form. It takes a (string) file name as parameter, and opens and closes the file of that name as part of its operation. It returns 1 if the data not written for any reason, and 0 if the output operation was successful.

The global flag *modified* is checked to see whether the tree has indeed changed. If it hasn't, no output takes place, and a "no write" return taken.

To actually write the data, a separate recursive routine called *traverse* is invoked. This is passed the first node immediately below the root, as the root node itself is implicit in the flattened file representation.

```
⟨ top level routine declarations 37 ⟩ +≡
  int putdata(filename)
       char *filename;
  {
    FILE *data_file;
    if (¬modified) return (1);
    data_file = fopen(filename, "w");
    if (data_file ≡ Λ) {
      msg("Cannot␣open␣data␣file");
      return (1);
    }
    traverse(root.child, 0, data_file);
    fclose(data_file);
    return (0);
  }
```

62. *traverse* does a depth first traversal of the data tree. Starting with the tree indicated by t, which is at depth l, lines of output are written to the output file f. Each line consists of $2 * l$ spaces, followed by the node selector, node name (the *primary* field), and then any associations attached to the node (via the *secondary* field), and finally the location (if present).

After the current node is printed, any children are then treated the same way through a recursive call at depth $l + 1$, and then all siblings are treated through a further recursive call at depth l.

```
⟨ top level routine declarations 37 ⟩ +≡
  void traverse(t, l, f)
       node *t;
       int l;
       FILE *f;
  {
    int i;
    list *p;
    char s;
    ⟨ Null Node Recursive Termination 63 ⟩
    ⟨ Indicate Depth of Node in Tree 64 ⟩
    ⟨ Print Selector and Primary Names 65 ⟩
    ⟨ Print all Associations 66 ⟩
    ⟨ Print the Location 67 ⟩
    traverse(t→child, l + 1, f);
    traverse(t→sibling, l, f);
  }
```

63. If the node to print is empty, print nothing: return.

```
⟨ Null Node Recursive Termination 63 ⟩ ≡
  if (t ≡ Λ) return;
```

This code is used in section 62.

64. Print $2 * l$ blanks to indicate the depth l of the current node in the tree.

```
⟨ Indicate Depth of Node in Tree 64 ⟩ ≡
  for (i = 0; i < l; i++) fprintf(f, "␣␣");
```

This code is used in section 62.

65. Print the selector character and the primary Node Name.

```
⟨ Print Selector and Primary Names 65 ⟩ ≡
  fprintf(f, "%c␣%s", t→selector, t→primary);
```

This code is used in section 62.

66. Print the list of associations.

```
⟨ Print all Associations 66 ⟩ ≡
  p = t→secondary;
  s = '(';
  while (p ≠ Λ) {
    fprintf(f, "%c%s", s, p→value);
    s = ',';
    p = p→next;
  }
  if (s ≡ ',') fprintf(f, ")");
```

This code is used in section 62.

67. Print the location of a file.

```
⟨ Print the Location 67 ⟩ ≡
  s = '{';
  if (t→location ≠ 1) {
    fprintf(f, "%c%d", s, t→location);
    s = ',';
  }
  if (s ≡ ',') fprintf(f, "}");
  fprintf(f, "\n");
```

This code is used in section 62.

68. References.

1. Introduction to the CR Classification System [1991] version, Computing Reviews, pp4-44, January 1994.

2. D.E.Knuth and S.Levy, *The CWEB System of Structured Documentation*, Addison-Wesley, 1994.

69. Index.

a: 45.
addassoc: 15, 30, 47.
addnode: 17, 30, 46.
assoc: 6.
ASSOCDB: 6, 59.
bak: 33.
browse: 10, 12, 13, 36.
c: 13, 43, 46, 49, 50, 51.
checkchild: 52.
child: 11, 38, 39, 40, 42, 46, 52, 53, 56, 58, 61, 62.
choose: 16, 23, 30, 42.
clearnodes: 13, 30, 39, 54.
command: 30, 49, 50.
context: 56, 57.
current: 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 38, 56, 58.
data_file: 55, 56, 57, 61.
DEBUG: 41, 57.
delassoc: 20, 30, 48.
displayassocs: 30, 37, 38.
displaylevel: 13, 19, 23, 25, 30, 38, 41, 54.
emptystr: 30, 44, 45.
exit: 10, 56.
f: 59, 62.
fclose: 61.
feof: 57.
fgetc: 55.
filename: 56, 61.
files: 59.
findpatterns: 18, 30, 54.
fn: 10, 33, 34, 35.
fnb: 33, 34, 35.
fopen: 56, 61.
foundnode: 13, 17, 19, 22, 54.
fprintf: 64, 65, 66, 67.
free: 18, 19, 25, 45.
full: 38.
getchar: 13, 14, 16, 17, 23, 43, 50.
getdata: 10, 30, 56.
getenv: 59.
getfilename: 10, 30, 59.
getstr: 15, 17, 20, 30, 43.
i: 51, 56, 62.
insert: 30, 45, 46.
instr: 56, 57.
isdigit: 57.
j: 50, 56.
k: 46, 47, 48, 56.
l: 38, 42, 45, 47, 48, 52, 55, 62.
lastnode: 13, 16, 19, 23, 54.
len: 51.
level: 56, 57, 58.
link: 33, 35.
list: 11, 37, 38, 47, 48, 52, 56, 57, 62.
list_struct: 11.
location: 11, 21, 38, 46, 57, 67.
LOWER: 29, 50, 51.
ls: 37, 38.
main: 10.
makebackup: 10, 12, 33.
malloc: 30, 43, 46, 47, 55, 57.
marked: 11, 18, 19, 39, 40, 46, 56, 57.
marknodes: 30, 40, 54.
MAXSTR: 27, 51.
modified: 21, 28, 46, 47, 48, 61.
msg: 19, 22, 23, 25, 26, 30, 54, 56, 58, 60, 61.
n: 45.
nc: 55, 56, 57.
new: 46, 56.
next: 11, 37, 47, 48, 52, 57, 66.
nextChar: 55, 56, 57.
nextnode: 19, 22, 30, 53, 54.
nextSecKey: 56, 57.
nl: 56, 57.
node: 11, 13, 28, 30, 37, 38, 39, 40, 42, 45, 46, 52, 53, 54, 56, 57, 62.
node_struct: 11.
p: 37, 38, 43, 45, 47, 48, 51, 55, 62.
parent: 11, 24, 38, 45, 46, 53, 56, 58.
pat: 51, 52.
patmatch: 30, 48, 51, 52.
pattern: 13, 18, 19, 22, 25, 28, 54.
pos: 53.
prevlevel: 56, 58.
primary: 11, 38, 45, 46, 52, 56, 57, 62, 65.
printf: 14, 37, 38, 41, 57, 60.
prkey: 13, 15, 17, 20.
putdata: 10, 30, 61.
PYR: 31.
readpattern: 18, 19, 25, 30, 50.
restorebackup: 10, 12, 35.
result: 10.
rmbackup: 12, 34.
root: 13, 19, 22, 23, 24, 25, 28, 38, 53, 56, 61.
s: 37, 38, 44, 45, 46, 51, 55, 62.
scanf: 21.
search: 19, 22, 25, 30, 52, 54.
sec: 52.
secondary: 11, 15, 20, 37, 52, 57, 62, 66.
sel: 13, 17.
selector: 11, 38, 42, 45, 46, 56, 57, 65.
sibling: 11, 38, 39, 40, 42, 45, 52, 53, 56, 58, 62.

sn: 54.
sp: 38.
sscanf: 57.
startpos: 52.
stdin: 14, 43, 50.
stk: 38.
str: 43, 50, 51, 60.
strcat: 33, 34, 35.
strcpy: 33, 34, 35, 55.
string: 7.
string_copy: 30, 43, 50, 55, 56, 57, 59.
strlen: 19, 22, 25, 48, 51, 54, 55.
strncmp: 51.
subtree: 39, 40, 41.
`SUN`: 31.
t: 62.
temp: 56.
thisnode: 54.
traverse: 30, 61, 62.
`TREE_DEPTH`: 27, 38, 56.
ungetc: 14, 43, 50.
unlink: 33, 34, 35.
`UPTOLOW`: 29.
val: 57.
value: 11, 37, 47, 48, 52, 57, 66.
what: 42.
where: 42, 52, 53.

⟨ Add New Node to Master Tree 58 ⟩ Used in section 57.
⟨ Add an Association to the Current Node 15 ⟩ Used in section 13.
⟨ Command Description 7 ⟩ Cited in section 6.
⟨ Debug Check for Marking 41 ⟩
⟨ Down to Given Node 16 ⟩ Used in section 13.
⟨ Edit a Child Node 17 ⟩ Used in section 13.
⟨ Find All Patterns 18 ⟩ Cited in section 19. Used in section 13.
⟨ Find Pattern in Previously Matched Nodes 19 ⟩ Used in section 13.
⟨ Global Data Structures 11 ⟩ Used in section 27.
⟨ Global Declarations 27 ⟩ Used in section 9.
⟨ Global Macro Definitions 29 ⟩ Used in section 27.
⟨ Global Procedure Templates 30 ⟩ Used in section 27.
⟨ Global System Dependencies 31 ⟩ Used in section 27.
⟨ Global Variables 28 ⟩ Used in section 27.
⟨ Go To Absolute Path Node 23 ⟩ Used in section 13.
⟨ Go Up To Parent Node 24 ⟩ Used in section 13.
⟨ I/O routine declarations 33, 34, 35 ⟩ Used in section 12.
⟨ Indicate Depth of Node in Tree 64 ⟩ Used in section 62.
⟨ Kill an Association 20 ⟩ Used in section 13.
⟨ Main Program 10 ⟩ Used in section 9.
⟨ Next Occurrence of Pattern 22 ⟩ Used in section 13.
⟨ Null Node Recursive Termination 63 ⟩ Used in section 62.
⟨ Pattern Match 25 ⟩ Used in section 13.
⟨ Print Selector and Primary Names 65 ⟩ Used in section 62.
⟨ Print all Associations 66 ⟩ Used in section 62.
⟨ Print the Location 67 ⟩ Used in section 62.
⟨ Procedure Declarations 12 ⟩ Used in section 9.
⟨ Scan Data File and Process It 57 ⟩ Used in section 56.
⟨ Set Location 21 ⟩ Used in section 13.
⟨ Supply Help 14 ⟩ Used in section 13.
⟨ Treat anything not recognized as Illegal 26 ⟩ Used in section 13.
⟨ top level routine declarations 37, 38, 39, 40, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 59, 60, 61, 62 ⟩
Cited in section 12. Used in section 12.
⟨ *browse* declaration 13 ⟩ Used in section 12.

ASSOC

	Section	Page
Background to this Program	1	1
User Notes	6	3
Top Level Description	9	4
Global Declarations	27	11
I/O Routine Declarations	32	13
Top-Level Routines	36	14
References	68	31
Index	69	32