IDSgrep, version 0.4

Matthew Skala

August 25, 2013

Contents

Quick start

Use `idsgrep` much as you would use `grep`:

`idsgrep` [⟨*options*⟩] ⟨*pattern*⟩ [⟨*file*⟩ ...]

Its general function is to search one or more files for items matching a pattern, like `grep` [8] but with a different pattern syntax. Although potentially usable for an unlimited range of tasks, `idsgrep`'s motivating application is to searching databases of Han script (Chinese, Japanese, etc.) character descriptions. It provides a much more powerful replacement for the "radical search" feature of dictionaries like Kiten [10] and WWWJDIC [6].

The syntax for matching patterns, and the range of command-line options available, are complicated. Later sections of this document explain those things in detail; for now, here are some examples.

`idsgrep 萌 dictionary`

A literal character searches for the decomposition of that character, exact match only.

`idsgrep -d 萌`

The `-d` option with empty argument searches a default collection of dictionaries.

`idsgrep -dtsuku 萌`

The `-d` option can be given an argument to choose a specific default dictionary. Note the argument must be given in the same `argv`-element with the `-d`; the syntax `-d tsuku` with a space would mean "Use the default dictionaries and search for the (syntactically invalid) pattern '`tsuku`.' "

`othersoft | idsgrep 萌`

Standard input will be used if no other input source is specified.

`idsgrep -d ...日`

Three dots match their argument anywhere, so this will match 日, 早, and 萌.

`idsgrep -d '?'`

A question mark, which will probably require shell quoting, matches anything. This is most useful as part of a more complex pattern.

`idsgrep -d '⿱?心'`

Unicode Ideographic Description Characters can be used to build up sequences that also incorporate the wildcards; this example matches characters consisting of something above 心, such as 忽 and 恋 but not 応.

`idsgrep -d '[tb](anything)心'`

There are ASCII aliases for operators that may be inconvenient to type; this query is functionally the same as the previous one.

`idsgrep -d '&!萌...|日月'`

Boolean prefix operators include `&` (AND), `|` (OR), and `!` (NOT). This example matches anything that contains 日 or 月 but is not 萌.

`idsgrep -d '*⿰日?'`

Asterisk makes children match in any order; this example matches 日 at left or right.

`idsgrep -d '@⿱⿱?日?'`

At-sign treats an operator as associative; this example matches both ⿱⿱?日? and ⿱?⿱日?.

`idsgrep -d '.../(femoral)'`

Slash invokes Perl-compatible regular expression matching, which might be useful for the EDICT2-based meaning dictionary.

`idsgrep -d '...=?'`

Equals escapes matching operators; this example searches for a literal question mark anywhere in the tree.

`idsgrep -d '\X840C'`

Several kinds of backslash escapes allow entering characters that might not otherwise be available.

`idsgrep -d -c indent 萌`

The `-c` option selects "cooked" or pretty-printed output modes.

`idsgrep -d -f FontFile.otf '#1'`

The `-f` option reads the character set of an OpenType font and makes it available as a user-defined matching predicate accessed with the hash-mark; in the example, it looks up each character in the default dictionaries.

`idsgrep -U '?'`
: The `-U` option generates a list of Unicode characters.

`idsgrep -Uxdb '?'`
: An optional argument to `-U` specifies information to include in the generated list entries: `x` for hexadecimal value, `d` for decimal, `b` for block name.

`idsgrep -U -f FontFile.otf '#1'`
: Combine `-U` and `-f` to list the characters in a font.

Introduction

The Han character set is open-ended. Although a few thousand characters suffice to write the languages most commonly written in Han script (namely Chinese and Japanese) most of the time, popular standards define tens of thousands of less-popular characters, and there are at least hundreds of thousands of rare characters known to occur in names, historical contexts, and in languages like Korean and Vietnamese that may still use Han script occasionally despite now being written primarily in other scripts.

Computer text processing systems that use fixed lists of characters will inevitably find themselves unable to represent some text. As a result, there is a need to *describe* characters in a standard way that may have no standard code points of their own. A similar need for descriptions of characters arises when looking up characters in a dictionary; a user may recognize some or all the visual features of a character (such as its parts and the way they are laid out) without knowing how to enter the character as a whole.

IDSgrep's main function is to query character description databases in a flexible way. This need was identified during development of the Tsukurimashou font family [15]; there, the visual appearance of Han character glyphs corresponds directly to the MetaPost code implementing them, and the desire for code re-use and consistency often motivates a close examination of the existing work to answer questions like "What other characters contain this shape, and how did we implement it last time?" Standard tools like `grep` [8] can sometimes be applied to answer such questions by searching for subroutine names in the source code, but the related question of "What other characters, not yet implemented, could we build that would use this shape?" requires comparing with some external database of the characters commonly used in the language. How can we run `grep` on the writing system itself?

Someone confronted with an unknown character and wanting to look it up in a more ordinary dictionary to find the meaning may, similarly, want to search for characters based on specific features while leaving others unspecified, with questions like "What characters exist that have the common 心 shape at the bottom, with the upper part divided into two things side by side? The two things at the top are shapes I don't recognize, printed too small for me to identify them more precisely." Existing dictionary-query methods are not adequate for some reasonable queries of this nature.

For instance, the radical-and-stroke-count method of traditional character dictionaries requires identifying the head radical and counting strokes, both of which may be difficult; dictionary codes like SKIP and Four Corners key on some layout attributes but not all; multi-radical search allows the user to choose whichever radicals they recognize, but it ignores layout entirely; and computer handwriting recognition generally works well if and only if the user is sure of the writing of the first few strokes in the character. Furthermore, these search schemes often are implemented only in heavy, non-portable, GUI software that cannot be automated and mixes poorly with standard computing environments. IDSgrep can answer the example query correctly with a single, simple command line (`idsgrep -d '[tb][lr]??心'`). This software is intended to bring the user-friendliness of `grep` to Han character dictionaries.

☡ Some passages in this manual are marked with the Knuthian "dangerous bend" sign in the margin. These cover obscure or difficult aspects of the software, less likely to be of interest to first-time users. Beginners are encouraged to skip these parts on the first reading, then come back and check them out later if interested.

What's new

The main new features in version 0.4 are:

- changes to the build system for better integration with Tsukurimashou;
- experimental support for bit vector indices;
- support for user-defined matching predicates, and in particular, the ability to match against the list of characters defined by a font file ("`-f`" option);

- built-in generation of Unicode character lists ("`-U`" option).

Download, build, test, and install

IDSgrep is distributed under the umbrella of the Tsukurimashou project on Sourceforge.JP [15], `http://tsukurimashou.sourceforge.jp/`. Releases of IDSgrep will appear on the project download page; development versions are available by SVN checkout from the `trunk/idsgrep` subdirectory of the repository. For the convenience of Github users, the Tsukurimashou (and thus IDSgrep) repository is also mirrored into a Github repository [16], but the project on Sourceforge.JP and its SVN repository remain the main public locations for IDSgrep development and all bug-tracker items should be filed there.

A minimal default build and install could run something like this:

```
tar -xzvf idsgrep-0.3.tar.gz
cd idsgrep-0.3
./configure
make
su -c 'make install'
```

IDSgrep can build dictionaries from the Tsukurimashou font package, which is available through the same Sourceforge.JP project as IDSgrep; from the KanjiVG database available at `http://kanjivg.tagaini.net/` [3]; from the CHISE IDS database available at `http://chise.zinbun.kyoto-u.ac.jp/dist/ids/` [1]; or from the EDICT2 database available at `http://www.csse.monash.edu.au/~jwb/edict.html` [5]. For an ideal complete installation of IDSgrep, one would download all those packages, build Tsukurimashou first, and make it and the dictionaries available to the IDSgrep `configure` script. A precompiled version of the CHISE IDS-derived dictionary is bundled in the IDSgrep distribution tarball, so that one should be available (though not necessarily up-to-date) without any dependencies.

Regular expression matching requires building with the Perl-Compatible Regular Expression (PCRE) library available at `http://www.pcre.org/` [9]. Many Linux distributions install this library by default. Without PCRE, the regular-expression matching features will not be included, and any attempt to do regular-expression matching will result in a fatal error.

Getting maximum benefit from the bit vector indexing features requires building with the BuDDy binary decision diagram library available at `http://sourceforge.net/projects/buddy/` [11]. Many Linux distributions install this library by default. Without BuDDy, bit vectors will still work, but will not speed up IDSgrep by as great a factor.

The `configure` script will by default make a reasonable effort to find the dependencies; in many common cases it is not necessary to specify them on the command line. Here is a more complete installation process relying on `configure` to find Tsukurimashou in a sibling directory and the other dictionaries in the current directory:

```
unzip tsukurimashou-0.6.zip
cd tsukurimashou-0.6
./configure
make
# install of Tsukurimashou not needed by IDSgrep
cd ..
tar -xzvf idsgrep-0.2.tar.gz
cd idsgrep-0.2
ln -s /some/where/else/kanjivg-20120219.xml.gz .
ln -s /some/where/else/edict2.gz .
ln -s /some/where/else/chise-ids-0.25 .
./configure
make
make check
su -c 'make install'
```

It is necessary to at least configure Tsukurimashou, if not fully build it, before building IDSgrep. The IDSgrep build will then invoke the Tsukurimashou build to create just the files needed by IDSgrep. It is not necessary to configure or build CHISE IDS (which would require first installing other parts of the larger CHISE system and probably XEmacs as well); IDSgrep only needs to look at the CHISE IDS data files.

If the default search fails, the filenames of KanjiVG (`.xml` or `.xml.gz`), EDICT2 (`.gz`), and the directories containing extracted distributions of Tsukurimashou and CHISE IDS can be specified on the `configure` command line with the `--with-kanjivg`, `--with-edict2`, `--with-tsuku-build`, and `--with-chise-ids` options. For other options, run `configure --help`. It's a reasonably standard GNU Autotools [7] configuration script, albeit with a lot of options for inapplicable installation directories removed to simplify the help message.

The EDICT2-based dictionary should preferably include character decompositions from some other dictionary; which one is selectable by the `--enable-edict-decomp` option. Allowed values include `chise`, `kanjivg`, `tsuku`, and `no`; the default of `auto` will try all of those in that order and use the first that works. The value `no` corresponds to simply mapping every character to itself without further decomposition; that is obviously not as informative as might be desired, but it will still allow for regular expression searches.

The "`check`" Makefile target runs the IDSgrep test suite. Some tests require the dictionary files and will be skipped if those are not present. There is also a test that will use Valgrind [14] if available, to check for memory-related problems; if Valgrind is not found in the `PATH`, this test will be skipped.

The IDSgrep installation process will attempt to build and install bit vector indices for whatever dictionaries it installs. Doing that entails running the `idsgrep` binary for the target system on the build system, and so it is unlikely to work in the case of Autotools-mediated cross-compilation. Cross-compilation has not been tested at all and would likely fail anyway, but this point seemed worth mentioning for the benefit of anyone who might be trying to push the limits. Similarly, the bit vector index files are architecture-specific. An `idsgrep` binary that encounters bit vector files for a foreign architecture will ignore them and use the unfiltered matching algorithm; unfiltered matching is much slower, though otherwise harmless. If you share parts of your installation between multiple architectures (for instance, on a heterogenous LAN), and you wish `idsgrep` binaries on every architecture to benefit from bit vectors, or if you are just finicky in a general way about keeping architecture-specific files and "pure" data separate from each other, then you may wish to pay close attention to where your dictionaries and indices are stored, and place bit vectors and dictionaries in architecture-specific space.

The `configure` script supports an `--enable-gcov` switch to enable meta-testing of the test suite's coverage. This feature requires that the Gcov coverage analyser be installed. To do a coverage analysis, run `configure` with `--enable-gcov` and any other desired options, then do `make clean` (necessary to be sure all object files are rebuilt with the coverage instrumentation) followed by `make check`. Most people would not want to install the IDSgrep binary itself when built under this option. As of version 0.4, the current test suite is not expected to achieve full coverage on most installations (though it should come close), so do not report failure of this test as a bug nor get too concerned about it.

Unicode IDSes

Although IDSgrep uses a more elaborate syntax, it is well to know about the Unicode Consortium's "Ideographic Description Sequences" (IDSes), which are a subset of IDSgrep's. These are documented more fully in the Unicode standard [19].

- A character from one of the Unified Han or CJK Radical ranges is a complete IDS and simply represents itself. For instance, "大" is a complete IDS.
- The Ideographic Description Characer (IDC) code points U+2FF0, U+2FF1, and U+2FF4 through U+2FFB, whose graphic images look like ⿰⿱⿴⿵⿶⿷⿸⿹⿺⿻, are prefix binary operators. One of these characters followed by two complete IDSes forms another complete IDS, representing a character formed by joining the two smaller characters in a way suggested by the name and graphical image of the IDC. For instance, "⿰日月" describes the character 明. These structures may be nested; for instance, "⿰言⿱五口" describes the character 語.
- The IDC code points U+2FF2 and U+2FF3, which look like ⿲⿳, are prefix ternary operators. (Unicode uses the less-standard word "trinary" to describe them.) One of them can be followed by three complete IDSes to form an IDS that describes a character made of three parts, much in the same manner as the binary operators. For instance, "⿱⿲糸言糸夊" describes the character 變.
- As of Unicode 6.1, IDS length is unlimited. Earlier versions specified that an IDS could not be more than 16 code points long overall nor contain more than six consecutive non-operator characters. This rule appears to be have been intended to make things easier for systems that need to be able to jump into the middle of text and quickly find the starts and ends of IDSes.
- IDSes non-bindingly "should" be as short as possible and should reflect "the natural radical-phonetic division for an ideograph if it has one."

The Unicode standard does not mention variation selectors in any IDS-related context, except that it offers the possibility of prefixing U+303E, the "ideographic variation mark," to the entire sequence to indicate a variation. Such a prefix is explicitly defined not to be counted as part of the IDS.

My opinion is that Unicode did not intend to permit variation selectors inside IDS syntax. Variation selectors arguably exist to patch over encoding inadequacies resulting from Unicode's internal politics. When a code point is not really specific enough, because it refers to two or more things which you think are not actually the same thing, then you can add a variation selector to indicate which thing you really mean. IDSes, on the other hand, bearing in mind that they are imported from GBK, operate at a lower level to specify characters in terms of parts that are assumed to be adequately encoded. If a code point to be used in an IDS is not specific enough, then that element should be described with a smaller fragment of IDS syntax instead of by using the ambiguous code point. If the closest match possible is still not perfect, then it is time to use U+303E. The fact they offer the U+303E mechanism for specifying variations offers further support to the idea that they did not intend to allow variation selectors *inside* IDSes.

However, it's a difficult question because IDSes, by addressing the visual appearance of characters instead of their semantics, fundamentally challenge the basic Unicode principle that code points specify characters and not glyphs. The distinction between characters and glyphs simply cannot be made perfectly in all cases. For use in cases where variation selectors appear to be appropriate, both CHISE IDS and IDSgrep extend IDS syntax in such a way as to allow them in some way.

Interface to CHISE IDS

The CHISE project [1] maintains a database of Han characters covering multiple languages as part of a larger processing environment that also includes a version of XEmacs [20] modified to follow the principles of the UTF-2000 initiative [12]. It also has connections to GlyphWiki [2]. These systems are documented primarily in Japanese; English-language documentation is sparse and not necessarily up to date.

For IDSgrep's purposes, the most interesting part of CHISE is a module called CHISE IDS, which includes a database of about 140000 characters (exact count depending on the version), with decompositions in its own extension of Unicode IDS syntax. The main purpose of this IDS database is to provide a search capability within the modified XEmacs; there is also code for a Web search form. From examination of the database files it appears that the rules for CHISE IDS's extended IDS syntax are more or less as follows.

- Generally, Unicode IDS rules apply.
- An XML entity-like sequence of the form `&NAME;` counts as a single ideograph. The field indicated by `NAME` is a symbolic identifier or database key defined internally by the project. Such identifiers have been observed to contain uppercase ASCII letters, numerals, hyphens, and plus signs; they usually consist of a short alphabetic prefix, a hyphen, and a number. These entity references are usually used to refer to characters for which CHISE has an encoding and Unicode doesn't.
- A Unicode variation sequence (an ideograph followed by a variation selector) counts as a single character.

Although CHISE IDS's extensions to IDS permit strings that would not be valid IDSgrep EIDS syntax, it is easy to convert them into EIDS format. IDSgrep includes a `chise2eids` Perl script for that purpose. The `configure` script will look for CHISE IDS in a directory named `chise-ids-*` in a short list of likely places, or use the value of the `--enable-chise-ids` command-line option if one is given. This directory should simply be an unpacked CHISE IDS distribution tarball, or a checkout from the CHISE IDS Git repository. It is not necessary to run CHISE's Makefile, which would require also having and installing other parts of the larger system.

As of this writing (March 6, 2013), the CHISE IDS distribution tarballs are available from `http://chise.zinbun.kyoto-u.ac.jp/dist/ids/`, and there is a Git repository at `http://git.chise.org/git/chise/ids.git`. However, on March 5, there was an announcement posted to the CHISE Project's mailing lists in English and Japanese saying that on March 11 all the servers at chise.org (which includes the mailing lists themselves) will cease to operate. It is not clear to me whether that is intended to be something temporary, or the permanent end of the project, but given that nothing seems to have happened in CHISE in years, I fear the worst. I have posted a snapshot of the current CHISE IDS Git repository in my own Github account at `https://github.com/mskala/chise-ids`; that should be a long-term stable source of the data used by IDSgrep.

The last distribution tarball of CHISE-IDS was version 0.25 dated June 2010. The Git versions are more recent and may be preferable. The directory created by checking out a Git version will probably not have a name recognized automatically by the build system, so it should be given on the `configure` command line with `--enable-chise-ids`.

Roughly 6% of the entries in the CHISE IDS database include invalid extended IDS syntax—most often in the form of too many children for the operators used, or less often, too few. Most but not all of the errors occur in the `IDS-HZK??.txt` files, which are unmaintained. It appears that the native search tools for the database generally work on the basis of pure substring searches, where the higher-level syntax errors that would be detected by the IDSgrep parser can go unnoticed. The `chise2eids` program generates a `chise.errs` file during build, listing all the syntax errors it finds (11748 of them in the current Git version as of this writing); invalid entries are otherwise ignored and will not appear in the main output file `chise.eids`. Although 6% may sound like a lot of errors, the invalid entries are generally in sufficiently obscure character components that it should have little practical effect on the quality of dictionary lookups: at worst, some character components may end up not broken down into pieces as small as would otherwise be possible.

CHISE IDS refers to individual characters in a more general way than just by single Unicode code point: sometimes it uses a variation sequence consisting of a kanji code point followed by a variation selector in the U+FE00 to U+FE0F or U+E0100 to U+E01EF ranges, and sometimes it uses a string that looks like an XML character entity reference, along the lines of "`&NAME;`." Both of these map naturally to IDSgrep's concept of a multi-character head. The two-code-point sequence U+840C U+E0101 is translated to the IDSgrep syntax "`<\X840c\X{E0101}>;`," and the XML-like syntax "`&FU-123;`" is translated to "`<FU-123>;`." CHISE IDS does not seem to refer to the same character in different ways (for instance, a code point with no variation selector somehow matching as a default to the same code point with a variation selector, which might be plausible under Unicode's definition of what variation selectors signify) and `chise2eids` does not attempt to accomodate anything like that.

The CHISE IDS database is covered by the GNU GPL version 2 or later, which is basically compatible with the GNU GPL version 3 used by IDSgrep. So distributions of IDSgrep can reasonably include a bundled copy of `chise.eids` for the benefit of users who don't want to download the separate package and generate their own. Without taking a position on whether the `chise.eids` file constitutes "object code" for the purposes of the GNU GPL as opposed to being modified source code in itself, I am willing to provide a copy of the CHISE IDS checkout I used to generate my version of `chise.eids`, to anyone who contacts me about it at `mskala@ansuz.sooke.bc.ca`. Going directly to the original distribution points at the URLs given above is probably a more convenient option for most users.

Interface to KanjiVG

The KanjiVG project [3] maintains a database of kanji (Han characters as used by Japanese) in an extended SVG format, which implies that it is XML. The `kvg2eids` Perl script, included as part of IDSgrep, is capable of reading this database and converting it to Extended Ideographic Description Sequences (EIDSes). As described above, if a reasonably recent version of KanjiVG's compressed XML file is available to `configure`, then IDSgrep's build will create such a dictionary and `make install` will install it.

KanjiVG describes characters primarily in terms of strokes, not multi-stroke components, and it attempts to follow the official stroke order and etymological component breakdown. That approach results in some peculiarities from the point of view of dictionary searching. For instance, in the kanji 園, the official stroke order is to write two strokes of the enclosing box, then the central glyph, then the bottom of the box. KanjiVG's XML file lists two "elements" identified with the kanji 囗, one for the first two strokes and one for the final stroke, with additional attributes specifying that they are actually two parts of the same element. KanjiVG has changed its own standard for how to represent this information in the recent past, and not all entries have been updated to the latest standard yet. The current version of `kvg2eids` does not correctly process 園 nor some other characters with parts written in nonsequential order. On that particular one it generates a special functor containing debugging information; for some others, it may actually generate an EIDS with the same radical appearing multiple times, following the structure described in KanjiVG whether it's what was intended or not. As a result, not all entries in the dictionary will be right. However, only a few are affected by this issue.

As of March 2012, I (Matthew Skala, the author of IDSgrep) have become a member of the KanjiVG project and there is some possibility that KanjiVG's database design will change in a way that makes it easier to recover spatial organization for searching with IDSgrep.

With the current versions of IDSgrep and KanjiVG, the KanjiVG-derived dictionary contains 6660 entries covering all the popularly-used Japanese kanji. Note that the KanjiVG input file, and presumably the resulting format-converted dictionary, are covered by a Creative Commons Attribution–ShareAlike license, distinct from the GNU GPL applicable to IDSgrep itself.

Interface to EDICT2

Jim Breen's JMdict/EDICT project maintains a file called EDICT2 [5] which is more like a traditional dictionary, with words and meanings, than a database of kanji. Such dictionaries are not the primary target of IDSgrep and IDSgrep's query syntax is not perfectly suited to them. However, the regular-expression matching features may make it practical to search EDICT2 with IDSgrep, and there is some value in being able to do sub-character structural searches on multi-character words.

If another dictionary besides EDICT2 is available (subject to configuration by `--enable-edict-decomp`), then the build system will generate and install a dictionary file called `edict.eids` which represents a database join of EDICT2 with the other dictionary. With no other dictionary, the file can still be generated but will contain no character decomposition information. A sample entry might look like this:

```
【明】,<明>⿰日月《[みん] (n) Ming (dynasty of China)》
```

The head for the entire entry is the head from the EDICT2 entry. Then the tree is a binary tree with a comma as the functor and the first child being the entire decomposition dictionary entry for the first character. The second child represents the rest of the entry. With a two-character or longer head, this child would also be a binary comma with the second character of the entry head as its first child. In this way the characters of the entry head are all represented as left children of commas, forming a linked-list structure (much like a Prolog linked-list with commas instead of dots as the functors). The final child at the bottom is a nullary node containing as its functor simply the rest of the EDICT2 entry.

The rationale for this syntax is that it allows a relatively simple way of querying multi-character words in EDICT2 using the existing IDSgrep query types. To find an exact match, just query the head (which will require head brackets and a semicolon if the query is more than one character long), as in `idsgrep -de '<教育>;'`. To search for the first few characters, commas can be imagined as separators (though their actual function is quite different) with a comma at the start and a question mark at the end, as in `idsgrep -de ',教,育?'`. These queries can be combined with the sub-character breakdown queries already supported by the decomposition dictionaries. For instance, `idsgrep -de ',教,...|日月!,??'` will search for, and give definitions of, words of exactly two characters in which the first is 教 and the second character contains 日 or 月 anywhere. The restriction to exactly two characters is accomplished by the sub-query "`!,??`", which fails to match on the binary comma that would be present at that point in a longer word.

EDICT2 is under the Creative Commons Attribution–ShareAlike license. Since KanjiVG is as well, that license would presumably also apply to a combined dictionary made from EDICT2 and KanjiVG. An EDICT2-only dictionary with no decompositions from other sources should similarly be under Creative Commons Attribution–ShareAlike. It might not be legal to distribute outside one's own organization a dictionary formed by joining EDICT2 with CHISE IDS or Tsukurimashou, because those sources are covered by versions of the GNU GPL, which is not compatible with the Creative Commons license.

Interface to Tsukurimashou

IDSgrep is closely connected with the Tsukuimashou font family [15]. They have the same author; it was largely for use in Tsukurimashou development that IDSgrep was developed at all; and IDSgrep's source control system is a subdirectory within Tsukurimashou's. Building IDSgrep in conjunction with Tsukurimashou allows IDSgrep to extract from the Tsukurimashou build system a dictionary of character decompositions as they appear in Tsukurimashou. The Tsukurimashou fonts are also necessary to build this IDSgrep user manual. However, IDSgrep is also distributed as a separate package, because it will be of use to non-users of Tsukurimashou, and the Tsukurimashou build system will not recurse into IDSgrep's directory and build IDSgrep by default; only if requested.

IDSgrep is one of several parasite packages of Tsukurimashou, using a mechanism introduced in Tsukurimashou 0.7 and IDSgrep 0.4. Previous versions used a different interface.

To build Tsukurimashou with IDSgrep: specify the "`--enable-parasites`" option to Tsukurimashou's `configure` script with an appropriate value, such as "`--enable-parasites=idsgrep`". See the Tsukurimashou documentation for other possible values of this option. Building Tsukurimashou will then implicitly build IDSgrep. It should be possible to pass IDSgrep `configure` options to Tsukurimashou's `configure` script and have them automatically passed down the chain (in the standard Autotools sub-package fashion) but that is not well-tested.

To build Tsukurimashou without IDSgrep: this is the default when you run the Tsukurimashou build from the root of the Tsukurimashou distribution. The IDSgrep source is included as a subdirectory in distributions of Tsukurimashou, but only built on request.

For a more customized build of IDSgrep, with or without Tsukurimashou: you can also run IDSgrep's `configure` in its own directory, and then do `make` (and the usual targets) there. It will look for Tsukurimashou (specifically, a build directory in which Tsukurimashou's `configure` has *already been executed*) as the parent directory and in a few other places, or you can specify the location of a Tsukurimashou build with the "`--with-tsuku-build`" option to IDSgrep's `configure`. If Tsukurimashou is not available, IDSgrep will build without creating the Tsukurimashou-derived dictionary file.

During IDSgrep's build, if it can access a Tsukurimashou build directory, it will recursively call `make eids` on Tsukurimashou's build system. That is a hook that causes Tsukurimashou's build system to generate the EIDS decomposition dictionary, which is then copied or linked back into IDSgrep's build directory and can be installed with IDSgrep's `make install`. IDSgrep's build will also look in Tsukurimashou's build directory for the font "Tsukurimashou Mincho" which is needed to build this user manual, and will make recursive calls to `make` for Tsukurimashou to build that if necessary. This kind of upward-callback `make` invocation is a little inefficient (in particular, it does not handle jobserver mode well) so it is better, if you want both packages, to use the centralized Tsukurimashou build system, which will do its own thing first and then call IDSgrep's build near the end in a better-integrated way. If you want to run "`make install`" just on IDSgrep and not on Tsukurimashou (which might be a reasonable thing to want because of operating system font installation issues), you should run just "`make`" in Tsukurimashou's directory, then `cd` to IDSgrep's directory and run "`make install`."

A note on TrueType/OpenType

This version of IDSgrep is designed to read TrueType or OpenType files (the distinction between the two is not relevant at this level) for character map information. The specification for the TrueType/OpenType file format reads like a parody. I'd like to take a moment to complain about a few things.

- Although the format contains binary fields which must be read in a specific byte order, one of the two magic numbers that can identify the file format (for a single font as opposed to a "collection") is 0x4F54544F, which is a palindrome at the byte level and thus useless for detecting byte order problems.
- The other possible magic number for non-collection files is 0x00010000, which is quite likely to occur in files that are not TrueType/OpenType files, making it harder to detect when one may have been passed a bad file.
- Many decades of research on error detection codes were ignored in the design of the OpenType checksum algorithm, which (among other issues) cannot detect any reordering of 32-bit words unless it crosses a table boundary. At least the algorithm produces its meaningless results fast; yay, efficiency!
- There are 32-bit byte offsets referenced to the start of the file. There are 32- and 16-bit byte offsets referenced to the start of the current table. There are 32- and 16-bit byte offsets referenced to the *locations of the offset fields themselves*, so a field at offset 0x1234 referring to another field at offset 0x5678 will contain 0x4444. There are also indices measured in units other than bytes.
- There are variable-length objects not tagged with their lengths except indirectly: they are presumably contained entirely within larger objects that are tagged with lengths.
- Consider the cmap format 4 subtable, which Microsoft says is their preferred format. It includes four variable-length arrays each containing segCount number of two-byte entries. The value of segCount is not directly recorded anywhere, but these values are all required in the header:
    - ◦ $2 \cdot \mathrm{segCount}$;
    - ◦ $2 \cdot 2^{\lfloor \log_2 \mathrm{segCount} \rfloor}$;

  - $\log_2(2 \cdot 2^{\lfloor \log_2 \mathrm{segCount} \rfloor}/2)$ (which is described like that in the spec); and of course
  - $2 \cdot \mathrm{segCount} - 2 \cdot 2^{\lfloor \log_2 \mathrm{segCount} \rfloor}$.

- The bizarre length-derived values in the format 4 header (and other similar sets of table-size-logarithm numbers that occur elsewhere in the file format) appear to be designed to support someone's binary search code. Instead of computing those numbers itself starting from the length, the search code can just use values straight from the table to initialize its variables. Consider what would happen if someone actually did that as the designers apparently intended, and the numbers happened to be incorrect in the file. If, for instance, numbers in the file were swapped around on 32-bit boundaries, the checksums wouldn't detect a problem; and the speed demons who think they need precomputed logarithms probably aren't wasting time checking checksums anyway. The code isn't checking whether the numbers are consistent (because to do that you would have to calculate them fresh, and then why bother storing them in the first place?), so it will end up "searching" into random areas of the file, or into uninitialized memory beyond. Now think about the relative costs of disk reads, network transfer, and arithmetic, and consider whether having those values precalculated and stored in the file would actually save any time even if they could be trusted.

- The cmap format 4 subtable consists of, in this order: fixed-length stuff totalling 14 bytes; one variable-length array of length $2 \cdot \mathrm{segCount}$ bytes; *one more two-byte fixed-length field*; three more variable-length arrays each of length $2 \cdot \mathrm{segCount}$ bytes; and finally, one more variable-length array whose length is not directly specified anywhere but could presumably be inferred by subtracting from the known size of the overall table. The four $2 \cdot \mathrm{segCount}$-byte arrays are actually the rearranged slices of a single logical array whose elements are four-field structures; but the extra reserved two bytes stuck in the middle of the table make a straightforward transposition impossible. Four-tuples of the same kind with the same four fields also occur in the format 2 subtable; but there, they occur as a single array with each record written in an 8-byte block.

- It is an intended, documented feature that some of the variable-length arrays in TrueType/OpenType may overlap with each other. As a result, bounds-checking, in addition to being intrinsically difficult because of the lack of information, would cause the reader to reject some files that the specification claims are legitimate.

- Code-injection bugs allowing execution of arbitrary code in a privileged context have been reported in software that implemented this file format without bounds-checking. This should surprise no one.

IDSgrep attempts to do all reasonable bounds-checking on the fields it needs, and to ignore fields it does not need; given a bad TrueType/OpenType file, it is intended that IDSgrep should be able to make the best of it and at worst fail gracefully with an error message. It should not be possible to crash IDSgrep by giving it a bad font file to read.

However, the nature of the file format means that at least in the current version, we can't be confident all possible problems have been foreseen and excluded. Let me know if you find a font file that makes IDSgrep crash and I'll try to fix it. IDSgrep probably should not be allowed to read font files supplied by untrusted sources such as Web users.

Invoking `idsgrep`

The command-line `idsgrep` utility works much like most other command-line programs, and like `grep` [8] in particular. It takes options and other arguments. The first non-option argument is an EIDS representing the matching pattern, and any remaining non-option arguments are taken as filenames to read. If there are no filenames, `idsgrep` will read from standard input. Output always goes to standard output.

When there is more than one file being read (either by direct specification or indirectly with the `-d` dictionary option), `idsgrep` will preface each EIDS in its output with "`:`⟨*filename*⟩`:`" to indicate in which file the EIDS was found. Note that under the EIDS syntax rules, that creates a unary node senior to the entire tree, so that the output remains in valid EIDS format, except in the case of filenames containing colons, which will be handled via backslash escapes in the future when those are fully implemented for output.

Command-line options

`-c`, `--cooking` Select the output generation and input canonicalization mode. Requires one argument, which may be one of the keywords `raw`, `rawnc`, `ascii`, `cooked`, or `indent`, to specify a preset mode; or a string of up to twelve decimal digits to control the output in more detail. The default mode is `raw`. See the section on "cooked output" in this manual for more details.

`-d`, `--dictionary` Read a dictionary from the standard location. There is a pathname for dictionaries hardcoded into the `idsgrep` binary, generally {*prefix*}`/share/dict`, and if this option is given, its argument (which may be empty) will be appended to the dictionary directory path, followed by "`*.eids`," and then treated as a shell glob pattern. Any matching files are then searched in addition to those otherwise specified on the command line. A small added wrinkle is that when more than one file is searched (resulting in :*filename*: tags on the output lines), any of them that came from the `-d` option will be abbreviated by omitting the hardcoded path name. The purpose of this option is to cover the common case of searching the installed dictionaries. Just specifying "`-d`" will search all the installed dictionaries; specifying an abbreviation of the dictionary name, as "`-dt`" or "`-dk`," will search just the matching one; and it remains possible to specify a file exactly or use standard input in the usual `grep`-like way.

`-f`, `--font-chars` Read a font file and make its character coverage available as a user-defined matching predicate through the "`#`" matching operator. In the current version, this feature can only read TrueType and OpenType files that contain Unicode (or near equivalent) mappings described with cmap subtable types 0, 2, 4, 12, or 13. This option may be specified multiple times, with successive invocations corresponding to user-defined predicates 1, 2, 3, and so on. The maximum number of user-defined predicates is limited to the number of bits in the largest integer type available to the C compiler; 32 or 64 on many systems.

`-h`, `--help` Display a short summary of these options.

`-G`, `--generate-index` Instead of searching for trees that match a matching pattern, generate and write to standard output a bit vector index of the specified file or files. This index, if written to a filename with a `.bvec` extension and placed alongside the input that generated it in a correspondingly-named `.eids` file, may speed up future searches. This feature is normally invoked automatically during program installation; users will only need to use it directly if they are building their own dictionaries. No matching pattern will be taken from the command line; all non-option arguments will be read as filenames. The `-U` option will be ignored. If there is more than one input file, the results for them will be concatenated, which is unlikely to be useful. See the chapter on bit vector indices in this manual for more information.

`-I`, `--ignore-indices` Do *not* look for or use any bit vector indices.

`-U`, `--unicode-list` Generate a dictionary of Unicode code points, and read that before reading any other dictionaries or input files that may be specified. The generated dictionary consists of a single line for each of the code points U+0000 through U+10FFFF in ascending order, excluding the surrogates but not any other invalid or non-character code points; on each line, there is a tree whose head is the character and whose body is either a nullary semicolon or (if the optional argument to `-U` was specified) a nullary functor containing semicolon-separated pieces of information selected by the characters of the optional argument. Characters permitted in the argument are "`b`" for the Unicode block name; "`d`" for the decimal value of the code point; and "`x`" for the hexadecimal value with "U+." For example, specifying "`-Uxdb`" will generate and scan a dictionary that includes the line "`<A>(U+0041;65;Basic Latin)`." This option is intended to be used together with `-f` to produce font coverage lists.

Bit vector indexing is of no use for the internally-generated Unicode list, but when the query tree has a head, IDSgrep will generate only the at most one dictionary entry that could match that query, giving something very much like the benefit of bit vector indexing. This option generates the entries as EIDS trees in an internal format, not as a byte stream, bypassing the input parser, so output from `-U` is always cooked even when a raw mode is selected with `-c` to be used for real input.

`-V`, `--version` Display the version and license information for IDSgrep.

`--bitvec-debug` Report detailed bit vector debugging and performance information to standard error. The information reported is terse, undocumented, and probably not of interest to most users. Whereas in ordinary operation, `idsgrep` will silently switch to plain tree-matching should the bit vector index be unavailable or invalid, with this option the absence of a valid bit vector index for an input file will be treated as a fatal error.

`--statistics` Report a line of machine-readable performance information to standard output at the end of the run. This may be useful in optimizing the bit vector features. See the chapter on bit vectors in this manual for more information about the format and significance of the statistics.

Environment variables

The `idsgrep` utility recognizes just one environment variable, `IDSGREP_DICTDIR`, which if present specifies a directory for the `-d` option to search instead of its hardcoded default.

Note that `idsgrep` does not pay attention to any other environment variables, and in particular, not `LC_ALL` and company. The input and output of this program are always UTF-8 encoded Unicode *regardless of locale settings.* Since the basic function of this program is closely tied to the Unicode-specific "ideographic description characters," it would be difficult if not impossible for it to work in any non-Unicode locale. Predictability is also important because of the likely usefulness of this software in automated contexts; if it followed locale environment variables, many users would have to carefully override those all the time to be sure of portability. Instead of creating that situation, `idsgrep` by design has a consistent input and output format on all systems and users are welcome to pipe things through a conversion program if necessary.

Technical details

This section is intended to describe IDSgrep's syntax and matching procedure in complete precise detail; and those things are, in turn, designed to be powerful rather than easy. As a result, the description may be confusing for some users. See the examples in the "Quick start" section for a more accessible introduction to how to use the utility.

The system is best understood in terms of three interconnected major concepts:

- an abstract data structure;
- a syntax for expressing instances of the data structure as "Extended Ideographic Description Sequences" (EIDSes);
- a function for determining whether two instances of the data structure "match."

Then the basic function of `idsgrep` is to take one EIDS as a matching pattern, scan a file containing many more, and write out the ones that match the matching pattern. The three major concepts are described, one each, in the following sections. A final section describes options for how the command-line `idsgrep` program generates EIDS syntax on output.

The data structure

An *EIDS tree* consists of the following:

- An optional *head*, which if present consists of a nonempty string of Unicode characters.
- A required *functor*, which is a nonempty string of Unicode characters.
- A required *arity*, which is an integer from 0 to 3 inclusive.
- A sequence of *children*, of length equal to the arity (no children if arity is zero). Each child is, recursively, an EIDS tree.

Trees with arity zero, one, two, and three are respectively called nullary, unary, binary, and ternary.

Note that these "nonempty strings of Unicode characters" will very often tend to be of length one (single characters) but that is not a requirement. They cannot be empty (length zero); the case of a tree without a head is properly described by "there is no head," not by "the head is the empty string." *At present* no Unicode canonicalization is performed, that being left to the user, but this may change in the future. Zero bytes (U+0000) are in principle permitted to occur in EIDS trees, but because Unix passes command-line arguments as null-terminated C strings, they can only be entered in matching patterns via backslash escape sequences.

Typically, these trees are used to describe kanji characters. The literal Unicode character being described will be the head, if there is a code point for it; the functor will be either an ideographic description character like ⿱ if the character can be subdivided, or else nullary ； if not. Then the children will correspond to the parts into which it can be decomposed. Some parts of the character may also be available as characters with Unicode code points in their own right; in that case, they will have heads of their own.

EIDS syntax

Unicode's IDS syntax serves a similar purpose to IDSgrep's extended IDS syntax, but it lacks sufficient expressive power to cover some of IDSgrep's needs. Nonetheless, EIDS syntax is noticeably derived from that of Unicode IDSes. Broadly speaking, EIDSes are IDSes extended to include heads (which we need for partial-character lookup); bracketed strings as functors (which we need for capturing arbitrary data); and with arbitrary limits on allowed characters and length relaxed (needed for complex characters and so that matching patterns can be expressed in the same syntax).

Here are some sample EIDSes:

大
⿱田⿰虫⿱土土
⿸厂⿱今止
【萌】⿱艹<明>⿰日月
【店】⿸广<占>⿱卜口

```
⿱⺾⿰日?
&...男...女
[tb]⺾[or][lr]?日[lr]日?
```

The first three of these examples are valid in the Unicode IDS syntax. The next two contain heads, and are typical of what might exist in a dictionary designed to be searched by the `idsgrep` command-line utility. The last three might be matching patterns a user would enter.

EIDS trees are written in a simple prefix notation that could be called "Polish notation" inasmuch as it is the reverse of "reverse Polish notation." To write a tree, simply write the head if there is one, the functor, and then if the tree is not nullary, write each of the children. Heads and the functors of trees of different arity are (unless otherwise specified below) written enclosed in different kinds of brackets that indicate the difference between heads and functors, and the arity of the tree when writing a functor.

The basic ASCII brackets for heads and functors are as follows:

head	`<`	`>`	`<example>`
nullary functor (0)	`(`	`)`	`(example)`
unary functor (1)	`.`	`.`	`.example.`
binary functor (2)	`[`	`]`	`[example]`
ternary functor (3)	`{`	`}`	`{example}`

Note that the opening and closing brackets for unary functors are both equal to the ASCII period, U+002E.

Some sequences of Unicode characters beginning with "\" (ASCII backslash, U+005C) are treated specially. Backslash followed by a character from a short list of ASCII Latin letters introduces an escape sequence used to substitute for a character that would otherwise be hard to type; backslash followed by any other character (including a second backslash) is equivalent to the other character, but without any special meaning it would otherwise have had. Thus, backslash can be used for instance to include literally in a bracketed string the closing bracket that otherwise would mark the end of the string.

The backslash-letter escapes are listed below. Note that the letters identifying the type of escape sequence are case-sensitive, and all are lower-case except "`\X`." However, for sequences that take a parameter, the parameters are not case-sensitive. Note that all characters inside an escape sequence must be literal ASCII, except in the "default" case of a single backslash used to escape a single non-ASCII character. It is not permitted to use recursive backslash escapes to create some of the characters that make up a multi-character escape sequence like "`\x{}`."

`\a`	ASCII BEL (U+0007)
`\b`	ASCII BS (U+0008)
`\c`*X*	ASCII control character *X*
`\e`	ASCII ESC (U+001B)
`\f`	ASCII FF (U+000C)
`\t`	ASCII HT (U+0009)
`\n`	ASCII LF (U+000A)
`\r`	ASCII CR (U+000D)
`\x`*HH*	two-digit Unicode hex
`\X`*HHHH*	four-digit Unicode hex
`\x{`*Hx*`}` `\X{`*Hx*`}`	variable-length Unicode hex

The `\c` escape takes a parameter consisting of a single ASCII Latin letter character (only); it is equivalent to typing Ctrl plus that letter (case insensitive) on a standard keyboard, that is the ASCII control code in the range U+0001 to U+001A obtained by subtracting 64 from the uppercase letter's ASCII code or 96 from the lowercase letter's ASCII code.

The hexadecimal escapes `\x` and `\X` offer a choice of two-digit, four-digit, or variable-length (enclosed by curly braces) hexadecimal specification of Unicode code points. The hex codes are case-insensitive. Values greater than 10FFFF, and therefore outside the Unicode range, will be replaced by the Unicode replacement character U+FFFD.

Parsing of bracketed strings has a few features worth noting. First, there is no special treatment of nested brackets. After the "`<`" that begins a head, for instance, the next unescaped "`>`" will end the head, regardless of how many other instances of "`<`" have been seen. However, because no head or functor can be less than one character long, a closing bracket immediately after the opening bracket (which would otherwise create an illegal empty string) is specially treated as the first character of the string and *not* as a closing bracket. Thus, "`())`" is legal syntax for a functor equal to a closing parenthesis, in a nullary tree; and "`...`" is a functor equal to a single ASCII period in a unary tree, an important example because it is the commonly-used match-anywhere operator. A bracket character specified via a backslash escape, whether by preceding the literal character with a backslash or by giving its hexadecimal code in a "`\x`" or "`\X`" construction, is never taken to start or end a bracketed string.

Each pair of ASCII brackets also has two pairs of generally non-ASCII synonyms, as follows:

<	>	【	】	〖	〗
(	)	（	）	⦅	⦆
.	.	:	:	・	・
[	]	［	］	⟦	⟧
{	}	〔	〕	〘	〙

The closing synonymous brackets for functors of unary trees are always identical to the opening brackets. A string may be opened by any of the three opening bracket characters for its type of string; but then it must be closed by the closing bracket character that goes with the opening bracket. Brackets from other pairs are taken literally and do not end the string. For instance, “`【<example>】`” is a head whose value consists of “`<example>`” including the ASCII angle brackets. There are several reasons for the existence of the synonyms:

- They look cool.
- There is an established tradition of using 【lenticular brackets】 for heads in printed dictionaries, which is exactly their meaning here.
- Allowing ASCII colons to bracket unary-node functors makes possible a more appealing and `grep`-like syntax for `idsgrep`'s output in the case of processing multiple input files.
- Allowing more than one way to bracket each kind of string makes it easier to express bracket characters that may occur literally in a string.
- The non-ASCII brackets may be easier to type without switching modes in some input methods.
- On the other hand, keeping an ASCII option for every bracket type allows matching patterns to be entered on ASCII-only terminals.
- Multiple bracket types allow for creating human-visible computer-invisible distinctions in dictionary files, for instance to flag pseudo-entries that contain metadata, without needing to create a special syntax for comments.

If a character other than an opening bracket occurs unescaped in an EIDS where an opening bracket would be expected, it is treated in one of three ways.

- ASCII whitespace and control characters, U+0000 to U+0020 inclusive, are ignored. In the future, this treatment might be extended to non-ASCII Unicode whitespace characters, which are best avoided because of the uncertainty.
- Some special characters, such as “`⿰`,” have “sugary implicit brackets.” If one of these characters appears outside of brackets, it will be interpreted as a functor whose value is a single-character string equal to the literal character, and a fixed arity that depends on which character it is. For instance, “`⿰`” and “`[⿰]`” will be parsed identically. A list of characters getting this treatment is below.
- Any other non-bracket character has a “syrupy implicit semicolon.” That means it will be interpreted as a complete nullary tree with a single-character head equal to the literal character, and a single semicolon as the functor. For instance, “`x`” and “`<x>(;)`” will be parsed identically. Because semicolon itself has sugary implicit nullary brackets, we could also write “`<x>;`” for the same effect.

Here are all the characters that have sugary implicit brackets, with the brackets they imply: `(;)` `(?)` `.!.` `./.` `.=.` `.*.` `.@.` `.#.` `[&]` `[,]` `[|]` `[⿰]` `[⿱]` `[⿴]` `[⿵]` `[⿶]` `[⿷]` `[⿸]` `[⿹]` `[⿺]` `[⿻]` `{⿲}` `{⿳}`

Note that the sugary and syrupy implications of a character are only relevant when the character occurs where an opening bracket of some type would otherwise be required; inside a bracketed string, characters are taken literally unless they end the string or make up escape sequences. Characters created by escape sequences are always syrupy outside a string and always literal inside a string; they never start or end bracketed strings nor have any special sugary meaning they would otherwise have.

Characters, for the purposes of EIDS parsing, are strictly single Unicode code points. Such things as combining accents and variation selectors are parsed as separate characters from the bases to which they may be applied. The sugary and syrupy parsing rules apply only to single characters. Thus, appropriate brackets are necessary whenever a sequence containing more than one code point is to be treated as a single head or functor.

It is an intentional consequence of these rules that all syntactically valid Unicode IDSes are syntactically valid EIDSes, but the converse is not true. CHISE IDS extended IDSes can easily be converted to this syntax but in general are not valid IDSgrep EIDSes in themselves.

Although it is technically not a parsing issue but rather a transformation applied to the tree after parsing, there is one more issue to mention: some functors have aliases. If a functor and arity matches one of the aliases on the following list, it will be replaced with the indicated single-character functor. The idea is to provide verbose ASCII names for single-character functors of special importance to the matching algorithm. Note that the single-character versions are always the canonical ones, and although the brackets are shown explicitly for clarity, they are nearly all characters from the "sugary implicit" list. This feature may be disabled or modified using some settings of the "`-c`" command-line option; see the section on output cooking for more information.

`(anything)`	⇒	`(?)`	`.anywhere.`	⇒	`...`
`.not.`	⇒	`.!.`	`.regex.`	⇒	`./.`
`.equal.`	⇒	`.=.`	`.unord.`	⇒	`.*.`
`.assoc.`	⇒	`.@.`	`.user.`	⇒	`.#.`
`[and]`	⇒	`[&]`	`[or]`	⇒	`[|]`
`[lr]`	⇒	`[⿰]`	`[tb]`	⇒	`[⿱]`
`[enclose]`	⇒	`[⿴]`	`[wrapu]`	⇒	`[⿵]`
`[wrapd]`	⇒	`[⿶]`	`[wrapl]`	⇒	`[⿷]`
`[wrapul]`	⇒	`[⿸]`	`[wrapur]`	⇒	`[⿹]`
`[wrapll]`	⇒	`[⿺]`	`[overlap]`	⇒	`[⿻]`
`{lcr}`	⇒	`{⿲}`	`{tcb}`	⇒	`{⿳}`

The `idsgrep` command-line utility attempts to follow Postel's Law with respect to byte sequences that are not valid UTF-8: "be conservative in what you do, be liberal in what you accept from others." [13] Jesus of Nazareth stated a similar principle somewhat earlier.[*] Accordingly, invalid UTF-8 on input is not in general treated as a fatal error. Handling of invalid UTF-8 represents a delicate balance of security issues: if invalid UTF-8 is treated as completely fatal, that creates the possibility for denial of service attacks, but if it is permitted to too great an extent, it can create opportunities for things like buffer overflows. In general, the `idsgrep` utility will not itself break when given bad UTF-8, nor will it make matters worse compared to a system that did not include `idsgrep`, but `idsgrep` cannot be counted on to actively protect some other piece of software that would otherwise be vulnerable to bad UTF-8.[†]

The parser will skip over (as if they did not exist at all) byte sequences that are not valid UTF-8, including the forbidden bytes 0xC0, 0xC1, and 0xF5 through 0xFF; continuation bytes outside valid multibyte sequences; "overlong" sequences (those that would otherwise be valid, but encode a given code point other than in the shortest possible way); surrogates; and sequences that encode code points outside the Unicode range. Depending on where they occur within a multibyte sequence, some of these things may result in the whole sequence being skipped instead of just the bad bytes, with the parser making its best guess as to what that means. Be aware that some other software may treat some of these things as valid.

When a code point outside the Unicode range, or a surrogate, is specified using a backslash hexadecimal escape, the parser will interpret it as if the substitute character U+FFFD had been specified instead. All UTF-8 sequences *actually generated by* the `idsgrep` program are guaranteed to be valid UTF-8, barring serious programming errors; and matching operations including PCRE matches occur only on the parsed internal representation which is valid UTF-8. Note that PCRE, despite having a deprecated syntax for sub-encoding byte matching, *cannot* be used to detect invalid bytes that the `idsgrep` parser skipped; it sees only what the parser validly parsed. However, since in its default mode the `idsgrep` program will echo through to the output the exact input byte sequence that was parsed to create a tree, not the internal representation, it is possible that non-UTF-8 input could result in non-UTF-8 output. Several cooked output modes, in which `idsgrep` generates its own UTF-8 from the internal representation and provides guarantees of valid UTF-8 or even valid ASCII output, are available but non-default.

Some byte sequences that are valid UTF-8 but not valid Unicode, for instance the sequence that encodes a reversed byte order mark, may possibly go undetected in the input and be allowed in the output, even when cooked, by the current version of `idsgrep`. It is intended that `idsgrep` should detect that kind of thing where it is reasonable to do so, and future versions may do it better than this one does; but some higher-level errors in Unicode usage, such as misuse of combining characters or variation selectors, will probably never fall within the scope of `idsgrep`.

Matching

The basic function of the `idsgrep` command-line utility is to evaluate each item in the database against a matching pattern. The matching patterns are similar in spirit to the "regular expressions" common throughout the Unix world; however, for theoretical and practical reasons standard regular expressions would be unsuitable for the applications considered

---

[*] "There is nothing from without a man, that entering into him can defile him: but the things which come out of him, those are they that defile the man." (Mark 7:15, KJV)

[†] Genesis 4:9.

by IDSgrep.

The main theoretical issue is that IDSes, whether IDSgrep-style "extended" or Unicode-style traditional ones, belong to the class of *context-free* languages. They describe tree-like structures nested to arbitrary depth, similar in nature to programming-language expressions containing balanced parentheses although balanced parantheses as such are not actually part of EIDS syntax. The natural way to parse these involves an abstract machine with a stack-like memory that can assume an infinite number of different states. Regular expressions can only be used to recognize the smaller, simpler class of *regular* languages, parsable by an abstract machine with a finite-state memory. It is not possible to write a correct regular expression that will match balanced parentheses. Some advanced software implementations of so-called "regular expressions" (for instance, Perl's) contain special features that make them more powerful than the standard theoretical model, so that they are capable of recognizing some languages that are non-regular, including balanced parentheses. It is also possible to fake a stack with a finite depth limit by writing a complicated regular expression, and that may be good enough in some practical cases. Some users may also settle for just doing a substring query with `grep` and calling the result close enough. But IDSgrep tries to do it in a way that is really right, and that is described precisely in this section.

We will define a function $\mathit{match}(x, y)$ which takes two EIDS trees as input and returns a Boolean value of true or false. We call x the *pattern* or *needle* and y the *subject* or *haystack*. The `idsgrep` command-line utility generally takes x from its command line and repeatedly evaluates this function for each EIDS it reads from its input; it then writes out all the values of y for which $\mathit{match}(x, y)$ is true.

The $\mathit{match}(x, y)$ function is defined as follows:

- If x and y both have heads, then $\mathit{match}(x, y)$ is true if and only if their heads are identical. Nothing else is examined (in particular, not the children). Then the two cases below do not apply.

- If x and y do not both have heads, then $\mathit{match}(x, y) = \mathit{match}'(x, y)$, whose value generally depends on the functor and arity of x. The match' function has many special cases described in the subsections below, expressing different kinds of special matching operations. These operations roughly correspond to the ASCII characters with sugary implicit brackets in EIDS syntax. They are shown with brackets for clarity in the discussion below, but users would generally type them without the brackets and depend on the sugar in actual use.

- If none of the subsections below applies, then $\mathit{match}'(x, y)$ is true if and only if x and y have identical functors, identical arities, and $\mathit{match}(x_i, y_i)$ is true recursively for all their corresponding children x_i, y_i. Note that match' recurses to match, not itself, so there is a chance for head matching on the children even if it was not relevant to the parent nodes.

Match anything The value of $\mathit{match}'((?), y)$ is always true. Thus, ? can be used as a wildcard in `idsgrep` patterns to match an entire subtree regardless of its structure. Mnemonic: question mark is a shell wildcard for matching a single character. The verbose ASCII name for "`(?)`" is "`(anything)`."

Match anywhere The value of $\mathit{match}'(\texttt{...}x, y)$ is true if and only if there exists any subtree of y (including the entirety of y) for which $\mathit{match}(x, y)$ is true. In other words, this will look for an instance of x anywhere inside y regardless of nesting level. Mnemonic: three dots suggest omitting a variable-length sequence, in this case the variable-length chain of ancestors above x. The verbose ASCII name for "`...`" is "`.anywhere.`."

Match children in any order The value of $\mathit{match}'(\texttt{.*.}x, y)$ is true if and only if there exists a permutation of the children of y such that $\mathit{match}(x, y')$ is true of the resulting modified y'. For instance, `*[a]bc` matches both `[a]bc` and `[a]cb`. This is obviously a no-operation (matches simply if x matches y, as if the asterisk were not applied) for trees of arity less than two. Mnemonic: asterisk is a general wildcard, and this is a general matching operation. The verbose ASCII name for "`.*.`" is "`.unord.`."

NOT The value of $\mathit{match}'(\texttt{.!.}x, y)$ is true if and only if $\mathit{match}(x, y)$ is false. It matches any tree *not* matched by x alone. Mnemonic: prefix exclamation point is logical NOT in many programming languages. The verbose ASCII name for "`.!.`" is "`.not.`."

AND The value of $\mathit{match}'(\texttt{[\&]}xy, z)$ is true if and only if $\mathit{match}(x, z) \wedge \mathit{match}(y, z)$. In other words, it matches all trees that are matched by both x and y; the set of strings matched by `[&]`xy is the intersection of the sets matched by x and by y. Mnemonic: ampersand is logical or bitwise AND in many programming languages. The verbose ASCII name for "`[&]`" is "`[and]`."

OR The value of $\mathit{match}'(\texttt{[|]}xy, z)$ is true if and only if $\mathit{match}(x, z) \vee \mathit{match}(y, z)$. In other words, it matches all trees that are matched by at least one of x or y; the set of strings matched by `[|]`xy is the union of the sets matched by x and by y. Mnemonic: ASCII vertical bar is logical or bitwise OR in many programming languages. The verbose ASCII name for "`[|]`" is "`[or]`."

Literal tree matching If x and y both have heads, then the value of $\mathit{match}'(\texttt{.=.}x, y)$ is true if and only if those heads are identical. Otherwise, it is true if and only if x and y have identical functors, identical arity, and $\mathit{match}(x_i, y_i)$ is true for each of their corresponding children.

The effect of this operation is to ignore any special $\mathit{match}'()$ semantics of x's functor; the trees are compared as if that functor were just an ordinary string, regardless of whether it might normally be special. Note that the full $\mathit{match}()$ is still done on the children with only the root taken literally; to do a completely literal match of the entire trees it is necessary to insert an additional copy of `.=.` above every node in the matching pattern, or at least every node that would otherwise have a special meaning for $\mathit{match}'()$, and even then heads will continue to have their usual effect of overriding recursion.[‡] Mnemonic: equals sign suggests the literal equality that is being tested rather than the more complicated comparisons that might otherwise be used. The verbose ASCII name for "`.=.`" is "`.equal.`."

For instance, this feature could allow searching for a unary tree whose functor actually is `!`, where just specifying such a tree directly as the matching pattern would instead (under the rule for "NOT" above) search for trees that do not match the only child of `!`. In the original application of searching kanji decomposition databases this operation is unlikely to be used because the special functors do not occur anyway, but it seems important for potential applications of IDSgrep to more general tree-querying, because otherwise some reasonable things people might want to look for could not be found at all.

Associative matching The value of $\mathit{match}'(\texttt{.@.}x, y)$ is calculated as follows. Create a new EIDS tree x', initially equal to x, which has the property that its root may be of unlimited arity. Then for every child of x' whose functor and arity are identical to the functor and arity of x, replace that child in x' with its children, in order. Repeat that operation until no more children of x' have functor and arity identical to the functor and arity of x. Compute y' from y by the same process. Then $\mathit{match}'(\texttt{.@.}x, y) = \mathit{match}(\texttt{.=.}x', y')$.

This matching operator is intended for the case of three or more things combined using a binary operator that has, or can be said to sometimes have, an associative law. For instance, the kanji 怠 could be described by "⿱⿱厶口心" (⿱厶口 over 心) or by "⿱厶⿱口心" (厶 over ⿱口心). Unicode might encourage use of the ternary operator ⿳ for this particular case instead, but that does not cover all reasonably-occurring cases, and the default databases seldom if ever use the Unicode ternary operators.

The difference between the representations is sometimes useful information that the database *should* retain; for instance, in the case of Tsukurimashou, "⿱⿱厶口心," "⿱厶⿱口心," and "⿳厶口心" would correspond to three very different stanzas of MetaPost source code, and the user might want a query that separates them. On the other hand, the user might instead have a more general query along the lines of "find three things stacked vertically with 心 at the bottom" and intend that that should match both cases of binary decomposition. The at-sign matching operation is meant for queries that don't care about the order of binary operators; without it, matching will by default follow the tree structure strictly.

Note that even with `.@.`, IDSgrep will not consider binary operators in any way interchangeable with ternary ones; users must still use `.|.` to achieve such an effect if desired. Although the at-sign is fully defined for all arities, it is only intended for use with binary trees. Note also that `.@.` and `.*.` behave according to their definitions. Incautious attempts to use them together will often fail to have the desired effects, because the definitions do not include special exceptions that some users might intuitively

---

[‡] It may be interesting to consider how one could write a pattern to test absolute identity of trees, with each node matching if and only if its head or lack thereof is identical to the desired target as well as the functors and arities matching and the same being true of all children.

expect for these two operators happening to occur near each other. In a pattern like "`*@[a][a]bcd`," `.*.` will recognize `.@.` as the functor of a unary tree and expand the single permutation of its one child, and so that pattern will match the same things as if the asterisk had not been present, namely "`[a][a]bcd`" and "`[a]b[a]cd]`" but not, for instance, "`[a][a]dcb`." In a pattern like "`@[a]b*[a]cd`," `.@.` will recognize `.*.` as a different arity and functor from `[a]` and choose not to expand it in x', with the result that that pattern matches the same things as if the at-sign had not been present, namely "`[a]b[a]cd`" and "`[a]b[a]dc`" but not "`[a][a]bcd`" nor "`[a][a]bdc`."

When considered as an operation on trees, what `.@.` does is fundamentally the same thing as the algebraic operation that considers $(a+b)+c$ equivalent to $a+(b+c)$, and for that reason it is called "associative" matching. The mnemonic for at-sign is that it is a fancy "a" for "associative." The verbose ASCII name for "`.@.`" is "`.assoc.`."

Regular expression matching If x and y both have heads, then $\mathit{match}'(\texttt{./.}x, y)$ is true if and only if the head of x, considered as a regular expression, matches the head of y. If x and y do not both have heads, then $\mathit{match}'(\texttt{./.}x, y)$ is true if and only if x and y have the same arity, the functor of x considered as a regular expression matches the functor of y, and $\mathit{match}(x_i, y_i)$ is true for each of their corresponding children. This operation is basically the same as the default matching operation, except that regular expression matching is used instead of strict equality for testing the heads and functors. Mnemonic: slash means regular expression matching in Perl. Verbose ASCII name: "`.regex.`."

Regular expression matching for the purposes of this operator is as defined by the Perl Compatible Regular Expressions library, in whichever version was linked with the `idsgrep` utility. Strings are passed into PCRE as UTF-8, and are guaranteed (because the EIDS parser decodes and re-encodes `idsgrep`'s input for internal use) to be valid UTF-8 when PCRE sees them regardless of user input; as such, PCRE is given the option flags that make it read UTF-8 without doing its own validity check. Use of the PCRE "`\C`" syntax for matching individual octets within UTF-8 is strongly not recommended. All other PCRE options are left to the defaults chosen when PCRE was compiled, even if those are silly. The character tables are PCRE's "C locale" defaults, not generated at runtime from the current locale. Things like case sensitivity can be controlled within the pattern using PCRE's syntax for doing so. In the event that `idsgrep` was compiled without the PCRE library (which is not recommended, but is possible), or that PCRE was compiled without UTF-8 support, then an attempt to evaluate the slash operator will trigger a fatal error.

A matching pattern given to PCRE will have already passed through the EIDS parser, which removes one level of backslash escaping. The pattern may also have been passed as a command-line argument to `idsgrep` by a shell, which may have undone another level of backslash escaping. Thus, it may be necessary to escape characters as many as three times in order to match them literally with the slash operator. Each of these levels may differ from the others in terms of the escape sequences it supports and their exact meanings. In many cases it doesn't really matter which level of processing evaluates the escaping. For instance, "`idsgrep "/(\t)"`," (shell evaluates "`\t`," EIDS and PCRE see a literal tab); "`idsgrep "/(\\t)"`," (shell removes one backslash, EIDS evaluates "`\t`," PCRE sees a literal tab); and "`idsgrep "/(\\\\t)"`," (shell removes two backslashes, EIDS removes one, PCRE evaluates "`\t`") will all match the same things. If it matters, however, then caution is necessary.

PCRE because of the limitations of its API effectively forbids zero bytes (U+0000) in its matching patterns, whereas EIDS allows them to exist within strings in general. The complexities of PCRE pattern syntax make it impractical for `idsgrep` to automatically escape zero bytes before passing the strings to PCRE; there are too many different cases possible for the context in which a zero byte might occur. Since the `idsgrep` utility takes its matching patterns from the Unix command line anyway, and Unix itself forbids literal zero bytes in command-line arguments, the case of literal zero bytes in a matching pattern can only occur when they are created deliberately by escape sequences at the level of the EIDS parser; and the simplest advice to users is "don't do that!"

Python, which like EIDS allows strings to contain zero bytes but has PCRE bindings and so faces the same issue, briefly attempted to work around this PCRE API limitation by auto-escaping. They eventually gave it up as too complicated and confusing. The consequence of PCRE's API design is that if the string given as a matching pattern contains a literal zero byte then the regular expression to be matched will consist of the prefix of the string up to but not including the first zero byte; anything after that will be ignored. Zero bytes are, nonetheless, permitted in the matching subject, and PCRE can search for them, but not by means of literal zero bytes in the pattern. For instance, the PCRE syntax “`\000`” (or just “`\0`” if the next character will not be an octal digit) matches a zero byte. As discussed above, additional escaping might be needed to ensure that PCRE, and not EIDS nor the shell, interprets the backslash escape.

User-defined matching predicates It is assumed that by some out-of-band means, we have defined a family of functions $U_i()$ for i from 1 up to some k. These functions take EIDS trees as input and return Boolean values (hence “predicates”).

Then the value of $\mathit{match}'(\texttt{.\#.}x, y)$ is determined as follows. First, an integer i is computed. If x has a head, its initial characters will be parsed as an ASCII decimal number using the C library’s `atoi(3)` function; i is the resulting value, if it is positive. If x has no head, the head of x cannot be parsed, or the head of x is parsed as zero or negative, then i is defined to be 1. Having defined i, if $U_i()$ exists then $\mathit{match}' = U_i(y)$. If $U_i()$ does not exist then match' is false. Mnemonic: hash-mark is used for parameter substitution in languages such as TEX, and this matching operation causes the matching pattern to take something external (the user-defined predicate) as a parameter.

In the current version, the functions $U_i()$ are always defined using the “`-f`” command-line option (or its long-named equivalent) and correspond to the character coverage of TrueType or OpenType fonts. The predicate returns true if and only if y has a head consisting of a single Unicode character that is covered by the font.

Cooked output

The default mode of operation for the `idsgrep` command-line utility is that whenever a matching tree is detected, the exact sequence of bytes that were parsed to generate that tree (including no skipped whitespace before it, and all skipped whitespace after it but before the next tree) will be copied through to the output. This mode of operation is called “raw.” Raw mode is easy to understand, efficient, preserves distinctions like different kinds of brackets in the input, and is as analogous as reasonably possible to the operation of `grep`. However, preserving the exact input bytes may preserve invalid UTF-8, valid but weird EIDS syntax, or non-ASCII characters users may find difficult to type or display, that may have existed in the input. The “`-c`” (“`--cooking`”) command-line option provides a wide range of ways for `idsgrep` to generate new EIDS syntax of its own, guaranteed to be valid, from the internal representation generated by the parser. The cooked output modes force the output into a well-behaved format independent of what the input looked like. Input canonicalization (such as the translation from “`[lr]`” to “⿰”) can also be controlled through this interface.

The “`-c`” option can be given a (lowercase ASCII Latin, unabbreviated) keyword as its argument, to select a preset output mode. That is the only recommended way to use this option. The available preset modes are as follows:

`raw` Raw mode: write out the exact input byte sequence that was parsed to generate the matching tree, *even if it is not valid UTF-8.* This is the default.

`rawnc` Raw with no canonicalization: raw mode output, but without the canonicalization transformation during input parsing.

`ascii` ASCII-only: all non-ASCII characters and ASCII control characters are replaced by escape sequences or subjected to the reverse of the input canonicalization transformation, to produce a result that should pass through most limited-character-set channels. Note that the plainest ASCII space (U+0020) is not escaped in this mode when EIDS syntax does not require it to be. This mode generally uses a lot of hexadecimal escapes and, in a dictionary-lookup context, may be useful for finding the hexadecimal code point value of an unknown character.

`cooked` Generic cooked mode: render trees as reasonably clean and appealing Unicode text similar but not necessarily identical to what appears in the pregenerated dictionary files. This will escape characters outside the Basic Multicharacter Plane; characters in all Private Use Areas; and any other characters that EIDS syntax *requires* must be escaped; but no others. It will choose an appropriate escaping method depending on the type of character. Generally, it will use black lenticular brackets for top-level heads, ASCII brackets elsewhere, and syntactic sugar and syrup to avoid brackets where possible (except for top-level heads).

`indent` Write trees on multiple lines with two-space indentation to show their structure as clearly as possible. One blank line (two newlines) between trees. In other ways this is similar to "`cooked`."

If not given a preset keyword, "`-c`" can be given a string of ASCII decimal digits. The decimal-string interface allows precise control of how output syntax will be generated, but it is somewhat experimental, very complicated, and may change incompatibly in future versions of this software. Use of this feature is not recommended. Nonetheless, the remainder of this section will attempt to document it.

The format specifier may be up to twelve digits long. If it is shorter than that, it is taken as a prefix with unspecified digits copied from the default specifier, which is "`100000013250`" and equivalent to the "`cooked`" preset. The two raw presets are handled as special cases; of the remaining cooked presets, "`ascii`" is equivalent to "`000000013551`" and "`indent`" is equivalent to "`100000223250`."

The first digit specifies the type of brackets to be used for the head of the root of the tree: 0 for "`<>`," 1 for "【】," or 2 for "〖〗." The second digit specifies the type of brackets for the head of any non-root node, using the same code.

The third digit specifies the type of brackets for nullary functors: 0 for "`()`," 1 for "（）," or 2 for "⦅⦆." Similarly, the fourth digit specifies the brackets for unary functors: 0 for "`..`," 1 for "`::`," or 2 for "・・"; the fifth digit specifies the brackets for binary functors: 0 for "`[]`," 1 for "［］," or 2 for "⟦⟧"; and the sixth digit specifies the brackets for ternary functors: 0 for "`{}`," 1 for "〔〕," or 2 for "〘〙".

The seventh digit describes how to insert newlines and indentation to pretty-print the tree structure. If it is 0, that will not be done. If it is 8, trees will be pretty-printed using one tab character per level; the number eight is a mnemonic for the fact that people generally expect those to be equivalent to eight spaces each. Any other decimal digit specifies that many spaces per level.

The eighth digit specifies the separator printed between trees: 0 for a null byte (U+0000), 1 for a newline, 2 for two newlines, or 3 for no separator at all.

The ninth digit specifies the circumstances under which the sugary and syrupy features of EIDS syntax should be used. It is a sum of binary flags: add 4 to use a syrupy semicolon when possible at the top level; 2 to use a syrupy semicolon when possible at other levels; and 1 to use sugary implicit brackets wherever possible.

The tenth digit specifies which characters should be escaped. Literal backslashes, and (within a bracketed string) literal instances of the close-bracket character that would otherwise end the string, must always be escaped. When the tenth digit is 0, those are the only characters that will be escaped. Other values add escaping for the following categories of characters, and do so cumulatively with each digit also escaping everything that would be escaped by all lesser digits.

1 Escape characters from the astral planes; that is, characters with code points greater than U+FFFF and thus outside the Basic Multilingual Plane.

2 Escape characters from the BMP Private Use Areas, U+E000 to U+F8FF. The other Private Use Areas are already escaped at level 1 by virtue of being outside the BMP.

3 Escape all non-ASCII characters (U+0080 and up) except the core Unified Han range (U+4E00 to U+9FFF).

4 Escape the core Unified Han range.

5 Escape the ASCII control characters (U+0000 to U+001F).

6 Escape closing brackets at the start of bracketed strings, which otherwise escape escaping because of a special case in the syntax definition.

7 Escape all characters. Depending on the value of the next digit, however, the ASCII Latin alphabet still might not be escaped.

The eleventh digit specifies *how* to escape whatever characters were selected for escaping by the tenth digit. The available values are as follows.

0 Use a single backslash followed by the literal character, only. The ASCII Latin alphabet cannot be escaped in this way and under this option, or options 1 or 5 which fall through to this case, will not be escaped at all. Since the literal characters remain in the text, this option is not suitable for sending output through any channel that is not clean for the full range of UTF-8 characters. However, unlike raw mode, this and all other cooked modes do guarantee to produce valid UTF-8, not arbitrary byte sequences.

1 Use a backslash-letter sequence for ASCII control characters U+0001 to U+001B, and otherwise follow option 0.

2 Use variable-length hexadecimal "`\x{}`" sequences for all characters that are selected to escape. This syntax can escape any character.

3 Use two-digit "`\x`*HH*" sequences wherever possible (that is, for ASCII and ISO-8859-1 characters), four-digit "`\X`*HHHH*" sequences for other characters on the Basic Multilingual Plane, and variable-length hexadecimal sequences otherwise.

4 Use four-digit "`\X`*HHHH*" sequences wherever possible (that is, for all characters on the BMP), and variable-length hexadecimal sequences otherwise.

5 Attempt to choose the simplest type of escape for each character depending on its value, just like option 3 except with backslash-letter escapes where possible (U+0001 to U+001B) and backslash-literal escapes for ASCII non-control characters (U+0020 to U+007E excluding the Latin alphabet). The ASCII Latin alphabet will not be escaped at all under this option.

The twelfth digit specifies canonicalization processing; that is, the translations on both input and output between alphabetic functor aliases like "`(anything)`" and their symbolic equivalents like "`(?)`." Note that in all cases the symbolic versions are the matching operators; if you disable input canonicalization and enter a matching pattern of "`(anything)`" it will be matched as an ordinary nullary functor containing a string of eight ASCII letters, not as the match-anything operator which is always named "`(?)`." The digit value is a sum of binary flags: add 4 to *disable* the default transformation of alphabetic aliases to symbolic names on input; plus 2 to enable a translation from alphabetic aliases to symbolic names on output, which is generally only meaningful if 4 was selected; plus 1 to enable a transformation from symbolic names back to alphabetic aliases on output.

Bit vector indices

Executive summary: if you leave it on the default settings and skip this chapter, it should just work.

In more detail: matching trees according to the rules in the previous chapter is potentially an expensive operation. The cases of tree-matching actually used in practice tend to be relatively easy ones, but parsing EIDS syntax is also expensive enough, even with an optimized implementation, that it may take as much time as matching or more; and every tree of the database must be parsed in order to attempt a match. A complete installation of IDSgrep may take a second or two on a typical PC to parse and search all the dictionaries, which is not long enough to be a problem in typical interactive use, but could become an issue if queries were being generated automatically, faster than a single user would type them.

The solution is to avoid doing tree matches, and avoid doing parsing, as much as possible. As of version 0.4, IDSgrep is capable of analysing a dictionary in advance and generating an index file representing useful information about the trees in a format that can be scanned quickly. It is not possible to correctly answer all queries solely from information in the index, but the hope is that for most queries, the program can rule out most dictionary entries as potential matches just from a fast examination of the index. Then for any entries not excluded by the index, it runs the more expensive parsing and tree matching operations on the actual data. The time saved by skipping input entries is supposed to more than compensate for the additional work of reading the index.

Exactly how this feature works is complicated, and is part of the author's academic research. This chapter starts with a short summary from a user's perspective of how to build and use bit vector indices. After that, all remaining sections are marked with dangerous bend symbols, and attempt to give some notes on the technical details for interested parties without claiming to really be a complete presentation of the mathematical underpinnings of the system. Those sections should not be read by anyone who is easily frightened. Watch for future publications covering this material more formally.

Building and using bit vector indices

If you run `make install` to install dictionaries, then the build system should build, install, and use bit vector indices for the installed dictionaries automatically.

Getting the most out of bit vector indexing requires building the `idsgrep` command-line utility with the BuDDy binary decision diagrams library available at `http://sourceforge.net/projects/buddy/` [11]. Without it, bit vectors will still provide some speed improvement, but not as much.

Bit vector indices properly used are expected to increase the speed of searching by about a factor of 15 in typical cases. The improvement factor varies a lot depending on a number of issues, and could be a thousand or more under optimal conditions. It should never be significantly less than one; that is, searching with a bit vector index should never take significantly longer than searching without one.

Bit vectors provide the greatest benefit when the query is simple (exact-matching a single syrupy character is best); when the dictionary entries themselves are small; when the query, regardless of its form, only matches a small number of results; when the query includes exact matching of heads or functors at the root level of the EIDS tree (such as a query starting with "`[lr]`") or in the root's immediate children; and when the query does not include special matching operators such as regular expressions and user-defined predicates.

Whenever the `idsgrep` utility reads a file whose pathname ends in "`.eids`"—regardless of whether that file was specified explicitly on the command line or indirectly via the `-d` option—it will look for an index file whose pathname is the same except with the `.eids` extension changed to `.bvec`. If such a file exists, can be read, has the correct 8-byte magic number at the start, *and has a timestamp no older than the timestamp of the* `.eids` *file*, then `idsgrep` will assume it is a bit vector index and use it to speed up the query process. Note that all those conditions must be met. If any of the conditions fail to be met, no error will be reported, but the scanner will be forced to read and parse the entire input file without using bit vector filtering. Once the `idsgrep` utility commits to start reading the index file past the header, it cannot switch to index-free searching and errors after that point will abort the search, just like errors in the EIDS input file.

To create a bit vector index, use the `-G` option to `idsgrep`, as in `idsgrep -G dictionary.eids > dictionary.bvec`. Something like this will be necessary if the dictionary file changes or if you create a new one of your own. An outdated index file will usually be locked out by the timestamp check, but if you force the issue (for instance, by changing the timestamps with `touch`), the search will most likely abort with a parsing error when it hits the changed part of the dictionary file.

Bit vector indices are only usable when reading from files with names ending in `.eids`. When reading from standard input or similar, or even just from files without `*.eids` filenames, bit vectors will not be used. Bit vectors are also not applicable to the internally-generated Unicode list associated with the `-U` option, although a somewhat similar feature (generating only the at most one entry that could match, if the query tree has a head) is always used where it applies. If you direct the `idsgrep` utility to read from multiple input sources in the same run, it will use bit vector indices on whichever input sources it can, even if that is not all of them.

You can force `idsgrep` to ignore bit vector indices with the `-I` option; that is unlikely to be useful except during speed tests, but one could maybe imagine a case where it's absolutely necessary to have a file named `*.bvec` which is not a bit vector index and must not be touched, or where even looking for the index file incurs undesired traffic on a network filesystem.

Using a (valid) bit vector index, or not using one, should only affect speed. It should never change which results are or are not returned from a query. If you manage to find a case where the same query and the same input file produce different hits depending on whether `-I` is used or a bit vector index generated by the same `idsgrep` binary, then that may be evidence of a bug; please report it.

Filtered matching

Let $\mathbb{E}$ be the set of EIDS trees. Let T and F represent Boolean true and false. The previous chapter defined a function $\mathit{match} : \mathbb{E} \times \mathbb{E} \to \{\mathsf{T}, \mathsf{F}\}$ for determining whether one EIDS tree matches another; if $\mathit{match}(N, H) = \mathsf{T}$ we say that N, which we call the *needle*, matches H, which we call the *haystack*.

The `idsgrep` binary has the job of evaluating $\mathit{match}(N, H)$ for one N specified on the command line and every H in the dictionary. The dictionary is large, but does not change frequently, whereas each invocation of `idsgrep` faces only one value of N, but it is fresh and may never have been seen before. And the *match* function is inconveniently expensive to calculate.

We will attempt to deal with this situation by defining three new functions $\mathit{filt} : \mathbb{E} \to \mathbb{F}$, $\mathit{vec} : \mathbb{E} \to \mathbb{V}$, and $\mathit{check} : \mathbb{F} \times \mathbb{V} \to \{\mathsf{T}, \mathsf{F}\}$. Elements of $\mathbb{V}$ that come out of *vec* are called *vectors* and elements of $\mathbb{F}$ that come out of *filt* are called *filters*. We want the following properties:

- *vec* might be expensive to compute, but we can store elements of $\mathbb{V}$ (its output) reasonably cheaply.
- *filt* might be relatively expensive to compute, and its output might be large, but we can at least afford to compute it, and store the result, once per run of `idsgrep`.
- *check* is cheap to compute.
- If $\mathit{match}(N, H)$ is true, then $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H))$ is definitely true. Therefore if $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H))$ is false then $\mathit{match}(N, H)$ must also be false.
- If $\mathit{match}(N, H)$ is false, then $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H))$ is usually false. This might not be guaranteed, but we want it to hold as often as possible.

With those properties, what we can do is run *vec* on all the trees in the dictionary and store the results in the index ahead of time. When the `idsgrep` binary runs, it can run *filt* just once on the input search pattern. Then for each entry in the index, it calculates $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H))$. That is supposed to be a cheap operation. If *check* returns true, then the dictionary entry *might possibly* be a match. At that point `idsgrep` can extract the location and length of the original tree data from the index, read the relevant chunk of the dictionary proper, parse it, and calculate *match* to determine whether it really is a match. We hope that when the dictionary entry is not really a match, *check* will usually return false. Whenever *check* is false, we can be sure that *match* would also return false, and so `idsgrep` can skip over that entry without doing the work of parsing or actually computing *match* the hard way.

This idea of doing an easy approximate check first, to save the cost of a more difficult accurate evaluation, should be quite familiar. Imagine a hiring committee quickly throwing out all the job applications from software engineers, then interviewing the computational linguists. Similar concepts appear throughout computer science: consider instruction and data caches; memoization for dynamic programming; solving relaxed versions of optimization problems; and especially Bloom filters [4]. The bit vector technique used in IDSgrep builds on Bloom filters and on the work of Skala and others [17] and Skala and Penn [18] on unification of types in logic programming systems.

Functional programming weenies will note that the real point of an element of $\mathbb{F}$ is that you can combine it with *check* to make a tasty curry: elements of $\mathbb{F}$ are important because of the functions $\mathbb{E} \to \{\mathsf{T}, \mathsf{F}\}$ they beget upon *check*. When the needle is a complicated EIDS tree with a lot of special matching operators in it, we hope that we can find filters for the subtrees and then combine them in some simple way to find a filter for the complicated needle. This pursuit is called the *filter calculus*. It is always possible, at least a little, because we could just declare the result of the filter calculus to be a filter that makes *check* return T identically on all vectors. That obeys all the properties it must obey. But of course we hope for filters to be as restrictive as possible, to allow us to rule out as many dictionary entries as possible, so the accept-everything filter is a last resort.

IDSgrep (when compiled with the necessary library) actually uses two layers of filtered matching, here called lambda filters and BDD filters. Both layers share the same vectors—that is, $\mathbb{V}$ and *vec* are the same for both. For each entry in the index, the `idsgrep` utility checks the lambda filter. If the lambda filter returns false, it stops. If the lambda filter returns true, it tries the BDD filter, and if that returns false, it stops. Only if both filters return true will it try parsing the tree and really calculating the *match* function. The filters are arranged in order of increasing cost: the lambda filter is expected to run faster than the BDD filter, which is expected to run much faster than the parse and tree test.

In IDSgrep 0.4, $\mathbb{V}$ is the space of 128-bit binary vectors. Each vector is conceptually divided into four 32-bit words; the actual implementation, intended for a 64-bit microcomputer, involves a two-element array of `uint_64` integers.

Vector values are calculated by the *vec* function (called `haystack_bits_fn` in its implementation in `bitvec.c`) as follows. The first (least significant) 32 bits of the vector are a Bloom filter encoding the head, and the functor and arity combined, of the root of the EIDS tree. Depending on a hash of the head, three of the bits are set to ones. They are chosen without replacement (unlike the bits in the classic Bloom filter) so it really is three distinct bits set to one—no collisions at this level. If there is no head, a special combination of three bits is set, corresponding to the hash of an empty string. Similarly, three bits are set depending on the functor and arity; these can collide with the head bits but not with each other. So among the first 32 bits of the vector, somewhere between three and six bits will be set in a pattern that depends on the head, functor, and arity of the root of the tree. This is a little less than the optimal density for a Bloom filter this size considered in isolation (from information theory: one wants the filter to end up with approximately equal numbers of ones and zeros), but because of the more complicated things happening elsewhere in the system, the reduced density here seems to work better.

The second 32-bit word encodes the head, functor, and arity of the first child of the root, using the same scheme. This is left zero for a nullary root. The third 32-bit word similarly encodes the *last* (not necessarily second) child of the root. When the root is unary, that means the only child gets encoded into *both* the second and third words of the overall vector. Finally, the middle child of a ternary root, and all grandchildren and lower descendants of any root, are all encoded into the fourth 32-bit word, all bitwise ORed on top of one another. Generate a bit vector index with debugging turned on, redirecting standard error to a file, to see what these vectors actually look like.

Lambda filters

Consider a very simple needle that matches if and only if the head of the haystack has a certain value. For instance, `<foo>!?`. That matches anything with the head "`foo`" under the head-to-head matching rule, and nothing else because `!?` (the inverse of the match-everything query) matches nothing. This is not typical of the queries users actually write, but we will build up to the more complicated behaviour of realistic queries.

Given the bit vector for a haystack tree, we could look at just the three bits in the first 32-bit word that (according to the hash function) correspond to "head equal to `foo`." If the head of the haystack is `foo`, then those bits will definitely all be ones. Otherwise, we expect them to behave like three bits chosen from a 32-bit word in which at most six bits chosen at random have been set, and the chances of *all three* randomly being ones under those circumstances is roughly $(6/32)^3 \approx 0.7\%$. (Not exactly, because of collisions and stuff, but that's a fair estimate.) So a decent thing for *check* to do, given this query, would be "look at the three bits associated with the head `foo`, and return T if more than two (i.e., all three) of them are set, F otherwise." That would have the desired properties of returning T for sure when the query matches, and not being very likely to return T under other circumstances.

Let $\mathbb{F}$ be the set of ordered pairs (m, λ) where m is a 128-bit bit vector called the *mask* and λ is a nonnegative integer called the *threshold*. We call these pairs *lambda filters*. Define $\mathit{check}((m, \lambda), v)$ to compute the bitwise AND of the mask m and vector v, count how many bits are set in the result, and then return T if and only if the count is strictly more than the threshold λ. The filter for `<foo>!?` will consist of a mask that selects the three bits for `foo`, and a threshold of 2. It matches if and only if those three bits are all set, which is definitely true for haystacks that match the needle and reasonably unlikely to be true for haystacks that do not match the needle. So far this is just the usual lookup algorithm for a Bloom filter. But by choosing different masks and thresholds, we can also do other kinds of filtering.

Suppose we have two filters (m_1, λ_1) and (m_2, λ_2). Let m_3 be the bitwise OR of m_1 and m_2, and let λ_3 be the minimum of λ_1 and λ_2. If a given vector contains more than λ_1 bits selected by m_1, then all those bits are also in m_3, so (m_3, λ_3) will also match. The same is true for (m_2, λ_2); so if either of the original filters matched, then the combined filter must match. In this way we can find a filter for the OR of two existing filters.

Note that the OR filter is not an if and only if: there could be combinations of bits that hit part of m_1 and part of m_2, so that neither (m_1, λ_1) nor (m_2, λ_2) would match but (m_3, λ_3) matches. We have lost some precision in the filter calculus, and if we kept ORing filters like this we would eventually end up with a saturated mask that matches everything. But we have not lost any recall. If $\lambda_1 \neq \lambda_2$, we can improve things a little by removing bits from whichever filter has the greater λ until the two thresholds become equal; that slows down the saturation a little in some cases.

Combining filters with AND is more complicated because there are multiple choices for the result. Given that there are more than λ_1 bits set out of m_1, and more than λ_2 bits set out of m_2, we can conclude various things about the number of bits that must be set in the intersection and each set difference of the two masks. For instance, knowing more than two bits out of four are set, and more than two bits out of a different set of four that overlaps in two places with the first set, we can conclude that more than zero bits of the two-bit intersection must be set; or more than three of the six-bit union. Both those, and several others, would be acceptable results for the AND operation in the filter calculus. IDSgrep attempts, heuristically, to guess which of these filters will be most useful and return the best one as the result of ANDing two lambda filters.

When we push a subtree further down in the EIDS tree, its filter changes. Suppose we have a filter that would be correct for a given needle if it were the root, but it actually appears as the right child of a binary root. Any bits in its mask that would have queried the first word (head, functor, and arity of root) must now query the third word (head, functor, and arity of last child). We can easily rearrange those bits in the mask. However, if the mask selects any bits from the second through fourth words when the needle is at the root, then when it is shifted down the tree all those bits must come from the last word of the haystack's vector. They could collide with each other. Then a haystack that formerly would have been hit by the mask in three places might only be hit once, because all three of those bits collided. The value of λ might need to be reduced by as much as a factor of three. IDSgrep computes all possible collisions and tries to use as large a value of λ for the modified filter as it can prove will still give correct results. Similar considerations apply to the operations of moving a needle's filter from the root to the left or middle children.

Given these operations of OR, AND, and moving a needle that refers to the root to refer to a child instead, IDSgrep can find a lambda filter for basic matching of any tree. The rules like "if there is a head, then it must match exactly, and if not, then the functor, arity, and all children must match" translate into filter calculus operations and can be applied recursively.

Special matching operators require additional handling. The `&` (Boolean AND) and `|` (Boolean OR) operators are easy. The `!` (Boolean NOT) operator unfortunately isn't: all we can say in pure filter calculus is that any filter might correspond to a match that might fail later, so the NOT of *any* filter has to be the match-everything filter. The IDSgrep implementation of NOT goes slightly beyond pure filter calculus by looking at the actual needle instead of only the needle's recursively-computed filter; so it is smart enough to recognize `!!` as a no-op, `!?` as match-nothing, and to apply de Morgan's laws to AND and OR. The unordered match operator `*` translates easily into the OR of all permutations of its child, and the literal match operator `=` is straightforward. Other match operators, such as regular expression, have semantics too complicated to easily handle, so the filter calculus just returns the match-everything filter for them.

Most of the output from `--bitvec-debug` consists of an indented trace of the lambda filter calculus operations performed while analysing the user's query.

BDD filters

Lambda filters do not capture everything we might want to know about the bits in a vector. Consider the filters (with vector length limited to four) $(0011, 1)$ and $(1100, 1)$. The OR of those filters in the lambda filter calculus will be $(1111, 1)$. But that will match if *any* two bits are set; the original filter would only match if the first two or the last two are set. We have lost some precision, which translates to eventually doing more tree matches than necessary, in exchange for the guarantee that the result of ORing two lambda filters will always be a lambda filter. BDD filters attempt to lose less precision by allowing the results of filter calculus operations to be more complicated. In exchange for this greater power, a lot of data structures have to be maintained in the background, and that is why we apply lambda filters first: lambda filter misses allow us to skip the greater work of testing the BDD filters, and BDD filter misses allow us to skip the even greater work of parsing and doing real tree matching.

A Binary Decision Diagram (BDD) is basically the most boring "Choose Your Own Adventure" ever. Every page just says "If binary variable such-and-such is true, then go to this other page; otherwise, go to that other page." Many of these options collide with each other, so that even though there may be trillions of paths through the BDD there are only two endings, neither of which involves Gorga the Space Monster. Spoiler: the two endings are "1" and "0."

More mathematically, a BDD is a data structure that represents a function that takes some bits as input and returns one bit of output. It can represent any binary function if you're willing to give it enough time and memory, and it is relatively easy to perform logic operations on functions represented as BDDs. Donald E. Knuth is a great fan of BDDs.

Let $\mathbb{F}$ be the set of BDDs on 128 variables. Define $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H))$ to simply evaluate the BDD $\mathit{filt}(N)$ on the 128 bits of $\mathit{vec}(H)$. The vector $\mathit{vec}(H)$ is as defined previously; it remains to define $\mathit{filt}(N)$.

For the query `<foo>!?`, which matches if and only if the head is `foo`, recall that there are three bits in the first word of the vector that will necessarily all be set by matching haystacks, and only rarely all set by non-matching haystacks. We can easily create a BDD that returns 1 if and only if all three of those bits are set. So far this is just the same as the corresponding lambda filter.

The difference is in the BDD filter calculus operations. We can take the OR of any two BDDs and the result will be a BDD that returns 1 if *and only if* at least one of the inputs would return 1. The equivalent operation on lambda filters would only be an if, not an if and only if; but with BDD filters there is no loss of precision in OR. Similarly, we can do AND without loss. Those two operations alone account for much of the filter calculus done in practice, and BDDs represent these functions exactly. The only loss of precision is in the original Bloom filter encoding.

Once we start shifting needles from the root to the children, more precision is lost because of bit collisions, just as with lambda filters. The BDD may be looking for two bits that both ended up in the same place, so if that one bit is set then the BDD sees both its inputs true even though really, without the collision, only one would be true. The NOT operation remains a problem too: in any pure filter calculus where the result of NOT depends only on the filter that was the input to NOT, because filters can really only return "maybe" and not exactly "yes," the result of NOT must always be identically true regardless of the input. Just as with lambda filters, IDSgrep looks at the needle itself, not purely at the filter, to get better results from NOT. More complicated operations, like match-anywhere, have reasonably straightforward implementations.

Memoization in the tree match

This feature does not involve bit vectors at all, but is described in the bit vector chapter because, like bit vectors, it is a speed enhancement that doesn't change the basic matching algorithm. There is no "r" in "memoization." That word is a shibboleth[*] for CS theorists.

For the most part, when a match is not ruled out by the bit vector indices, IDSgrep uses a straightforward recursive-descent algorithm to implement the tree matching function *match*. That works well in practice for the kinds of queries typically encountered. However, it is possible to construct pathological queries on which the recursive matching algorithm will misbehave. For instance, a query with many nested instances of `.anywhere.` can bog down attempting to match against a deep tree, as it tries all possibilities for the intermediate nodes that anchor the different `.anywhere.` operators, even though the choices are all equivalent. Matching according to IDSgrep's matching rules should in fact be possible in polynomial time, because we can match each node in the needle tree once against each node in the haystack tree. Assuming the match function is well-defined we only need do that once per pair of nodes; so this suggests a dynamic programming algorithm. The dynamic programming algorithm is unlikely to be a good idea in the usual case, because of excessive overhead maintaining the table and the possibility of doing extra, unnecessary matches if we're not careful. Recursive descent, with short-circuiting of unnecessary subtree tests once the answer is known, seems to perform much better in practice in ordinary cases despite its exponential worst-case bound. Nonetheless, to cover bad cases that may occur in the input, IDSgrep implements memoization of tree matches when it appears likely to be helpful.

The matching operators that could cause trouble are "`...`" (`.anywhere.`) and "`.*.`" (`.unord.`). All other special matching operators, and the default matching rules, recurse at most once into each child; but match-anywhere attempts to match its needle tree once against every descendant of its haystack tree, and unordered match attempts to match its needle once against each of the up to six permutations of its haystack's children. The time to do matching by recursive descent ends up having an exponent determined by the number of uses of those operators in the matching pattern.

So IDSgrep counts how many times either of those operators occurs in the matching pattern, and uses that count to determine both whether memoization would help, and how big a table size to use. The memoized match is straightforward. Before testing a needle tree against a haystack tree, IDSgrep checks whether that pair of needle and haystack is recorded in the memoization hash table. If it is, the answer is taken from the table instead of calculated fresh. Otherwise, once it has been calculated, the result goes in the table.

Implementation details

Storing bit vectors requires storing two `uint64_t`s and an `off_t`, with possible alignment padding, per entry. That works out to 24 bytes on typical newer 64-bit and 32-bit platforms. Some older 32-bit platforms with 32-bit `off_t`s may only require 20 bytes per index entry, trading off the decreased space against likely problems should a dictionary ever grow larger than 4G. For comparison, the average length of a dictionary entry in the CHISE, KanjiVG, and Tsukurimashou dictionaries as of this writing is about 40 bytes; four times that for EDICT2. So storing the indices represents a significant, but not crippling, increase in storage requirements. The performance improvement numbers do account for the time taken to read the additional data from disk—it just isn't enough to be a problem compared to the savings in parsing and tree matching.

The format of the bit vector index file is specific to the version of IDSgrep and the computer architecture including the C compiler. If you use a bit vector file with a mismatched `idsgrep` binary, it will almost certainly be detected as invalid and ignored. Building with or without BuDDy will not make a difference to the file format; the difference is a different search algorithm applied to the same data.

It is possible that the magic numbers may not perfectly track incompatible changes among different development versions of IDSgrep that you might check out of SVN, but they should definitely work correctly to differentiate between incompatible versions that have been formally released.

One issue for BDD filters that did not occur with lambda filters is that there is no practical limit to the size of a BDD on 128 bits, so it could grow until it exhausts memory or consumes so much time for filter calculus operations that it ends up giving no benefit over just using the match-everything filter and falling back on the original tree-matching algorithm. In order to prevent this kind of failure, bearing in mind that it is always safe to change a filter to return hits on *more* vectors as long as it doesn't lose any existing hits, IDSgrep keeps an eye on the size of the BDDs

---

[*] Judges 12:6.

returned by filter calculus operations. If a BDD exceeds 1000 nodes, then IDSgrep applies existential quantification to shrink it.

For a chosen variable, existential quantification changes the BDD to one that will return true if and only if *there exists any value* of the chosen variable that would (given the values of the other variables) allow the result of the BDD to be true. A roughly equivalent operation on a lambda filter might be to remove a bit from the mask and subtract one from λ, necessarily making the filter looser. It may not be obvious that doing this will necessarily make a BDD simpler, but after we apply such a quantifier, no more nodes referring to the chosen variable can remain in the BDD. The number of variables supporting the BDD necessarily decreases. Do it to every variable and the BDD must end up identically true (unless it was identically false to begin with, in which case it remains so). So by existentially quantifying variables in turn, we can guarantee that the BDD will at some point become smaller than 1000 nodes.

The threshold of 1000 was chosen by educated guess and by running some test suite queries (those from `test/speed` and `test/kvg-grone`) to see the effect on the overall running time of using different thresholds. For the typically small queries in `test/speed`, the final BDDs end up with an average of 47 nodes each; increasing the threshold provides steady speed improvement from about 10 to 100, and has little or no effect on `test/speed` after that. The single query in `test/kvg-grone` can potentially generate a very large BDD (at least 70000 nodes), but for that query having the BDD doesn't help much anyway. A threshold of 1000 nodes is the point at which the overhead of creating the BDD for `test/kvg-grone` starts to become significant (10% slower overall than the smallest thesholds, where the BDD is forced to triviality in negligible time). So this seems a good compromise point. It provides performance about as good as any other tested value for the queries from `test/speed`, which are expected to be typical of actual use; it provides some headroom (a factor of 10) over the minimum sufficient for those queries; and it is still small enough to prevent pathological or malicious queries, like the one in `test/kvg-grone`, from slowing the system down significantly.

As BDDs in BDD-using applications go, these are quite small ones. It appears that that is because the functions IDSgrep computes with BDDs are monotonic functions, which seem to create very good cases for the data structure. It also means that the BuDDy library's lack of negated edges is no problem. A brief attempt to tighten the filters by making the functions non-monotonic had to be aborted; it caused the time and space spent on filter calculus to blow up exponentially, even with careful attention to the variable ordering, in exchange for only very small improvements in the precision of the filters. Better to spend the time doing a few more tree matches. It also appears that I was lucky with my choice of variable ordering. The default ordering that resulted from using the bits in the vector straight through from LSB to MSB happens to give good BDDs (as long as they are monotonic). That means it's unnecessary to put a lot of effort into automatic reordering of the variables to optimize the BDD calculations.

The particular case of many `.anywhere.` operators nested inside one another with no heads and no other nodes in between might be detected and automatically simplified. Any number of those will match the same set of trees as a single such operator, so the extras can be simply removed without affecting the semantivs. But it is easy to construct more elaborate queries that still take exponential time without memoization and that the system cannot reasonably simplify. Users are unlikely to do that by accident, but now that I've mentioned the possibility, someone will try.

Let k represent the number of `.anywhere.` and `.unord.` operators in the matching pattern. It is actually determined by looking at the reference counts for the strings consisting of a single ASCII period and a single ASCII asterisk in an internal string table, so there may be an overcount if those strings happen to occur as heads or as non-unary functors. If k is less than three, memoization is not expected to be helpful, and so is not done. Then assuming k was at least three, if it is less than ten it is set to ten and if it is greater than 22 it is set to 22, and then memoization will be done with a hash table of 2^{k+1} entries. The idea here is to make the hash table grow as the number of entries needed grows, but always at least 2048 entries because there is little benefit from making it very small, and never more than 8M entries because if very large, memory becomes a bigger problem than time. The constants here were chosen by informal experiment, to switch over to memoized matching at the level of query complexity where it starts to be useful, and keep the tables just large enough that making them significantly larger provides little benefit.

Collisions are resolved by just overwriting the old entries; and the table gets emptied on each new top-level tree match. Table entries have generation numbers, so the emptying is done in constant time by incrementing the generation instead of really rewriting the table.

The keys used for the hash table are the actual pointers. Thus, it is possible for the performance of this feature to be nondeterministic on some platforms. There are some additional wrinkles in the code because of the life cycle of pointers to tree nodes: the nodes created during parsing of input live at least as long as hash table generations, but there can be a few nodes created and deleted on the fly during matching, and those cannot be saved in the hash table lest the pointers be reused and invalidated. It happens that the parser already set a flag in each node it created, for its own internal purposes in determining when the node was ready to come off the parsing stack. Nodes created on the fly during matching lack that flag. So the tree match memoization looks for the parser's flag to determine which nodes it is allowed to cache.

The format of the statistics line generated by the `--statistics` option is space-separated fields; the first is "`STATS`" and then the rest are mostly decimal numbers, in this order:

- bit vector (lambda filter) checks;
- lambda filter hits;
- BDD hits (necessarily zero if BDDs not compiled in; the number of BDD *checks* when BDDs are used is always exactly the value of the previous field and thus not reported separately);
- tree checks (may be greater than bit vector hits, because of unindexed input which skips directly to the tree checking step);
- tree hits (these result in output of matched trees);
- memoization checks (may be much larger than number of tree checks, because memoization happens inside the recursion of the tree check, but only on sufficiently complicated needles);
- memoization hits;
- user CPU time (reported as seconds with a decimal fraction down to microsecond precision as in the `struct rusage`, but your operating system probably rounds these numbers to 1/100 or 1/1000 of a second);
- node count in the BDD (zero if none was used or the feature is absent); and
- the query tree, in cooked EIDS format.

Bibliography

[1] CHISE project. Online `http://www.chise.org/`.

[2] GlyphWiki. Online `http://en.glyphwiki.org/wiki/GlyphWiki:MainPage`.

[3] Ulrich Apel. KanjiVG. Online `http://kanjivg.tagaini.net/`.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] Jim Breen. The EDICT dictionary file. Online `http://www.csse.monash.edu.au/~jwb/edict.html`.

[6] Jim Breen. WWWJDIC: Online Japanese Dictionary Service. Online `http://www.csse.monash.edu.au/~jwb/cgi-bin/wwwjdic.cgi`.

[7] Alexandre Duret-Lutz. Using GNU Autotools. Online `http://www.lrde.epita.fr/~adl/dl/autotools.pdf`.

[8] Free Software Foundation. GNU Grep 2.9. Online `http://www.gnu.org/software/grep/manual/grep.html`.

[9] Philip Hazel. Pcre—perl compatible regulat expressions. Online `http://www.pcre.org/`.

[10] Jason Katz-Brown. The Kiten Handbook, revision 1.2. Online `http://docs.kde.org/development/en/kdeedu/kiten/index.html`.

[11] Jørn Lind-Nielsen. BuDDy: A BDD package. Online `http://buddy.sourceforge.net/manual/main.html`.

[12] 守岡知彦 [Morioka Tomohiko]. UTF-2000 プロジェクト [The UTF-2000 Project]. 漢字と情報 *[Kanji and Information]*, (2):4–6, March 2001. In Japanese. Online `http://www.kanji.zinbun.kyoto-u.ac.jp/publications/kanji-and-info-2.pdf`.

[13] Jon Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Online `http://www.ietf.org/rfc/rfc793.txt`.

[14] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30. USENIX, 2005.

[15] Matthew Skala. Tsukurimashou Font Family and IDSgrep. Online `http://tsukurimashou.sourceforge.jp/`.

[16] Matthew Skala. Tsukurimashou Github repository. Online `http://github.com/mskala/Tsukurimashou`.

[17] Matthew Skala, Victoria Krakovna, János Kramár, and Gerald Penn. A generalized-zero-preserving method for compact encoding of concept lattices. In *48th Annual Meeting of the Association for Computational Linguistics (ACL 2010), Uppsala, Sweden, July 11–16, 2010*, pages 1512–1521. Association for Computational Linguistics, 2010.

[18] Matthew Skala and Gerald Penn. Approximate bit vectors for fast unification. In *The Mathematics of Language: 12th Biennial Conference (MOL 12), Nara, Japan, September 6–8, 2011*, volume 6878 of *Lecture Notes in Artificial Intelligence*, pages 158–173. Springer, 2011.

[19] Unicode Consortium. Ideographic description characters. In *The Unicode Standard, Version 6.0.0*, section 12.2. The Unicode Consortium, Mountain View, USA, 2011. Online `http://www.unicode.org/versions/Unicode6.0.0/ch12.pdf`.

[20] Ben Wing et al. XEmacs: The next generation of Emacs. Online `http://www.xemacs.org/`.