ID database utilities

Programs for simple, fast, high-capacity cross-referencing
for version 3.2

Greg McGary
Tom Horsley

1 Introduction

An *ID database* is a binary file containing a list of file names, a list of tokens, and a sparse matrix indicating which tokens appear in which files.

With this database and some tools to query it (described in this manual), many text-searching tasks become simpler and faster. For example, you can list all files that reference a particular `#include` file throughout a huge source hierarchy, search for all the memos containing references to a project, or automatically invoke an editor on all files containing references to some function or variable. Anyone with a large software project to maintain, or a large set of text files to organize, can benefit from the ID utilities.

Although the name 'ID' is short for 'identifier', the ID utilities handle more than just identifiers; they also treat other kinds of tokens, most notably numeric constants, and the contents of certain character strings. Thus, this manual will use the word *token* as a term that is inclusive of identifiers, numbers and strings.

There are several programs in the ID utilities family:

'`mkid`' scans files for tokens and builds the ID database file.

'`lid`' queries the ID database for tokens, then reports matching file names or matching lines.

'`fid`' lists all tokens recorded in the database for given files, or tokens common to two files.

'`fnid`' matches the file names in the database, rather than the tokens.

'`xtokid`' extracts raw tokens—helps with testing of new '`mkid`' scanners.

In addition, the ID utilities have historically provided several query programs which are specializations of '`lid`':

'`gid`' (alias for '`lid -R grep`') lists all lines containing the requested pattern.

'`eid`' (alias for '`lid -R edit`') invokes an editor on all files containing the requested pattern, and if possible, initiates a text search for that pattern.

'`aid`' (alias for '`lid -ils`') treats the requested pattern as a case-insensitive literal substring.

Please report bugs to '`bug-gnu-utils@gnu.ai.mit.edu`'. Remember to include the version number, machine architecture, input files, and any other information needed to reproduce the bug:

your input, what you expected, what you got, and why it is wrong. Diffs are welcome, but please include a description of the problem as well, since this is sometimes difficult to infer. See section "Bugs" in *GNU CC*.

2 Quick Start Procedure

Unpack the distribution.

Type '`./configure`'

Type '`make`'

Type '`make install`' as a user with the appropriate privileges (e.g., '`bin`' or perhaps even '`root`').

Type '`cd /usr/include; mkid`' to build an ID database covering all of the system header files.

Type '`lid FILE`', then '`gid strtok`', then '`aid stdout`'.

You have just built, installed and used the most common commands of the GNU ID utilities. If you ever need help remembering which system header files contain a particular declaration, or reference a particular symbol, you'll want to keep the ID file you built in '`/usr/include`' for later use. If your working directory is elsewhere at the time, simply provide the '`-f /usr/include`' option to '`lid`' (see Section 3.2 [Reading options], page 4).

3 Common command-line options

Certain options, and regular expression syntax, are shared by various groupings of the ID utilities. We describe these in the sections below, rather than repeating them for each program.

3.1 Options Common to All Programs

'`--help`' Print a usage message listing all available options, then exit successfully.

'`--version`'
Print the version number, then exit successfully.

3.2 Options for Programs that Read ID Databases

'`-f` *filename*'
'`--file=`*filename*'
Filename is the ID database to read when processing queries. At present, only a single '`--file`' option is processed, but in future releases, more than one ID database may be named on the command line.

'`$IDPATH`'
'`IDPATH`' is an environment variable that contains a colon-separated list of ID database names. If this variable is present, and no '`--file`' options are presented on the command line, the ID databases named in '`IDPATH`' are implied.[1]

If no ID databases are specified either on the command line or via the '`IDPATH`' environment variable, then the ID utilities search for a file named '`ID`' in the current working directory, and then in successive parent directories.

3.3 Options for Programs that Write ID Databases

[1] At present, this feature is fully implemented, since only the first of a list of ID database names is processed.

‘`-o` *filename*’
‘`--output=`*filename*’
: The ‘`--output`’ option names the file in which to write a new ID database. If no ‘`--output`’ (or ‘`--file`’) option is present, an output file named ‘`ID`’ is implied.

‘`-f` *filename*’
‘`--file=`*filename*’
: This is a synonym for ‘`--output`’

3.4 Options for Programs that Walk File and Directory Trees.

The programs ‘`mkid`’ and ‘`xtokid`’ accept the names of files and directories on the command line. Files are scanned if there is a scanner available and enabled for the file’s source language. Directories are recursively descended, searching for files whose names match the rules listed in the *language map* file (see Section 3.6.1 [Language map], page 7).

The following option controls the file tree walker:

‘`-p` *names*’
‘`--prune=`*names*’
: One or more file or directory names may appear in *names*. The file tree walker will stop short at these files and directories and their contents will not be scanned.

3.5 Options for Programs that List File Names

The programs ‘`lid`’ and ‘`fnid`’ can print lists of file names as the result of queries. The following option controls how these lists are formatted:

‘`-S` *style*’
‘`--separator=`*style*’
: *Style* may be one of ‘`braces`’, ‘`space`’ or ‘`newline`’.

  The *style* of ‘`braces`’ means that file names with common directory prefix and common suffix are printed using the shell’s brace notation in order to compress the output. For example, ‘`../src/foo.c ../src/bar.c`’ can be printed in brace notation as ‘`../src/{foo,bar}.c`’.

  The *style*s of ‘`space`’ and ‘`newline`’ mean that file names are separated spaces or by newlines, respectively.

If the list of files is being printed on a terminal, brace notation is the default. If not, file names are separated by spaces if the *key* is included in the output, and by newlines the *key style* is '`none`' (see Chapter 5 [lid invocation], page 13).

3.6 Options for Programs that Scan Source Files

'`mkid`' and '`xtokid`' walk file trees, select source files by name, and extract tokens from source files. They accept the following options:

'`-m` *mapfile*'
'`--lang-map=`*mapfile*'
: *mapfile* contains rules for determining the source languages from file names. See Section 3.6.1 [Language map], page 7

'`-i` *languages*'
'`--include=`*languages*'
: The '`--include`' option names *languages* whose source files should be scanned and incorporated into the ID database. By default, all languages known to the ID utilities are enabled.

'`-x` *languages*'
'`--exclude=`*languages*'
: The '`--exclude`' option names *languages* whose source files should *not* be scanned. The default list of excluded languages is empty. Note that only one of '`--include`' or '`--exclude`' may be specified on the command line for a single run.

'`-l` *language*`:`*options*'
'`--lang-option=`*language*`:`*options*'
: Language-specific scanners also accept options. *Language* denotes the desired scanner, and *option* are the command-line options that should be passed through to it. For example, to pass the *-x –coke-bottle* options to the scanner for the language *swizzle*, pass this: *-l swizzle:*"*-x –coke-bottle*", or this: *-lang-option=swizzle:*"*-x –coke-bottle*", or this: *-l swizzle-x -l swizzle:–coke-bottle*. Use the '`--help`' option to see the command-line option summary for

To determine which tokens to extract from a file and store in the database, '`mkid`' calls a *scanner*; we say a scanner *recognizes* a particular language. Scanners for several languages are built-in to '`mkid`'; you can add your own scanners as well, as explained in Section 3.6.5 [Defining scanners], page 10.

The ID utilities determine which scanner to use for a particular file by consulting the language-map file. Scanners for several are already built-in to the ID utilities. You can see which languages have built-in scanners, and examine their language-specific options by invoking '`mkid --help`' or '`xtokid --help`'.

3.6.1 Mapping file names to source languages

The file '`id-lang.map`', installed by default in '`$(prefix)/share/id-lang.map`', contains rules for mapping file names to source languages. Each rule comprises three parts: a shell *glob* pattern, a language name, and language-specific scanner options.

The special pattern '`**`' denotes the default source language. This is the language that's assigned to file names that don't match any other pattern.

The special pattern '`***`' should be followed by a file name. The named file should contain more language-map rules and is included at this point.

The order in which rules are presented in a language-map file is significant. This order influences the order in which files are displayed as the result of queries. For example, the distributed language-map file places all rules for C *.h* files ahead of *.c* files, so that in general, declarations will precede definitions in query output. The same thing is done for C++ and its many different source file name extensions.

Here is a pared-down version of the '`id-lang.map`' file distributed with the ID utilities:

```
# Default language
** IGNORE # Although this is listed first,
# the default language pattern is
# logically matched last.

# Backup files
*~ IGNORE
*.bak IGNORE
*.bk[0-9] IGNORE

# SCCS files
[sp].* IGNORE

# list header files before code files
*.h C
```

```
*.h.in C
*.H C++
*.hh C++
*.hpp C++
*.hxx C++

# list C 'meta' files next
*.l C
*.lex C
*.y C
*.yacc C

# list C code files after header files
*.c C
*.C C++
*.cc C++
*.cpp C++
*.cxx C++

# list assembly language after C
*.[sS] asm --comment=;
*.asm asm --comment=;

# [nt]roff
*.[0-9] roff
*.ms roff
*.me roff
*.mm roff

# TeX and friends
*.tex TeX
*.ltx TeX
*.texi texinfo
*.texinfo texinfo
```

3.6.2 C/C++ Language Scanner

The C scanner is the most commonly used. Files that match the glob pattern '`*.h`', '`*.c`', as well as '`yacc`' files that match '`*.y`' or '`*.yacc`', and '`lex`' files that match '`*.l`' or '`*.lex`', are processed with this scanner.

Scanner-specific options (Note, these options are presented *without* the required '`-l`' or '`--lang-option=`' prefix):

‘`-k` *character-class*’
‘`--keep=`*character-class*’
: Consider the characters in *character-class* as valid constituents of identifier names. For example, if you are indexing C code that contains ‘`$`’ in some of its identifiers, you can include these by using ‘`--lang-option=C:--keep=$`’, or ‘`-l C:"-k $"`’ (if you don’t like to type so much).

‘`-i` *character-class*’
‘`--ignore=`*character-class*’
: x mkiConsider the characters in *character-class* as valid constituents of identifier names, but discard all tokens containing these characters. For example, if some C code has identifiers containing ‘`$`’, but you don’t want these cluttering up your ID database, use ‘`--lang-option=C:--ignore=$`’, or the terser equivalent ‘`-l C:"-i $"`’.

‘`-u`’
‘`--strip-underscore`’
: Strip one leading underscore from C identifiers encapsulated as character strings. This option is useful if you are indexing C code that contains symbol-table name strings for systems that prepend an underscore to external symbols. By default, the leading underscore is retained.

3.6.3 Assembly Language Scanner

Assembly languages use a variety of commenting conventions, and allow a variety of special characters to *dirty up* local symbols, preventing name space conflicts with symbols defined by higher-level languages. Also, some compilation systems prepend an underscore to external symbols. The options listed below are designed to address these differences.

‘`-c` *character-class*’
‘`--comment=`*character-class*’
: The characters in *character-class* are considered left delimiters for comments that extend until the end of the current line.

‘`-k` *character-class*’
‘`--keep=`*character-class*’
: Consider the characters of *character-class* as valid constituents of identifier names. For example, if you are indexing assembly code that prepends ‘`.`’ to assembler directives, and prepends ‘`%`’ to register names, you can keep these characters in the tokens by specifying ‘`--lang-option=asm:--keep=.%`’, or ‘`-l asm:"-k .%"`’.

‘`-i` *character-class*’
‘`--ignore=`*character-class*’
: Consider the characters of *character-class* as valid consituents of identifier names, but discard all tokens containing these characters. For example, if you don’t want to clutter your ID database with assembler directives that begin with a leading ‘`.`’ or with assembler labels that contain ‘`@`’, use ‘`--lang-option=asm:--ignore=.@`’, or ‘`-l asm:"-i .@"`’.

‘`-u`’
‘`--strip-underscore`’
: Strip one leading underscore from identifiers. This option is useful if your compilation system prepends an underscore to external symbols. By stripping the underscore, you can canonicalize such names and bring them into conformance the way they are expressed in the C language. By default, the leading underscore is retained.

‘`-n`’
‘`--no-cpp`’
: Do not recognize C preprocessor directives. By default, such lines are handled in the same way as they are by the C language scanner.

3.6.4 Text Scanner

The plain text scanner is intended for human-language documents, or as the scanner of last resort for files that have no scanner that is more specific. It is customizable to the extent that character classes can be designated as token constituents or as token delimiters. The default token constituents are the alpha-numerics; all other characters are considered token delimiters.

‘`-i` *character-class*’
‘`--include=`*character-class*’
: Include characters belonging to *character-class* in tokens.

‘`-x` *character-class*’
‘`--exclude=`*character-class*’
: Exclude characters belonging to *character-class* from tokens, i.e., treat them as token delimiters.

3.6.5 Defining New Scanners in the Source Code

To add a new scanner in source code, you should add a new section to the file ‘`scanners.c`’. It might be easiest to clone one of the existing scanners and modify it as necessary. For the

hypothetical language *foo*, you must define the functions `get_token_foo`, `parse_args_foo`, `help_me_foo`, as well as the tables `long_options_foo` and `args_foo`. If your scanner is modelled after one of the existing scanners, you'll also need a character-attribute table `ctype_foo`.

This is not a terribly difficult programming task, but it requires recompiling and installing the new version of '`mkid`' and '`xtokid`'. You should use '`xtokid`' to test the operation of the new scanner.

Once these functions and tables are ready, add function prototypes and an entry to to the `languages_0` table near the beginning of the file.

Be warned that the existing scanners are built for speed, not elegance or readability. You might wish to create a new scanner that's easier to read and understand if you don't feel that speed is so important.

4 ‘mkid’: Creating an ID Database

‘mkid’ builds an ID database. It accepts the names of files and/or directories on the command line, selects files that have an enabled scanner, then extracts and stores tokens from those files. The resulting ID database is architecture- and byte-order-independent so it can be shared among all systems.

The primary virtues of ‘mkid’ are speed and high capacity. The size of the source trees it can index is limited only by available system memory. ‘mkid’’s indexing algorithm is very space-efficient and exhibits excellent locality-of-reference, and so is capable of operating with a working-set size that is only half the size of its virtual address space. A typical UNIX-like operating system with 16 megabytes of system memory should be able to build an ID database covering approximately 12,000-14,000 source files totalling approximately 50–100 Megabytes. A 66 Mhz 486 computer can build such a large ID database in approximately 10-15 minutes.

In a future release, ‘mkid’ will be able to incrementally update an ID database much faster than it can build one from scratch. Until this feature becomes available, it might be a good idea to schedule a ‘cron’ job to regularly update large ID databases during off-hours.

‘mkid’ writes the ID file, therefore it accepts the ‘--output’ (and ‘--file’) options as described in Section 3.3 [Writing options], page 4. ‘mkid’ extracts tokens from source files, therefore it accepts the ‘--lang-map’, ‘--include’, ‘--exclude’, and ‘--lang-option’ options, as well as the language-specific scanner options, all of which are described in Section 3.6 [Extraction options], page 6. ‘mkid’ walks file trees, therefore it handles file and directory names on its command line and the ‘--prune’ option as described in Section 3.4 [Walker options], page 5.

In addition, ‘mkid’ accepts the following command-line options:

‘-s’
‘--statistics’
: ‘mkid’ reports statistics about resource usage at the end of its run.

‘-v’
‘--verbose’
: ‘mkid’ reports statistics about each file as it is scanned, and about the resource usage of its indexing algorithm at regular intervals.

5 `lid`: Querying an ID Database by Token

The '`lid`' program accepts *patterns* on the command line which it matches against the tokens stored in an ID database. The interpretation of a *pattern* is determined by the makeup of the *pattern* string itself, or can be overridden by command-line options. If a *pattern* contains regular expression meta-characters, it is used to perform a regular-expression substring search. If no such meta-characters are present, *pattern* is used to perform a literal word search. (By default, all searches are sensitive to alphabetic case.) If no *pattern* is supplied on the command line, '`lid`' lists every entry in the ID database.

'`lid`' reads the ID database, therefore it accepts the '`--file`' option, and consults the '`IDPATH`' environment variable, as described in Section 3.2 [Reading options], page 4. '`lid`' lists file names, therefore it accepts the '`--separator`' option, as described in Section 3.5 [File listing options], page 5.

In addition, `lid` accepts the following command-line options:

'`-i`'
'`--ignore-case`'
: Ignoring differences in alphabetic case between the *pattern* and the tokens in the ID database.

'`-l`'
'`--literal`'
: Match *pattern* as a literal string. Use this option if *pattern* contains regular-expression meta-characters, but you don't wish to perform a regular-expression search.

'`-r`'
'`--regexp`'
: Match *pattern* as an *extended* regular expression[1]. Use this option if no regular-expression expression meta-characters are present in *pattern*, but you wish to force a regular-expression search (note: in this case, a *literal substring* search might be faster).

'`-w`'
'`--word`'
: Match *pattern* using a word-delimited (non substring) search. This is the default for literal searches.

[1] Extended regular expressions are the same as those accepted by '`egrep`'.

'`-s`'
'`--substring`'

Match *pattern* using a substring (non word-delimited) search. This is the default for regular expression searches.

'`-k` *style*'
'`--key=`*style*'

Style can be one of '`token`', '`pattern`' or '`none`'. This option controls how the subject of the query is presented. This is best illustrated by example:

```
$ lid --key=token '^dest.'
destaddr        libsys/memcpy.c
destination     libsys/regex.c
destlst         libsys/rx.c
destpos         libsys/rx.c
destset         libsys/rx.h libsys/rx.c

$ lid --key=pattern '^dest.'
^dest.          libsys/rx.h libsys/{memcpy,regex,rx}.c

$ lid --key=none '^dest.'
libsys/rx.h libsys/{memcpy,regex,rx}.c
```

When '`--key`' is either '`token`' or '`pattern`', the first column of output is a *token* or *pattern*, respectively. When '`--key`' is '`none`', neither of these is printed, and the file name list begins immediately. The default is '`token`'.

'`-R` *style*'
'`--result=`*style*'

Style can be one of '`filenames`', '`grep`', '`edit`' or '`none`'. This option controls how the value associated with the query's *key* presented. When *style* is '`filenames`', a list of file names is printed (this is the default). When *style* is '`grep`', the lines that match *pattern* are printed in the same format as '`egrep -n`'. When *style* is '`edit`', the file names are passed to an editor, and if possible *pattern* is passed as an initial search string (see Section 5.3 [eid invocation], page 16). When *style* is '`none`', the file names are not processed in any way. This can be useful if you wish to see what tokens match a *pattern*, but don't care about where they reside.

'`-d`'
'`-o`'
'`-x`'

These options may be used in any combination to specify the radix of numeric matches. '`-d`' allows matching on decimal numbers, '`-o`' on octal numbers, and '`-x`' on hexadecimal numbers. Any combination of these options may be used. The default is to match all three radixes.

'-F *range*'
'--frequency=*range*'

Match tokens whose occurrence count falls in *range*. *Range* may be expressed as a single number *n*, or as a range *n*..*m*. Either limit of the range may be omitted (e.g., ..*m*, or *n*...). If the lower limit *n* is omitted, it defaults to 1. If the upper limit is omitted, it defaults in the present implementation to 65535, the maximum value of an unsigned 16-bit integer.

Particularly useful queries are 'lid -F1', which helps locate identifiers that are defined but never used, or are used but never defined. Similarly, lid -F2 can help find functions that possess a prototype declaration and a definition, but are never called.

'-a *number*'
'--ambiguous=*number*'

List identifiers (not numbers) that are ambiguous for the first *number* characters. This feature might be in useful when porting programs to ancient pea-brained compilers that don't support long identifier names. However, the best long-term option is to set such systems on fire.

5.1 Aliases for Specialized 'lid' Queries

Historically, the ID utilities have provided several query interfaces which are specializations of lid (see Chapter 5 [lid invocation], page 13).

'gid' (alias for 'lid -R grep') lists all lines containing the requested pattern.

'eid' (alias for 'lid -R edit') invokes an editor on all files containing the requested pattern, and optionally initiates a text search for that pattern.

'aid' (alias for 'lid -ils') treats the requested pattern as a case-insensitive literal substring.

5.2 GNU Emacs query interface

The id-utils source distribution comes with a file 'id-utils.el', which defines a GNU Emacs interface to gid. To install it, put 'id-utils.el' somewhere that Emacs will find it (i.e., in your load-path) and put

```
(autoload 'gid "gid" nil t)
```

in one of Emacs' initialization files, e.g., '`~/.emacs`'. You will then be able to use *M-x gid* to run the command.

The `gid` function prompts you with the word around point. If you want to search for something else, simply delete the line and type the pattern of interest.

The function then runs the `gid` program in a '`*compilation*`' buffer, so the normal `next-error` function can be used to visit all the places the identifier is found (see section "Compilation" in *The GNU Emacs Manual*).

5.3 `eid`: Invoking an Editor on Query Results

'`lid -R edit`' is an editing interface for the ID utilities that is most commonly used with '`vi`'. Emacs users should use the interface defined in `id-utils.el` (see Section 5.2 [Emacs gid interface], page 15). The ID utilities include an alias called '`eid`', and for the sake of brevity, we'll use this alias for the remainder of this section. '`eid`' performs a '`lid`'-style, then asks if you wish to edit the files. If your query yields more than one line of output, you will be prompted after each line. This is the prompt you'll see:

```
Edit? [y1-9^S/nq]
```

You may respond with:

'`y`' Edit all files listed.

'`1...9`' Edit all files starting at the n+1'st file.

'*/string* or *CTRL-S regexp*'
Search into the file list, and begin editing with the first file name that matches the regular expression *regexp*.

'`n`' Don't edit any files. If another line of query output is pending, advance to that line, for which another '`Edit?`' prompt will appear.

'`q`' Quit—don't edit any files, and don't process any more lines of query output.

Here is an example:

```
prompt$ eid FILE \^print
FILE           {ansi2knr,fid,filenames,idfile,idx,lid,misc,...}.c
```

```
Edit? [y1-9^S/nq] n
^print          {ansi2knr,fid,getopt,getopt1,lid,mkid,regex,scanners}.c
Edit? [y1-9^S/nq] 2
```

This will start editing at 'getopt'.c.

eid invokes the editor defined by the environment variable 'VISUAL'. If 'VISUAL' is undefined, it uses the environment variable 'EDITOR' instead. If 'EDITOR' is undefined, it defaults to 'vi'. It is possible for 'eid' to pass the editor an initial search pattern so that your cursor will immediately alight on the token of interest. This feature is controlled by the following environment variables:

'EIDARG' A printf(3) format string for the editor argument to search for the matching token. For vi, this should be '+/%s/'.

'EIDLDEL' The regular-expression meta-character(s) for delimiting the beginning of a word (the "eid' Left DELimiter'). eid inserts this in front of the matching token when a word-search is desired. For 'vi', this should be '\<'.

'EIDRDEL' The regular-expression meta-character(s) for delimiting the end of a word (the "eid' Right DELimiter'). eid inserts this in end of the matching token when a word-search is desired. For 'vi', this should be '\>'.

6 `fid`: Listing a file's tokens

'`fid`' prints the tokens found in a given file. If two file names are passed on the command line, '`fid`' prints the tokens that are common to both files (i.e., the *set intersection* of the two token sets).

'`lid`' reads the ID database, therefore it accepts the '`--file`' option, and consults the '`IDPATH`' environment variable, as described in Section 3.2 [Reading options], page 4.

If the standard output is attached to a terminal, the printed tokens are separated by spaces. Otherwise, the tokens are printed one per line.

7 `fnid`: Looking up filenames

`fnid` queries the list of file names stored in the ID database. It accepts shell *wildcard* patterns on the command line. If no pattern is supplied, '`*`' is implied. '`fnid`' prints the file names that match the given patterns.

`fnid` prints file names, and as such accepts the '`--separator`' option as described in Section 3.5 [File listing options], page 5.

For example, the command:

```
fnid \*.c
```

lists all the '`.c`' files in the database. (The '\' here protects the '`*`' from being expanded by the shell.)

8 ‘xtokid’: Testing Language Scanners

‘xtokid’ accepts the names of files and/or directories on the command line, then extracts and prints a stream of tokens from those files for which it has a valid, enabled scanner. This is useful primarily for debugging new ‘mkid’ scanners (see Section 3.6.5 [Defining scanners], page 10).

‘xtokid’ extracts tokens from source files, therefore it accepts the ‘--lang-map’, ‘--include’, ‘--exclude’, and ‘--lang-option’ options, as well as the language-specific scanner options, all of which are described in Section 3.6 [Extraction options], page 6. ‘xtokid’ walks file trees, therefore it handles file and directory names on its command line and the ‘--prune’ option as described in Section 3.4 [Walker options], page 5.

The name ‘xtokid’ indicates that it is the “eXtract TOKens ID utility”.

9 Past and Future

Greg McGary conceived of the ideas behind the ID utilities when he began working on the Unix kernel in 1984. He needed a navigation tool to help him find his way around the expansive, unfamiliar landscape. The first `id-utils`-like tools were shell scripts, and produced an ASCII database that looks much like the output of '`lid ".*"`'. It took over an hour on a VAX 11/750 to build a database for a 4.1BSD derived kernel. The first version of '`lid`' used the UNIX system utility `look`, modified to handle very long lines.

In 1986, Greg rewrote the shell scripts in C to improve performance. Build times for the ID file were shortened by an order of magnitude. The ID utilities were first posted to '`comp.sources.unix`' in September 1987 under the name `id`.

Over the next few years, several versions diverged from the original source. Tom Horsley at Harris Computer Systems Division stepped forward to take over maintenance and integrated some of the fixes from divergent versions. A first release of the renamed '`mkid`' version 2 was posted to '`alt.sources`' near the end of 1990. At that time, Tom wrote a Texinfo manual with the encouragement the net community. (Tom especially thanks Doug Scofield and Bill Leonard whom he dragooned into helping poorfraed and edit—they found several problems in the initial version.) Karl Berry revamped the manual for Texinfo style, indexing, and organization in 1995.

In January 1995, Greg McGary reemerged as the primary maintainer and launched development of '`mkid`' version 3, whose primary new feature is an efficient algorithm for building databases that is linear in both time and space over the size of the input text. (The old algorithm was quadratic in space so it was incapable of handling very large source trees.) For the first time, the code was released under the GNU Public License.

In June 1996, the package was renamed again to `id-utils` and was released for the first time under FSF copyright as part of the GNU system. All programs had their command-line arguments completely revised. The '`mkid`' and '`xtokid`' programs also gained a file-tree walker, so that directory names can be passed on the command line instead of the names of every individual file. Greg reorganized and rewrote most of the Texinfo manual to reflect these changes.

Future releases of `id-utils` might include:

an optional coupling with GNU `grep`, so that `grep` can use an ID database for hints
a `cscope` work-alike query interface
incremental update of the ID database.

Index

*

-

A

B

C

D

E

F

G

H

I

L

M

N

O

R

S

T

V

Table of Contents