The ID Database

mkid and friends
lid, gid, aid, eid, pid, iid

Tom Horsley

This document was prepared using version 2.32 of the `texinfo.tex` macro package, but I suspect any version greater or equal to 2.0 will work.

1 Overview

An ID database is simply a file containing a list of file names, a list of identifiers, and a binary relation (stored as a bit matrix) indicating which of the identifiers appear in each file. With this database and some tools to manipulate the data, a host of tasks become simpler and faster. You can `grep` through hundreds of files for a name, skipping the files that don't contain the name. You can search for all the memos containing references to a project. You can edit every file that calls some function, adding a new required argument. Anyone with a large software project to maintain, or a large set of text files to organize can benefit from the ID database and the tools that manipulate it.

There are several programs in the ID family. The `mkid` program scans the files, finds the identifiers and builds the ID database. The `lid` and `aid` tools are used to generate lists of file names containing an identifier (perhaps to recompile every file that references a macro which just changed). The `eid` program will invoke an editor on each of the files containing an identifier and the `gid` program will `grep` for an identifier in the subset of files known to contain it. The `pid` tool is used to query the path names of the files in the database (rather than the contents). Finally, the `iid` tool is an interactive program supporting complex queries to intersect and join sets of file names.

1.1 History

Most of the ID programs were written by Greg McGary and first posted to the net in 1987. Since then several versions have diverged from the original source. The `iid` program was written by Tom Horsley at Harris. This version (the one you are getting with this document) is based on the latest source in use at Harris Computer Systems Division. It contains a few patches from other sources, but none of the major changes made at other sites. This release is the first attempt to start the process of merging the separate versions.

A pre-release version of `mkid` was posted to `alt.sources` near the end of 1990. At that time I mentioned that I was thinking about writing a texinfo manual. I got a lot of encouragement from the net to do so, and this manual is the result. I would like to thank Doug Scofield and Bill Leonard whom I dragooned into helping me poorf raed and edit — they found several problems in the initial version.

2 Mkid

The `mkid` program builds the ID database. To do this it must scan each of the files included in the database. This often takes quite a while, but once the work is done the query programs run very rapidly.

The `mkid` program knows how to scan a variety of of files. For example, it knows how to skip over comments and strings in a C program, only picking out the identifiers used in the code.

Identifiers are not the only thing included in the database. Numbers are also scanned and included in the database indexed by their binary value. Since the same number can be written many different ways (47, 0x2f, 057 in a C program for instance), this feature allows you to find hard coded uses of constants without regard to the radix used to specify them.

All the places in this document where identifiers are written about should really mention identifiers and numbers, but that gets fairly clumsy after a while, so you should always keep in mind that numbers are included in the database as well as identifiers.

2.1 Mkid Command Line Options

`mkid` [`-v`] [`-S`*scanarg*] [`-a`*arg-file*] [`-`] [`-f`*out-file*] [`-s`*directory*] [`-r`*directory*] [`-u`] [`files`...] [Command]

`-v` Verbose. Mkid tells you as it scans each file and indicates which scanner it is using. It also summarizes some statistics about the database at the end.

`-S`*scanarg*
The `-S` option is used to specify arguments to the various language scanners. See Section 2.1.1 [Scanner Arguments], page 4, for details.

`-a`*arg-file*
Name a file containing additional command line arguments (one per line). This may be used to specify lists of file names longer than will fit on a command line.

`-` A simple `-` by itself means read arguments from stdin.

`-f`*out-file*
Specify the name of the database file to create. The default name is `ID` (in the current directory), but you may specify any name. The file names stored in the database will be stored relative to the directory containing the database, so if you move the database after creating it, you may have trouble finding files unless they remain in the same relative position.

`-s`*directory*
Specify a directory to search for SCCS files in case a file named on the command line does not exist.

`-r`*directory*
Specify a directory to search for RCS files in case a file named on the command line does not exist.

`-u` The `-u` option updates an existing database by rescanning any files that have changed since the database was written. Unfortunately you cannot incrementally add new files to a database.

`files` Remaining arguments are names of files to be scanned and included in the database.

2.1.1 Scanner Arguments

Scanner arguments all start with `-S`. Scanner arguments are used to tell `mkid` which language scanner to use for which files, to pass language specific options to the individual scanners, and to get some limited online help about scanner options.

`Mkid` usually determines which language scanner to use on a file by looking at the suffix of the file name. The suffix starts at the last '`.`' in a file name and includes the '`.`' and all remaining characters (for example the suffix of `fred.c` is `.c`). Not all files have a suffix, and not all suffixes are bound to a specific language by mkid. If `mkid` cannot determine what language a file is, it will use the language bound to the `.default` suffix. The plain text scanner is normally bound to `.default`, but the `-S` option can be used to change any language bindings.

There are several different forms for scanner options:

`-S.<`*`suffix`*`>=<`*`language`*`>`
`Mkid` determines which language scanner to use on a file by examining the file name suffix. The '`.`' is part of the suffix and must be specified in this form of the `-S` option. For example '`-S.y=c`' tells `mkid` to use the '`c`' language scanner for all files ending in the '`.y`' suffix.

`-S.<`*`suffix`*`>=?`
`Mkid` has several built in suffixes it already recognizes. Passing a '`?`' will cause it to print the language it will use to scan files with that suffix.

`-S?=<`*`language`*`>`
This form will print which suffixes are scanned with the given language.

`-S?=?` This prints all the suffix$\mapsto$language bindings recognized by `mkid`.

`-S<`*`language`*`>-<`*`arg`*`>`
Each language scanner accepts scanner dependent arguments. This form of the `-S` option is used to pass arbitrary arguments to the language scanners.

`-S<`*`language`*`>?`
Passing a '`?`' instead of a language option will print a brief summary of the options recognized by the specified language scanner.

`-S<`*`new language`*`>/<`*`builtin language`*`>/<`*`filter command`*`>`
This form specifies a new language defined in terms of a builtin language and a shell command that will be used to filter the file prior to passing on to the builtin language scanner.

2.2 Builtin Scanners

If you run `mkid -S?=?` you will find bindings for a number of languages; unfortunately pascal, though mentioned in the list, is not actually supported. The supported languages are documented below[1].

2.2.1 C

The C scanner is probably the most popular. It scans identifiers out of C programs, skipping over comments and strings in the process. The normal `.c` and `.h` suffixes are automatically recognized as C language, as well as the more obscure `.y` (yacc) and `.l` (lex) suffixes.

The `-S` options recognized by the C scanner are:

`-Sc-s<`*`character`*`>`
: Allow the specified <*character*> in identifiers (some dialects of C allow `$` in identifiers, so you could say `-Sc-s$` to accept that dialect).

`-Sc-u`
: Don't strip leading underscores from identifier names (this is the default mode of operation).

`-Sc+u`
: Do strip leading underscores from identifier names (I don't know why you would want to do this in C programs, but the option is available).

2.2.2 Plain Text

The plain text scanner is designed for scanning documents. This is typically the scanner used when adding custom scanners, and several custom scanners are built in to `mkid` and defined in terms of filters and the text scanner. A troff scanner runs `deroff` over the file then feeds the result to the text scanner. A compressed man page scanner runs `pcat` piped into `col -b`, and a TeX scanner runs `detex`.

Options:

`-Stext+a<`*`character`*`>`
: Include the specified character in identifiers. By default, standard C identifiers are recognized.

`-Stext-a<`*`character`*`>`
: Exclude the specified character from identifiers.

`-Stext+s<`*`character`*`>`
: Squeeze the specified character out of identifiers. By default, the characters '`'`', '`-`', and '`.`' are squeezed out of identifiers. This generates transformations like *fred's*$\mapsto$*freds* or *a.s.p.c.a.*$\mapsto$*aspca*.

`-Stext-s<`*`character`*`>`
: Do not squeeze out the specified character.

[1] This is not strictly true — vhil is a supported language, but it is an obsolete and arcane dialect of C and should be ignored

2.2.3 Assembler

Assemblers come in several flavors, so there are several options to control scanning of assembly code:

`-Sasm-c<`*`character`*`>`
: The specified character starts a comment that extends to end of line (in many assemblers this is a semicolon or number sign — there is no default value for this).

`-Sasm+u`
: Strip the leading underscores off identifiers (the default behavior).

`-Sasm-u`
: Do not strip the leading underscores.

`-Sasm+a<`*`character`*`>`
: The specified character is allowed in identifiers.

`-Sasm-a<`*`character`*`>`
: The specified character is allowed in identifiers, but any identifier containing that character is ignored (often a '`.`' or '`@`' will be used to indicate an internal temp label, you may want to ignore these).

`-Sasm+p`
: Recognize C preprocessor directives in assembler source (default).

`-Sasm-p`
: Do not recognize C preprocessor directives in assembler source.

`-Sasm+C`
: Skip over C style comments in assembler source (default).

`-Sasm-C`
: Do not skip over C style comments in assembler source.

2.3 Adding Your Own Scanner

There are two ways to add new scanners to `mkid`. The first is to modify the code in `getscan.c` and add a new `scan-*.c` file with the code for your scanner. This is not too hard, but it requires relinking and installing a new version of `mkid`, which might be inconvenient, and would lead to the proliferation of `mkid` versions.

The second technique uses the `-S<lang>/<lang>/<filter>` form of the `-S` option to specify a new language scanner. In this form the first language is the name of the new language to be defined, the second language is the name of an existing language scanner to be invoked on the output of the filter command specified as the third component of the `-S` option.

The filter is an arbitrary shell command. Somewhere in the filter string, a `%s` should occur. This `%s` is replaced by the name of the source file being scanned, the shell command is invoked, and whatever comes out on *stdout* is scanned using the builtin scanner.

For example, no scanner is provided for texinfo files (like this one). If I wished to index the contents of this file, but avoid indexing the texinfo directives, I would need a filter that stripped out the texinfo directives, but left the remainder of the file intact. I could then use the plain text scanner on the remainder. A quick way to specify this might be:

```
'-S/texinfo/text/sed s,@[a-z]*,,g < %s'
```

This defines a new language scanner (*texinfo*) defined in terms of a `sed` command to strip out texinfo directives (at signs followed by letters). Once the directives are stripped, the remaining text is run through the plain text scanner.

This is just an example, to do a better job I would actually need to delete some lines (such as those beginning with `@end`) as well as deleting the `@` directives embedded in the text.

2.4 Mkid Examples

The simplest example of `mkid` is something like:

```
mkid *.[chy]
```

This will build an ID database indexing all the identifiers and numbers in the `.c`, `.h`, and `.y` files in the current directory. Because those suffixes are already known to `mkid` as C language files, no other special arguments are required.

From a simple example, lets go to a more complex one. Suppose you want to build a database indexing the contents of all the *man* pages. Since `mkid` already knows how to deal with `.z` files, let's assume your system is using the `compress` program to store compressed cattable versions of the *man* pages. The `compress` program creates files with a `.Z` suffix, so `mkid` will have to be told how to scan `.Z` files. The following code shows how to combine the `find` command with the special scanner arguments to `mkid` to generate the required ID database:

```
cd /usr/catman
find . -name '*.Z' -print | mkid '-Sman/text/uncompress -c < %s' -S.Z=man -
```

This example first switches to the `/usr/catman` directory where the compressed *man* pages are stored. The `find` command then finds all the `.Z` files under that directory and prints their names. This list is piped into the `mkid` program. The `-` argument by itself (at the end of the line) tells `mkid` to read arguments (in this case the list of file names) from *stdin*. The first `-S` argument defines a new language (*man*) in terms of the `uncompress` utility and the existing text scanner. The second `-S` argument tells `mkid` to treat all `.Z` files as language *man*. In practice, you might find the `mkid` arguments need to be even more complex, something like:

```
mkid '-Sman/text/uncompress -c < %s | col -b' -S.Z=man -
```

This will take the additional step of getting rid of any underlining and backspacing which might be present in the compressed *man* pages.

3 Database Query Tools

The ID database is useless without database query tools. The remainder of this document describes those tools.

The `lid`, `gid`, `aid`, `eid`, and `pid` programs are all the same program installed with links to different names. The name used to invoke the program determines how it will act.

The `iid` program is an interactive query shell that sits on top of the other query tools.

3.1 Common Options

Since many of the programs are really links to one common program, it is only reasonable to expect that most of the query tools would share common command line options. Not all options make sense for all programs, but they are all described here. The description of each program gives the options that program uses.

`-f<file>` Read the database specified by <*file*>. Normally the tools look for a file named `ID` in either the current directory or in any of the directories above the current directory. This means you can keep a global `ID` database in the root of a large source tree and use the query tools from anywhere within that tree.

`-r<directory>`
The query tools usually assume the file names in the database are relative to the directory holding the database. The `-r` option tells the tools to look for the files relative to <*directory*> regardless of the location of the database.

`-c` This is shorthand for `-r‘pwd‘`. It tells the query tools to assume the file names are stored relative to the current working directory.

`-e` Force the pattern arguments to be treated as regular expressions. Normally the query tools attempt to guess if the patterns are regular expressions or simple identifiers by looking for special characters in the pattern.

`-w` Force the pattern arguments to be treated as simple words even if they contain special regular expression characters.

`-k` Normally the query tools that generate lists of file names attempt to compress the lists using the `csh` brace notation. This option suppresses the file name compression and outputs each name in full. (This is particularly useful if you are a `ksh` user and want to feed the list of names to another command — the `-k` option comes from the `k` in `ksh`).

`-g` It is possible to build the query tools so the `-k` option is the default behavior. If this is the case for your system, the `-g` option turns on the globbing of file names using the `csh` brace notation.

`-n` Normally the query tools that generate lists of file names also list the matching identifier at the head of the list of names. This is irritating if you want just a list of names to feed to another command, so the `-n` option suppresses the identifier and lists only file names.

`-b` This option is only used by the `pid` tool. It restricts `pid` to pattern match only the basename part of a file name. Normally the absolute file name is matched against the pattern.

`-d -o -x -a`
: These options may be used in any combination to limit the radix of numeric matches. The `-d` option will allow matches on decimal numbers, `-o` on octal, and `-x` on hexadecimal numbers. The `-a` option is shorthand for specifying all three. Any combination of these options may be used.

`-m`
: Merge multiple lines of output into a single line. (If your query matches more than one identifier the default action is to generate a separate line of output for each matching identifier).

`-s`
: Search for identifiers that appear only once in the database. This helps to locate identifiers that are defined but never used.

`-u<`*`number`*`>`
: List identifiers that conflict in the first <*number*> characters. This could be useful porting programs to brain-dead computers that refuse to support long identifiers, but your best long term option is to set such computers on fire.

3.2 Patterns

You can attempt to match either simple identifiers or numbers in a query, or you can specify a regular expression pattern which may match many different identifiers in the database. The query programs use either *regex* and *regcmp* or *re_comp* and *re_exec*, depending on which one is available in the library on your system. These might not always support the exact same regular expression syntax, so consult your local *man* pages to find out. Any regular expression routines should support the following syntax:

`.`
: A dot matches any character.

`[ ]`
: Brackets match any of the characters specified within the brackets. You can match any characters *except* the ones in brackets by typing `^` as the first character. A range of characters can be specified using `-`.

`*`
: An asterisk means repeat the previous pattern zero or more times.

`^`
: An `^` at the beginning of a pattern means the pattern must match starting at the first character of the identifier.

`$`
: A `$` at the end of the pattern means the pattern must match ending at the last character in the identifier.

3.3 Lid

`lid` [`-f<`*`file`*`>`] [`-u<`*`n`*`>`] [`-r<`*`dir`*`>`] [`-ewdoxamskgnc`] *patterns...* [Command]

The `lid` program stands for *lookup identifier*. It searches the database for any identifiers matching the patterns and prints the names of the files that match each pattern. The exact format of the output depends on the options.

3.4 Aid

`aid` [`-f<`*`file`*`>`] [`-u<`*`n`*`>`] [`-r<`*`dir`*`>`] [`-doxamskgnc`] *patterns. . .* [Command]

The `aid` command is an abbreviation for *apropos identifier*. The patterns cannot be regular expressions, but it looks for them using a case insensitive match, and any pattern that is a substring of an identifier in the database will match that identifier.

For example '`aid get`' might match the identifiers `fgets`, `GETLINE`, and `getchar`.

3.5 Gid

`gid` [`-f<`*`file`*`>`] [`-u<`*`n`*`>`] [`-r<`*`dir`*`>`] [`-doxasc`] *patterns. . .* [Command]

The `gid` command stands for *grep for identifiers*. It finds identifiers in the database that match the specified patterns, then `greps` for those identifiers in just the set of files containing matches. In a large source tree, this saves a fantastic amount of time.

There is an *emacs* interface to this program (see Section 5.1 [Gnuemacs Interface], page 17). If you are an *emacs* user, you will probably prefer the *emacs* interface over the `eid` tool.

3.6 Eid

`eid` [`-f<`*`file`*`>`] [`-u<`*`n`*`>`] [`-r<`*`dir`*`>`] [`-doxasc`] *patterns. . .* [Command]

The `eid` command allows you to invoke an editor on each file containing a matching pattern. The `EDITOR` environment variable is the name of the program to be invoked. If the specified editor can accept an initial search argument on the command line, you can use the `EIDARG`, `EIDLDEL`, and `EIDRDEL` environment variables to specify the form of that argument.

`EDITOR`	The name of the editor program to invoke.
`EIDARG`	A printf string giving the form of the argument to pass containing the initial search string (the matching identifier). For `vi` it should be set to '`+/%s/'`'.
`EIDLDEL`	A string giving the regular expression pattern that forces a match at the beginning (left end) of a word. This string is inserted in front of the matching identifier when composing the search argument. For `vi`, this should be '`\<`'.
`EIDRDEL`	The matching right end word delimiter. For `vi`, use '`\>`'.

3.7 Pid

`pid` [`-f<`*`file`*`>`] [`-u<`*`n`*`>`] [`-r<`*`dir`*`>`] [`-ebkgnc`] *patterns. . .* [Command]

The `pid` tool is unlike all the other tools. It matches the patterns against the file names in the database rather than the identifiers in the database. Patterns are treated as shell wild card patterns unless the `-e` option is given, in which case full regular expression matching is done.

The wild card pattern is matched against the absolute path name of the file. Most shells treat slashes '`/`' and file names that start with dot '`.`' specially, `pid` does not do this. It simply attempts to match the absolute path name string against the wild card pattern.

The `-b` option restricts the pattern matching to the base name of the file (all the leading directory names are stripped prior to pattern matching).

4 Iid

`iid` [-a] [-c<*command*>] [-H] [Command]

`-a` Normally `iid` uses the `lid` command to search for names. If you give the `-a` option on the command line, then it will use `aid` as the default search engine.

`-c<`*command*`>`
In normal operation, `iid` starts up and prompts you for commands used to build sets of files. The `-c` option is used to pass a single query command to `iid` which it then executes and exits.

`-H` The `-H` option prints a short help message and exits. To get more help use the `help` command from inside `iid`.

The `iid` program is an interactive ID query tool. It operates by running the other query programs (such as `lid` and `aid`) and creating sets of file names returned by these queries. It also provides operators for `anding` and `oring` these sets to create new sets.

The `PAGER` environment variable names the program `iid` uses to display files. If you use `emacs`, you might want to set `PAGER` so it invokes the `emacsclient` program. Check the file `lisp/server.el` in the emacs source tree for documentation on this. It is useful not only with X windows, but also when running `iid` from an emacs shell buffer. There is also a somewhat spiffier version called gnuserv by Andy Norman (`ange%anorman@hplabs.hp.com`) which appeared in `comp.emacs` sometime in 1989.

4.1 Ss and Files commands

The primary query commands are `ss` (for select sets) and `files` (for show file names). These commands both take a query expression as an argument.

`ss` *query* [Subcommand]
The `ss` command runs a query and builds a set (or sets) of file names. The result is printed as a summary of the sets constructed showing how many file names are in each set.

`files` *query* [Subcommand]
The `files` command is like the `ss` command, but rather than printing a summary, it displays the full list of matching file names.

`f` *query* [Subcommand]
The `f` command is merely a shorthand notation for `files`.

Database queries are simple expressions with operators like `and` and `or`. Parentheses can be used to group operations. The complete set of operators is summarized below:

`pattern` Any pattern not recognized as one of the keywords in this table is treated as an identifier to be searched for in the database. It is passed as an argument to the default search program (normally `lid`, but `aid` is used if the `-a` option was given when `iid` was started). The result of this operation is a set of file names, and it is assigned a unique set number.

`lid` — `lid` is a keyword. It is used to invoke `lid` with the list of identifiers following it as arguments. This forces the use of `lid` regardless of the state of the `-a` option (see Section 3.3 [Lid], page 10).

`aid` — The `aid` keyword is like the `lid` keyword, but it forces the use of the `aid` program (see Section 3.4 [Aid], page 11).

`match` — The `match` operator invokes the `pid` program to do pattern matching on file names rather than identifiers. The set generated contains the file names that match the specified patterns (see Section 3.7 [Pid], page 11).

`or` — The `or` operator takes two sets of file names as arguments and generates a new set containing all the files from both sets.

`and` — The `and` operator takes two sets of file names and generates a new set containing only files from both sets.

`not` — The `not` operator inverts a set of file names, producing the set of all files not in the input set.

`set number`
A set number consists of the letter `s` followed immediately by a number. This refers to one of the sets created by a previous query operation. During one `iid` session, each query generates a unique set number, so any previously generated set may be used as part of any new query by referring to the set number.

The `not` operator has the highest precedence with `and` coming in the middle and `or` having the lowest precedence. The operator names are recognized using case insensitive matching, so `AND`, `and`, and `aNd` are all the same as far as `iid` is concerned. If you wish to use a keyword as an operand to one of the query programs, you must enclose it in quotes. Any patterns containing shell special characters must also be properly quoted or escaped, since the query commands are run by invoking them with the shell.

Summary of query expression syntax:

```
A <query> is:
   <set number>
   <identifier>
   lid <identifier list>
   aid <identifier list>
   match <wild card list>
   <query> or <query>
   <query> and <query>
   not <query>
   ( <query> )
```

4.2 Sets

`sets` [Subcommand]

The `sets` command displays all the sets created so far. Each one is described by the query command that generated it.

4.3 Show

`show` *set* [Subcommand]

`p` *set* [Subcommand]

The `show` and `p` commands are equivalent. They both accept a set number as an argument and run the program given in the `PAGER` environment variable with the file names in that set as arguments.

4.4 Begin

`begin` *directory* [Subcommand]

`b` *directory* [Subcommand]

The `begin` command (and its abbreviated version `b`) is used to begin a new `iid` session in a different directory (which presumably contains a different database). It flushes all the sets created so far and switches to the specified directory. It is equivalent to exiting `iid`, changing directories in the shell, and running `iid` again.

4.5 Help

`help` [Subcommand]

`h` [Subcommand]

`?` [Subcommand]

The `help`, `h`, and `?` command are three different ways to ask for help. They all invoke the `PAGER` program to display a short help file.

4.6 Off

`off` [Subcommand]

`quit` [Subcommand]

`q` [Subcommand]

These three command (or just an end of file) all cause `iid` to exit.

4.7 Shell Commands as Queries

When the first word on an `iid` command is not recognized as a builtin `iid` command, `iid` assumes the command is a shell command which will write a list of file names to *stdout*. This list of file names is used to generate a new set of files.

Any set numbers that appear as arguments to this command are expanded into lists of file names prior to running the command.

4.8 Shell Escape

If a command starts with a bang (`!`) character, the remainder of the line is run as a shell command. Any set numbers that appear as arguments to this command are expanded into lists of file names prior to running the command.

5 Other Tools

This chapter describes some support tools that work with the other ID programs.

5.1 Gnuemacs Interface

The source distribution comes with a file named `gid.el`. This is a Gnuemacs interface to the `gid` tool. If you put the file where emacs can find it (somewhere in your `EMACSLOADPATH`) and put `(autoload 'gid "gid" nil t)` in your `.emacs` file, you will be able to invoke the `gid` function using *`M-x gid`*.

This function prompts you with the word the cursor is on. If you want to search for a different pattern, simply delete the line and type the pattern of interest.

It runs `gid` 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 Gnuemacs Manual*).

5.2 Fid

`fid` [`-f<`*`file`*`>`] *file1* [*file2*] [Command]

`-f<`*`file`*`>`	Look in the named database.
`file1`	List the identifiers contained in file1 according to the database.
`file2`	If a second file is given, list only the identifiers both files have in common.

The `fid` program provides an inverse query. Instead of listing files containing some identifier, it lists the identifiers found in a file.

5.3 Idx

`idx` [`-s<`*`directory`*`>`] [`-r<`*`directory`*`>`] [`-S<`*`scanarg`*`>`] *files. . .* [Command]

The `-s`, `-r`, and `-S` arguments to `idx` are identical to the same arguments on `mkid` (see Section 2.1 [Mkid Command Line Options], page 3).

The `idx` command is more of a test frame for scanners than a tool designed to be independently useful. It takes the same scanner arguments as `mkid`, but rather than building a database, it prints the identifiers found to *stdout*, one per line. You can use it to try out a scanner on a sample file to make sure it is extracting the identifiers you believe it should extract.

Command Index

Table of Contents