IDSgrep, version 0.2

Matthew Skala

March 17, 2012

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 [5] 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 [6] and WWWJDIC [3].

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 escaping, 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.

Introduction

The Han character set is open-ended. Although a few thousand characters suffice to write the languages most commonly written in Han script languages (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 [8]; 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` [5] 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 忄 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]??忄'`). This software is intended to bring the user-friendliness of `grep` to Han character dictionaries.

What's new

The main new features in version 0.2 are:

- implementations of all the planned matching operators except `@` (associative) and `/` (regular expression);
- a full test suite and some fixes for bugs found while creating it; and
- the EDICT2-derived dictionary, and the binary comma sugar character to support it.

Download, build, test, and install

IDSgrep is distributed under the umbrella of the Tsukurimashou project on Sourceforge.JP [8], `http://en.sourceforge.jp/projects/tsukurimashou/`. 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 [9], 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-0.2.tar.gz
cd idsgrep-0.2
./configure
make
su -c 'make install'
```

IDSgrep as such does not include a dictionary, but it 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/ [1], or (only if KanjiVG is also available) from the EDICT2 database available at http://www.csse.monash.edu.au/~jwb/edict.html [2]. 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. 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 KanjiVG and EDICT2 in the current directory and Tsukurimashou in a sibling 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 .
./configure
make
make check
su -c 'make install'
```

If the default search fails, the filenames of KanjiVG (.xml or .xml.gz), EDICT2 (.gz), and the top directory of Tsukurimashou can be specified on the configure command line with the --with-kanjivg, --with-edict2, and --with-tsuku-build options. For other options, run configure --help. It's a reasonably standard GNU Autotools [4] configuration script, albeit with a lot of options for inapplicable installation directories removed to simplify the help message.

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 [7] if available, to check for memory-related problems; if Valgrind is not found in the PATH, this test will be skipped.

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 an IDSgrep binary built under this option.

Interface to KanjiVG

The KanjiVG project [1] 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 radicals, and it attempts to follow the official stroke order and etymological radical 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 口, 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 [2] 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, planned future regular-expression matching features may make it more practical to search EDICT2 with IDSgrep, and even in the current version, there is some value in being able to do sub-character structural searches on multi-character words.

If both EDICT2 and KanjiVG are available to the IDSgrep build system, it will invoke the `ed2eids` script and generate and install a dictionary file called `edict.eids`, which represents a database join of the two dictionaries. 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 `kanjivg.eids` 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 -ded '<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 -ded ',教,育?'`. These queries can be combined with the sub-character breakdown queries already supported by the KanjiVG-based dictionary. For instance, `idsgrep -ded ',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.

Since both EDICT2 and KanjiVG are under the Creative Commons Attribution–ShareAlike license, that license presumably also applies to the combined dictionary made from them.

Interface to Tsukurimashou

IDSgrep is closely connected with the Tsukuimashou font family [8]. 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 and Tsukurimashou are distributed as separate packages, because they have very different audiences and build prerequisites. Many people who can use one will be unable to use the other, so it seems inappropriate to force all users to download both.

When IDSgrep's `configure` script runs, it looks for a valid Tsukurimashou build directory. Ideally, that

would be one in which Tsukurimashou has actually been fully built; but a directory where the Tsukurimashou `configure` script has been executed is enough. If a valid Tsukurimashou build directory is found automatically or specified with the `--with-tsuku-build` option to `configure`, then when `make` is run on IDSgrep, it will recursively go call `make eids` in the Tsukurimashou build. 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.

Note that neither Tsukurimashou nor IDSgrep is a true "sub-package" of the other in the sense of Autotools [4], as mediated by the `SUBDIRS` Automake variable and so on, notwithstanding that a checked-out SVN working copy of Tsukurimashou will contain a working copy of IDSgrep in a subdirectory. Running the Tsukurimashou build will not invoke the IDSgrep build at all; and running the IDSgrep build is not a good way to trigger a full Tsukurimashou build, because it won't use the preferred `-j` option, track all dependencies in detail, nor generate anything that doesn't happen to be a prerequisite for the files IDSgrep needs. If you want to build both systems, it's best to build Tsukurimashou first and then build IDSgrep pointing at Tsukurimashou. Also, these two packages do not necessarily have the same portability considerations, and it's possible that the link between them may fail even on systems where each package builds correctly by itself (for instance, possibly on some systems where GNU Make is installed but non-default). The link between Tsukurimashou and IDSgrep provides some convenience for my own frequent case of making changes to both packages at once.

In order for IDSgrep to work together with Tsukurimashou, it is necessary that the Tsukurimashou build be one that supports the `make eids` target in the first place. No released version contains such support yet, but it is planned for Tsukurimashou 0.6. Development versions of Tsukurimashou in the SVN repository have included EIDS support since early January 2012.

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 [10].

- 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, "⿳糸言糸夂" describes the character 變.
- An IDS may not be more than 16 code points long overall nor contain more than six consecutive non-operator characters. This rule appears to be 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.

Invoking idsgrep

The command-line idsgrep utility works much like most other command-line programs, and like grep [5] 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 implemented.

Command-line options

-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.

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

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

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 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.

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 called, respectively, 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 at present there is no way to enter them in matching patterns because Unix passes command-line arguments as null-terminated C strings.

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>⿰口口
⿰艹⿴日?
&...男...女

[tb]+[or][lr]?[lr]?

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.

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 ">" 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, "0)" 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.

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 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 behave normally.

It is planned that in the future, idsgrep's parser will also recognize some backslash escape sequences. This is not yet implemented.

It is a consequence of these rules that all syntactically valid Unicode IDSes are syntactically valid EIDSes, but the converse is not true.

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.

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

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 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 $match(x, y)$ 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 $match(x, y)$ is true.

The $match(x, y)$ function is defined as follows:

- If x and y both have heads, then $match(x, y)$ 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 $match(x, y) = match'(x, y)$, 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 $match'(x, y)$ is true if and only if x and y have identical functors, identical arities, and $match(x_i, y_i)$ 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.

Very few of the features below actually exist in the alpha version 0.2. The others are documented to give readers some idea of planned future development.

Match anything The value of $match'((?), y)$ 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 $match'(\ldots x, y)$ is true if and only if there exists any subtree of y (including the entirety of y) for which $match'(x, y)$ 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 $match'(.*.x, y)$ is true if and only if there exists a permutation of the children of y such that $match(x, y')$ 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 $match'(.!.x, y)$ is true if and only if $match(x, y)$ 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 $match'([\&]xy, z)$ is true if and only if $match(x, z) \wedge match(y, z)$. 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 $match'([|]xy, z)$ is true if and only if $match(x, z) \vee match(y, z)$. 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 $match'(.=.x, y)$ 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 $match(x_i, y_i)$ is true for each of their corresponding children.

The effect of this operation is to ignore any special $match'()$ 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 *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 $match'()$, 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

*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.

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 $match'(\texttt{.@.}x, y)$ 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 $match'(\texttt{.@.}x, y) = match(\texttt{.=.}x', y')$.

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 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.`"

This feature is not yet implemented in version 0.2.

Regular expression matching It is planned that some future version (likely version 0.3) will support special behaviour for $match'(\texttt{./.}x, y)$ to call a regular expression library and do string matching within heads or functors, but the detailed semantics of how that will work are not yet decided. The mnemonic is that slash is Perl's regular-expression match operator; the motivating application is to further development of IDSgrep's own dictionary-generating programs, which tend to create long nullary functors full of debugging information when they encounter constructs they don't understand in the other-format dictionaries they read.

Bibliography

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

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

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

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

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

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

[7] 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.

[8] Matthew Skala. Tsukurimashou Font Family and IDSgrep. Online http://en.sourceforge.jp/projects/tsukurimashou/.

[9] Matthew Skala. Tsukurimashou github repository. Online http://github.com/mskala/Tsukurimashou.

[10] 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.