combine Manual

Flexible Data Manipulation
Edition 0.3.2 for Release 0.3.2

by Daniel P. Valentine

This manual is for `combine` (version 0.3.2).

No publication data available yet.

combine Manual

This manual is for combine version 0.3.2. It provides general information for processing data files with the combine utility.

1 Overview of combine

1.1 Introduction

combine is primarily a program for merging files on a common key. While such things as table joins on keys are common in database systems like MySQL, there seems to be a marked failure in the availability for such processing on text files. combine is intended to fill that gap.

Another way of looking at combine is as the kernel of a database system without all the overhead (and safeguards) associated with such systems. The missing baggage that appeals most to me is the requirement to load data into somebody else's format before working with it. combine's design is intended to allow it to work with most data directly from the place where it already is.

In looking around for existing software that wanted to do what I wanted to do, the closest I came was the command join in GNU and other operating systems. The join command has some limitations that I needed to overcome. In particular, in matching it works on only one field each in only two files. For such a match, on files whose fields are separated by a delimiter, I'm sure that join is a more efficient choice. Someday I'll test that assumption.

Once I started writing the program, I had to come up with a name. Given that one of the earliest applications that I imagined for such a program would be to prepare data for storage in a data warehouse, I thought to where the things that are stored in physical warehouses come from. At that point, I came up with the name DataFactory. Unfortunately, just as I got ready to release it, I noticed that someone else has that as a registered trademark.

As a result, I have come up with the name combine. Like the farm implement of the same name, this program can be used to separate the wheat from the chaff. I also like it because it has a similarity to join reminiscent of the similarity of function.

1.1.1 Ways to Use combine

Here are some almost real-world applications of combine that may whet your imagination for your own uses.

1.1.1.1 Encrypted Data

Suppose you have a file that you would normally want to keep encrypted (say, your customer information, which includes their credit card numbers). Suppose further that you occasionally receive updates of some less sensitive information about the customers.

To update the sensitive file, you could unencrypt the file, load all the data from both files into a database, merge the data, extract the result from the database (making sure to wipe the disk storage that was used), and re-encrypt. That leaves the information open to DBAs and the like during the process, and it sure seems like a lot of steps.

Using combine and your favorite encryption program, the data never has to touch the disk unencrypted (except in a swap file) so casual observers are unlikely to run across sensitive information. In addition an updated encrypted file can be created in one command line, piping the input and output of the sensitive data through your encryption program.

Here's a sample command that might do the trick.

```
gpg -d < secret_file \
| combine -w -o 1-300 \
              -r update_information -k 1-10 -m 1-10 -o 11-20 -p \
| gpg -e > updated_secret_file
```

1.1.1.2 Key Validation

Another example that is often important when receiving data from the outside world is data validation.

Perhaps you receive product sales data from a number of retailers, which you need to combine with product information you have elsewhere. To confirm that the file you received is valid for reporting, you might need to check the product codes, ZIP codes, and retailer codes against your known valid values.

All the comparisons can be done in a single command, which will result in a status which will flag us down if anything did not match to our lists of expected values.

```
combine -w -o 1- \
            -r products.txt -k 1-18 -m 11-28 \
            -r zip.txt -k 1-5 -m 19-23 \
            -r customer.txt -k 1-10 -m 1-10 \
            input.txt \
  | cmp input.txt
result=$?
```

That's probably enough if we are pretty sure that the incoming data is usually clean. If something does not match up, we can investigate by hand. On the other hand, if we expect to find differences a little more frequently, we can make some small changes.

The following command makes the match optional, but puts a constant '$' at the end of the record for each match among the three keys to be validated. When there isn't a match, `combine` puts a space into the record in place of the '$'. We can then search for something other than '$$$' at the end of the record to know which records didn't match.

```
combine -w -o 1- \
            -r products.txt -k 1-18 -m 11-28 -k '$' -p \
            -r zip.txt -k 1-5 -m 19-23 -k '$' -p \
            -r customer.txt -k 1-10 -m 1-10 -k '$' -p \
            input.txt \
  | grep -v '\$\$\$$' > input.nomatch
```

1.2 Reporting Bugs

When you find a bug, please be so kind to report it in a way that will help the maintainer to understand why it is a bug and how to make it happen (so that the correction can be tested). The information necessary to make such a determination will include such things as a description of the symptoms (not just your impression of the cause), the command line you used to run the program, the options that were used when building the program, and anything else that you are not absolutely sure is irrelevant. You can also include the things that you are absolutely sure are irrelevant, as one is often sure without good reason.

Thank you for taking advantage of this free software. Your information on bugs in your use of the software are very valuable in keeping it as useful as possible for all users. Please report bugs to <dpv@world.std.com>

2 Invoking `combine`

`combine` is invoked with the command

```
combine [general options | data file options]... [reference file options]...
[data files]
```

The various options are described in the following sections.

The arguments to the command are data files, an optional list of file names to be processed or nothing, indicating that the input will come from `stdin`.

2.1 General Options

The general options are not specific to any particular file and apply to the entire current run of the program. They are listed below.

'`-R`'
'`--check-all-reference`'
'`--no-check-all-reference`'

For each data record, check for a match with each of the reference files. This option is set by default. When turned off with '`--no-check-all-reference`' and a match to at least one reference file is required for the data file record to be considerd for output, processing of that data file record will be stopped as soon as a required match is not fulfilled. This could increase speed somewhat and will change the meaning of any flags or counters set in reference-file-based input.

'`-f`'
'`--flag`'

In any output based on a reference file, print a flag with the value '`1`' if the record was matched by a data record or '`0`' otherwise.

'`-n`'
'`--count`'

In any output based on a reference file, print a new field which is a count of the number of data records that matched the current reference record.

'`-z` *`number`*'
'`--counter-size=`*`number`*'

If counters or sums of data fields are being written into reference-based output, make them *number* bytes long.

'`--statistics`'
'`--no-statistics`'

At the end of the processing, write to `stderr` counts of the number of records read and matched from each file processed. The option is enabled by default. When turned off with '`--no-statistics`' print no statistics.

'`--verbose`'

Write all the information that is possible about the processing to `stderr`. This includes a summary of the options that have been set and running counters of the number of records processed.

2.2 File Options

There are some attributes that apply to any file. They include such things as the way to tell one record from another, the way to identify fields within a record, and the fiewlds in a file that you want to write out.

‘`-d` *`string`*’
‘`--output-field-delimiter=`*`string`*’

Print *string* between each pair of fields written to output files. This can be a comma, a tab character, your mother’s maiden name, or the full text of the GNU Public License. It is your choice. Of course, it is most useful for it to be something that would not normally occur in your data fields and something that your target system will understand. Because `combine` doesn’t care about such things, you can even use the null character (`\0`) or the newline character (`\n`).

‘`-b` *`string`*’
‘`--output-record-delimiter=`*`string`*’

Similarly here *string* is whatever string you need to use to tell the records apart in output files. If you don’t specify it, `combine` will assume that the usual text format applies, the newline on GNU operating systems.

‘`-D` *`string`*’
‘`--input-field-delimiter=`*`string`*’

Look for *string* in the associated input file to determine where fields start and end. It can be any string, but it should be the string that really separates fields in your file. Because `combine` doesn’t care about such things, you can even use the null character (`\0`) or the newline character (`\n`).

‘`-B` *`string`*’
‘`--input-record-delimiter=`*`string`*’

Similarly here *string* is whatever string you need to use to tell the records apart in input files. If you don’t specify it, `combine` will assume that the usual text format applies, the newline on GNU operating systems.

‘`-L` *`number`*’
‘`--input-record-length=`*`number`*’

If the records are all of the same length (whether of not there is a linefeed or a ‘`W`’ between them), you can specify a record length. `combine` will then split the file into records all of exactly *NUMBER* bytes long. If there are linefeeds or carriage returns between records, be sure to add them to the count. If the length is not specified, `combine` will assume the records have a delimiter between them.

‘`-p`’
‘`--match-optional`’

Make it unnecessary to find a match to this file for inclusion in data file based output and for the further processing of a data record. All matches are required by default.

When applied to a reference file, this means that records based on a data record may be written without a match to a key in this reference file, and any output from fields in this file will be left blank (or substituted if specified with the '-O' option).

When applied to a data file, this means that in the file based on the data file records, records will be written at the end of processing for reference file records that were not matched by a data file record. Any output specified from fields in the data file will be left blank (or substituted if specified with the '-O' option). If there is more than one reference file, care should be taken, because the unmatched records from each reference file will each become a record in the output file. This has no effect on reference-file-based output.

'-P'
'--exclude-match'

Require no match to this file for inclusion in data file based output and for the further processing of a data record. All matches are required and included by default.

When applied to a reference file, this means that records based on a data record may only be written when there is no match to a key in this reference file, and any output from fields in this file will be left blank (or substituted if specified with the '-O' option).

When applied to a data file, this means that in the file based on the data file records, records will only be written at the end of processing for reference file records that were not matched by a data file record. Any output specified from fields in the data file will be left blank (or substituted if specified with the '-O' option). If there is more than one reference file, care should be taken, because the unmatched records from each reference file will each become a record in the output file. This has no effect on reference-file-based output.

2.3 Data Files

A data file is either stdin, one or more files, or both. If there is more than one source, they should all share a format that fits the specifications provided for output fields, fields holding numeric values to be summed, and fields to be used as keys for matching.

The following are the commands that specifically affect the processing of the data file. Some share the same name with options that can also apply to reference files. Those must be on your command line before you present your first reference file. The others could appear anywhere on the command line; however, convention dictates that all the data-file options should be together before the first reference file is named.

'-w'
'--write-output'

Signals the program that output records should be written every time a data record satisfies the matching criteria with the reference files. The record written will combine all specified output fields from the data file record, the matching reference file record, and any specified constant values. This option is a positional option for all files, and must appear before the first reference file is named.

'`-t` *filename*'
'`--output-file=`*filename*'

If provided, write the data-based output to *filename*. Otherwise the output will go to `stdout`. This option only makes sense if you plan to write data-file-based output. This is a positional option for all files and must appear before the first reference file is named.

'`-o` *range_string*'
'`--output-fields=`*range_string*'

Write the fields specified by *range_string* as part of the record in any data-based output. The range specifications share a common format with all field specifications for `combine`. This option only makes sense if you plan to write data-file-based output. This is a positional option and to apply to the data file it must appear before the first reference file is named.

'`-K` *string*'
'`--output-constant=`*string*'

Write *string* to the data-file-based output. This option only makes sense if you plan to write data-file-based output. This is a positional option and to apply to the data file it must appear before the first reference file is named.

'`-s` *range_string*'
'`--sum-fields=`*range_string*'

In any reference-file-based output write a sum of the value of the fields specified by *range_string*. The sum for a given reference file record will come from all data file records that matched it. This option only makes sense if you have requested reference-file-based output.

If a precision is provided for a sum field, that precision is honored, and decimal fractions up to that many digits will be included in the sum. Any further precision on the input fields will be ignored (without rounding) in the sum. The resulting sum will be written with a decimal point and the number of fractional digits implied by the precision.

2.4 Reference Files

A reference file record is expected to match on a set of key fields to a data file record. The parts of a reference file that are necessary for processing are read entirely into memory. You can specify as many reference files as you want, depending only on the amount of memory your system can spare to hold them. For any reference file, it is minimally required that you specify a file name, a specification of the key fields in the reference file, and a specification of the matching key fields in the data file.

The following are the options that are related to reference files. They are all positional, and they apply to the processing of the previously named reference file. (Except of course for the reference file name itself, which applies to itself.)

'`-r` *`filename`*'
'`--reference-file=`*`filename`*'

Use *filename* as a reference file to match to the data file in processing. This option introduces a block of positional options that relate to this reference file's processing in DataFactoy.

'`-k` *`range_string`*'
'`--key-fields=`*`range_string`*'

Use the fields specified by *range_string* as a key to match to a corresponding key in the data file.

'`-m` *`range_string`*'
'`--data-key-fields=`*`range_string`*'

Use the fields specified by *range_string* as the corresponding key to a key taken from a reference file.

'`-a` *`range_string`*'
'`--hierarchy-key-fields=`*`range_string`*'

Use the fields specified by *range_string* as a key to perform a recursive hierarchical match within the reference file. This key will be matched against values specified in the regular key on the reference file.

'`-u`'
'`--unique`'

Keep only one record for the reference file in memory for each distinct key. By default `combine` maintains all the records from the reference file in memory for processing. This default allows for cartesian products when a key exists multiple times in both the reference and data files.

'`-h` *`number`*'
'`--hash-size=`*`number`*'

Use the *number* provided as a base size for allocating a hash table to store the records from this reference file. If this number is too small, `combine` will fail when it tries to record a record it has no room for. If it is only a little bit too small, it will cause inefficiency as searching for open space in the hash table will be difficult.

'`-H` *`keyword`*'
'`--hash-movement=`*`keyword`*'

One of the keywords `binary`, `number`, `beginning`, or `end`, indicating how to turn the key into a number with the best variability and least overlap. The wise choice of this option can cut processing time significantly. The binary option is the default, and treats the last few bytes (8 on most computers) of the key string(s) as a big number. The number option converts the entire key to a number assuming it is a numeric string. The other two take the least significant 3 bits from each of the first or last few (21 where a 64 bit integer is available) bytes in the key strings and turns them into a number.

'-w'
'--write-output'

Signals the program that output records should be written for every record stored for this reference file. This will either be one record for every record in the reference file or one record for every distinct set of keys in the reference file, depending on the setting of the option '--unique'. The record written will include all specified output fields from the reference file record, any specified constant value for this reference file, and any flag, counter, or sums requested.

'-t *filename*'
'--output-file=*filename*'

If provided, write the output based on this reference file to *filename*. Otherwise the output will go to stdout. This option only makes sense if you plan to write output based on this reference file.

'-o *range_string*'
'--output-fields=*range_string*'

Write the fields specified by *range_string* as part of the record in any reference-file- or data-file-based output. The range specifications share a common format with all field specifications for combine.

'-K *string*'
'--output-constant=*string*'

Write *string* to the reference- or data-file-based output.

'-U'
'--up-hierarchy'

When traversing the hierarchy from a given reference-file record, use the values on that record in the '--hierarchy-key-fields' fields to connect to the '--key-fields' fields of other records from the reference file. For most purposes, the presence of the connection on the first record suggests a single parent in a standard hierarchy. The hierarchy traversal stops when the '--hierarchy-key-fields' fields are empty.

If this option is not set, the '--key-fields' fields are used to search for the same values in the '--hierarchy-key-fields' fields of other records in the same file. This allows multiple children of an initial record, and suggests going down in the hierarchy. The hierarchy traversal stops when no further connection can be made. The traversal is depth-first.

'-l'
'--hierarchy-leaf-only'

When traversing a hierarchy, treat only the endpoints as matching records. Nodes that have onward connections are ignored except for navigating to the leaf nodes.

'-F *number*'
'--flatten-hierarchy=*number*'

When traversing a hierarchy, act as the 'hierarchy-leaf-only', except save information about the intervening nodes. Repeat the 'output-fields' fields *number* times (leaving them blank if there were fewer levels), starting from the first reference record matched.

2.5 Output Files

There are two basic kinds of output files: one based on the data records and reference records that match them, the other based on a full set of records from one reference file with flags, counts, or sums based on the aggregate of the matching data records.

The output file based on the data file consists of information from the data file records and any matching reference file records. The records that go into data-based output files can be figured out as follows:

no reference file

If there is no reference file, there will be one record for every record in the data file, with the exception of any records that were elimitated through an extension filter. (see Chapter 4 [Extending combine], page 16.)

reference files with '`--unique`' and '`--match-optional`' options

If all reference files are specified with the '`--unique`' and '`--match-optional`' options, then the records selected for insertion into the data-based output file will be the same as those that would be selected without a reference file.

reference files without the '`--unique`' option

If a reference file is not given the '`--unique`' option and there is more than one reference record that matches a given data record, then the data record will be represented more than once in the output file, each time combined with information from a different matching reference record. If there is more than one reference file where this is the case, the result will be multiplicative (e.g. 2 matches in each of 2 reference files will produce 4 records). This is the default setting for a reference file.

reference files without the '`--match-optional`' option

If a reference file is not given the '`--match-optional`' option, then any data record that does not have a match in the reference file will not be represented in the output file. This is the default setting.

The fields that can appear in data=file-based output can come from the data-file record and any matching reference file records.

Reference=file-based output files are simpler. Depending on the existence or not of the '`--unique`' option, the file will have an entry for each of the unique keys or for each of the records in the reference file, respectively.

The fields in the reference=file-based output are exclusively from the reference file, except for optional fields that can be summarized from fields on matching data-file records.

The order of the fields in an output record can either be according to the default or it can be explicitly specified by the user.

In data=file-based output, the standard field order is as follows. All the fields listed are printed in this order if they are specified. If an output field delimiter is specified, it is put between every pair of adjacent fields. If there is no match for a given reference file (and the '`--match-optional`' option is set for the file), all the fields that would normally be provided by that file are filled with spaces for fixed-width fields or zero-length for delimited output.

- All the data-file output fields (in order)
- The constant string set for the data file
- For each reference file
  - The constant string set for the reference file
  - All the reference-file output fields

In reference=file-based output, the standard field order is as follows. All the fields listed are printed in this order if they are specified. If an output field delimiter is specified, it is put between every pair of adjacent fields.

- All the reference-file output fields OR the key fields if no output fields are given
- A 1/0 flag indicating whether there was any match
- A counter of the number of data records matched
- A sum of each of the data-file sum fields from each matching data-file record

The order of the fields in any output file can be customized using the '`--field-order`' (or '`-O`') option. The argument for the option is a comma-separated list of field identifiers. Each field identifier has 2 parts, a source and a type, separated by a period (.).

The sources are composed of an '`r`' for reference file or '`d`' for data file followed by an optional number. The number indicates which reference file the field comes from and is ignored for data files. Without a number, the first of each is taken.

A third source '`s`' represents a substitution in the event that the preceding reference file field could not be provided because there was no match between that reference file and the data file. The number following it, if blank or zero, tells Datafactory to take the field from the data file. Any other number means the corresponding reference file. This allows the conditional update of fields from the data file, or a prioritization of selections from a variety of reference files. If you are working with fixed-width fields, you should ensure that the lengths of the various fields in the substitution chain are the same.

The types are composed similarly. The identifiers are listed below. The number is ignored for identifiers of string constants, flags, and counters. For output fields, a hyphen-separated range of fields can be written to avoid having to write a long list. Any number provided is the number of the field in the order it was specified in the '`-o`' or '`-s`' option on the command line. In delimited-field files this may differ from the field number used in those options.

'`o`' Output fields from either reference or data files.

'`k`' String constant for either reference or data files.

'`f`' Flag (1/0) for reference files.

'`n`' Counter for reference files.

'`s`' Sum field for reference files.

Here is an example:

```
--field-order d.o1,d.o2,d.o3,d.k,r1.o1,s2.o1,s0.o4

--field-order d.o1-3,d.k,r1.o1,s2.o1,s0.o4
```

In this case, the first three fields from the data file are followed by the constant string from the data file. Then, if there was a match to reference file 1, the first field from that file is taken, otherwise if there was a match to reference file 2, the first field from that file is taken. If neither file matched the data record, the fourth field from the data record is taken instead.

The second line is equivalent, using a range of fields for convenience.

2.6 Field Specifiers

There are a number of options that require a list of fields from one of the files. All take an argument consisting of a comma-separated list of individual field specifications.

The individual field specifications are of the form '`s-e.p(instruction)`'. As a standard (with no field delimiter given for the file), '`s`' is a number indicating the starting position of the field within the given file, and '`e`' is a number pointing out the ending position. If there is no hyphen, then the single number '`s`' represents a one-byte field at position '`s`'. If the first number and a hyphen exist without the number '`e`', then the field is assumed to start at '`s`' and extend to the end of the record. The numbering of the bytes in the record starts at 1.

If you do provide a delimiter for identifying the borders between fields in your file, then `combine` does not need to count bytes, and you just need to provide the field number (starting at 1) of the fields you want to declare. Currently, giving a range of fields results in `combine` acting as though you listed every field from the lower end to the upper end of the range individually. Any precision or extensibility information you provide will be applied to each field in the range.

The optional '`.p`' gives the number of digits of precision with which the field should be considered. At present its only practical use is in fields to be summed from data files, where it is used in converting between string and number formats.

The optional '`(instruction)`' is intended for extensibility. For output fields and key fields it is a scheme command that should be run when the field is read into the system. If the associated field specification is a key for matching reference and data records, this can affect the way the records are matched. If it is a field to be written to an output file, it will affect what the user sees.

To refer to the field actually read from the source data within the scheme command, use the following sceme variable in the appropriate place in the command: '`input-field`'

Scheme commands attached to individual fields need to return a character string and have available only the current text of the field as described above[1]. In delimited output, feel free to alter the length of the return string at will. In fixed-width output, it will be your responsibility to ensure that you return a string of a useful length if you want to maintain the fixed structure.

In the following string,

[1] That's not strictly true. I discovered the other day that I can create variables in some extension commands and then use them in a subsequent command. That requires a little documentation of the order of processing so you get the values you want, but I haven't done that yet. For now, you can know that a set of field specifiers of the same type (like output fields for reference file 1 or sum fields from the data file) are processed in order.

```
'1-10,15-20.2,30(HiMom input-field),40-'
```

The fist field runs from position 1 to 10 in the file. The second from 15 to 20, with a stated (but possibly meaningless) precision of 2. The third is the single byte at position 30, to be adjusted by replacing it with whatever the instruction `HiMom` does to it. The fourth field starts at position 40 and extends to the end of the record.

In a delimited example, the following two sets of options are equivalent.

```
-D ',' -o 4,5,6,7,8
-D ',' -o 4-8
```

In both cases the fourth, fifth, sixth, seventh, and eighth comma-delimited fields are each treated as an individual field for processing. Note that if you also provide a field order (with the '`-O`' option), that order would refer to these fields as 1, 2, 3, 4, and 5 because of the order in which you specified them, not caring about the order in the original file.

2.7 Emulation

Because of its simularity with `join`, `combine` has an emulation mode that allows you to use the syntax of the `join` command to start `combine`.

The emulation can be started by using '`--emulate join`' option as the fiest option after the command name. After that, you can use the same options you get with join.

For details of the `join` command, see See section "Join Invocation" in *GNU Coreutils Manual*. When tested against the test cases packaged with GNU Coreutils, results are identical to join with a couple of exceptions:

- The sort order can be different. `combine` produces records in the order of the second input file, wich any unmatched records from the first input file following in an arbitrary order.
- I change the arguments to the '`-o`' option in the test script to have quotes around the field order list when it is separated by spaces. `combine` can handle the space-delimited list, but the standard argument handler `getopt_long` does not interpret them as a single argument. I don't see the need to overcome that.
- There is not a specific test, but I have not yet implemented case-insensitive matching. It would have failed if tested.
- One more option that is not implemented in the emulation are the '`-j1`' and '`-j2`' methods of specifying separate keys for the two files. Use '`-1`' and '`-2`' for that. The two obsolete options would be interpreted as speciying field 1 or field 2 as the common join key, respectively.

There are also a number of features of `combine` that come through in the emulation. The main features relate to the keys: the sort order of the records in relation to the keys does not matter to `combine`, also `combine` allows you to specify a list of key fields (comma-delimited) rather than just one as arguments to '`-1`', '`-2`', and '`-j`'. You should make sure that the number of key fields is the same.

Another feature is that the second input file can actually be as many files as you want. That way you can avoid putting the records from several files together if not otherwise necessary.

3 How combine Processes Files

The base of combine reads records from a data file (or a series of them in a row) and if there is an output request for data records, it writes the requested fields out to a file or to stdout. Here is an example of this most simple version of events.

```
combine --write-output --output-fields=1-
```

This is essentially an expensive pipe. It reads from stdin and writes the entire record back to stdout.

Introducing a reference file gives more options. Now combine reads the reference file into memory before reading the data file. For every data record, combine then checks to see if it has a match. The following example limits the simple pipe above by restricting the output to those records from stdin that share the first 10 bytes in common with a record in the reference file.

```
combine -w -o 1- -r reference_file.txt --key-fields=1-10 \
            --data-key-fields=1-10 --unique
```

Note that the option '--unique' is used here to prevent more than one copy of a key from being stored by combine. Without it, duplicate keys in the reference file, when matched, would result in more than one copy of the matching data record.

The other option with a reference file is to have output based on the records in that file, with indicators of how the data file records were able to match to them. In the next example, the same match as above is done, but this time we write out a record for every unique key, with a flag set to '1' if it was matched by a data record or '0' otherwise. It still reads the data records from stdin and writes the output records to stdout.

```
combine -r -f reference_file.txt -k 1-10 -m 1-10 -u -w -o 1-10
```

Of course, you might want both sets of output at the same time: the list of data records that matched the keys in the reference file and a list of keys in the reference file with an indication of which ones were matched. In the prior two examples the two different kinds of output were written to stdout. You can do that still if you like, and then do a little post-processing to determine where the data-based records leave off and the reference-based records begins. A simpler way, however, is to let combine write the information to separate files.

In the following example we combine the output specifications from the prior two examples and give them each a filename. Note that the first one has a spelled-out '--output-file' while the second one uses the shorter 1-letter option '-t'.

```
combine -w -o 1- --output-file testdata.txt \
            -r -f reference_file.txt -k 1-10 -m 1-10 \
            -u -w -o 1-10 -t testflag.txt
```

4 Extending combine

If combine was built with Guile (GNU's Ubiquitous Intelligent Extensibility Language), you can do anything you want (within reason) to extend combine. This would have been done when combine was compiled and installed on your computer. In a number of places, there are built-in opportunities to call Guile with the data that is currently in process. Using these options, you can use your favorite modules or write your own functions in scheme to manipulate the data and to adjust how DafaFactory operates on it.

The most common method (in my current usage) of extending combine is to alter the values of fields from the input files before they are used for matching or for output. This is done inside the field list by adding the scheme statement after the range and precision. This is covered in the section on field specifications. See Section 2.6 [Field Specifiers], page 13, for details.

Another useful option is the ability to initialize Guile with your own program. To do this, you can use the '`--extension-init-file`' (or '`-X`') followed by the file name to ask combine to load that scheme file into Guile before any processing. In that way your functions will be available when you need them in the running of the program. It certainly beats writing something complicated on the command line.

In addition, there are Guile modules included in the distribution, which can be used in extension scripts.

4.1 Extension Options

The reamining extensibility options are called at various points in the program: when it starts, when a file is started, when a match is found, when a record is read, when a record is written, when a file is closed, and at the very end of the program. The options are listed below along with the way to get access to the relevant data.

The various non-field=specific options are as follows. They all occur as arguments to the option '`--extension`' (or '`-x`').

'`l`*`scheme-command`*'

Filter records from the current file using the scheme command provided. The scheme command must return '`#t`' (to keep processing the record) or '`#f`' (to ignore this record and move on to the next). The variables '`reference-field-n`' or '`data-field-n`' will be available to the scheme command, depending on whether the record to be filtered is from the data file or a reference file. In the variable names '`n`' represents the number of the specified output field, numbered from 1.

'`m`*`scheme-command`*'

Validate a proposed match using the scheme command provided. The scheme command must return '`#t`' (to confirm that this is a good match) or '`#f`' (to tell combine that this is not a match). The variables '`reference-field-n`' and '`data-field-n`' will be available to the scheme command from the reference and data records involved in a particular match. In the variable names '`n`' represents the number of the specified output field, numbered from 1. The extension specification affects the match between the data file and the last named reference file.

'h*scheme-command*'
: Validate a proposed match between two records in the same hierarchy using the scheme command provided. The scheme command must return '#t' (to confirm that this is a good match) or '#f' (to tell combine that this is not a match). The variables 'reference-field-n' and 'prior-reference-field-n' will be available to the scheme command from the prior and current reference records involved in a particular match. In the variable names 'n' represents the number of the specified output field, numbered from 1. The extension specification affects the match while traversing the hierarchs in the last named reference file.

'r*scheme-command*'
: Modify a record that has just been read using the scheme command provided. The scheme command must return a string, which will become the new value of the input record to be processed. The input record iteself can be referred to in the scheme command by using the variable 'input-record' in the scheme command at the right place. The records affected by this option are the records from the most recently named reference file, or from the data file if no reference file has yet been named.

As an example, consider that you may have received a file from someone who strips all the trailing spaces from the end of a record, but you need to treat it with a fixed-width record layout. Assuming that you have definied a scheme function rpad in the initialization file 'util.scm', you can use the following command to get at the field in positions 200-219, with spaces in place of the missing rest of the record.

```
combine -X util.scm -x 'r(rpad input-record 219 #\space)' \
        -o 200-219 trimmed_file.txt
```

The same syntax works with the other '--extension' options.

4.2 Guile Modules

Here we talk about Guile modules that are distributed with combine. At the moment, those are limited to date processing.

In addition, the file 'util.scm' in the distribution contains a few functions I have found handy. They are not documented here.

4.2.1 Calendar Functions

Included in the combine package are two Guile modules to work with dates from a number of calendars, both obscure and common. The basis for them is the set of calendar functions that are shipped with Emacs.

The reason that these functions deserve special notice here is that date comparisons are a common type of comparison that often cannot be made directly on a character string. For example I might have trouble knowing if "20030922" is the same date as "22 September 2003" if I compared strings; however, comparing them as dates allows me to find a match.

We can even compare between calendars, ensuring that "1 Tishri 5764" is recognized as the same date as "20030927".

The calendar module can be invoked as `(use-modules (date calendar))`. It provides functions for converting from a variety of calendars to and from and absolute date count, whose 0-day is the imaginary date 31 December 1 B.C. In the functions, the absolute date is treated as a single number, and the dates are lists of numbers in `(month day year)` format unless otherwise specified.

The calendar functions are as follow:

- calendar-julian-from-absolute
- calendar-absolute-from-julian
- calendar-absolute-from-islamic
- calendar-islamic-from-absolute
- calendar-hebrew-from-absolute
- calendar-absolute-from-hebrew
- calendar-absolute-from-chinese ‘`(cycle year month day)`’
- calendar-chinese-from-absolute ‘`(cycle year month day)`’
- calendar-absolute-from-french
- calendar-french-from-absolute
- calendar-absolute-from-coptic
- calendar-coptic-from-absolute
- calendar-absolute-from-ethiopic
- calendar-ethiopic-from-absolute
- calendar-absolute-from-iso ‘`(week day year)`’
- calendar-iso-from-absolute ‘`(week day year)`’
- calendar-absolute-from-persian
- calendar-persian-from-absolute
- calendar-mayan-long-count-from-absolute ‘`(baktun katun tun uinal kin)`’
- calendar-mayan-haab-from-absolute ‘`(day . month)`’
- calendar-mayan-tzolkin-from-absolute ‘`(day . name)`’
- calendar-445fiscal-from-absolute ‘`(month workday year)`’
- calendar-absolute-from-445fiscal ‘`(month workday year)`’

4.2.2 Calendar Reference

Here are some variables that can be used as references to get names associated with the numbers that the date conversion functions produce for months.

‘`gregorian-day-name-alist`’

An associative list giving the weekdays in the Gregorian calendar in a variety of languages. Each element of this list is a list composed of a 2-letter language code (lowercase) and a list of 7 day names.

'gregorian-month-name-alist'
: An associative list giving the months in the Gregorian calendar in a variety of languages. Each element of this list is a list composed of a 2-letter language code (lowercase) and a list of 12 month names.

'calendar-islamic-month-name-array'
: A list of the months in the Islamic calendar.

'calendar-hebrew-month-name-array-common-year'
: A list of the months in the standard Hebrew calendar.

'calendar-hebrew-month-name-array-leap-year'
: A list of the months in the leap year Hebrew calendar.

'chinese-calendar-celestial-stem'
'chinese-calendar-terrestrial-branch'
'lunar-phase-name-alist'
'solar-n-hemi-seasons-alist'
'solar-s-hemi-seasons-alist'
'french-calendar-month-name-array'
: A list of the months in the French Revolutionary calendar.

'french-calendar-multibyte-month-name-array'
: A list of the months in the French Revolutionary calendar, using multibyte codes to represent the accented characters.

'french-calendar-day-name-array'
: A list of the days in the French Revolutionary calendar.

'french-calendar-multibyte-special-days-array'
: A list of the special days (non weekdays) in the French Revolutionary calendar, using multibyte codes to represent the accented characters.

'french-calendar-special-days-array'
: A list of the special days (non weekdays) in the French Revolutionary calendar.

'coptic-calendar-month-name-array'
: A list of the months in the Coptic calendar.

'ethiopic-calendar-month-name-array'
: A list of the months in the Ethiopic calendar.

'persian-calendar-month-name-array'
: A list of the months in the Persian calendar.

'calendar-mayan-haab-month-name-array'
'calendar-mayan-tzolkin-names-array'

4.2.3 Calendar Parsing

The calendar parsing module can be invoked as `(use-modules (date parse))`.

The most useful function in the module is `parse-date`. It takes as arguments a date string and an output format. The date string is parsed as well as possible in descending order of preference for format in case of ambiguity. The function returns the date triplet (or other such representation) suggested by the format string.

The supported format strings are the words in the function names of the form `calendar-xxxx-from-absolute` that would take the place of the `xxxx`. See Section 4.2.1 [Calendar Functions], page 17, for more information.

The parsing of the date string depends on the setting of a couple of variables. Look inside the file '`parse.scm`' for details. The list *parse-date-expected-order* lists the order in which the parser should look for the year, month, and day in case of ambiguity. The list *parse-date-method-preference* give more general format preferences, such as 8-digit, delimited, or a word for the month and the expected incoming calendar.

Here are a few examples of pasing a date and putting it out in some formats:

```
guile> (use-modules (date parse))
guile> (parse-date "27 September 2003" "gregorian")
(9 27 2003)
guile> (parse-date "27 September 2003" "julian")
(9 14 2003)
```

The 13 day difference in the calendars is the reason that the Orthodox Christmas is 2 weeks after the Roman Catholic Christmas.

```
guile> (parse-date "27 September 2003" "hebrew")
(7 1 5764)
```

Note that the Hebrew date is Rosh HaShannah, the first day of the year 5764. The reason that the month is listed as 7 rather than 1 is inherited from the Emacs calendar implementation. Using the month list in *calendar-hebrew-month-name-array-common-year* or *calendar-hebrew-month-name-array-leap-year* correctly gives `"Tishri"`, but since the extra month comes mid-year, it is easier for programming to shift around a bit.

```
guile> (parse-date "27 September 2003" "islamic")
(7 30 1424)
guile> (parse-date "27 September 2003" "iso")
(39 6 2003)
```

This is the 6th day (Saturday) of week 39 of the year.

```
guile> (parse-date "27 September 2003" "mayan-long-count")
(12 19 10 11 7)
```

I won't get into the detail, but the five numbers reflect the date in the Mayan calendar as currently understood.

Generally, I'd recommend using the more specific functions if you are sure of the date format you expect. For comparing dates, I would further recommend comparing the absolute day count rather than any more formatted format.

5 Applying combine

There are lots of ways to apply `combine` to problems that will pop up in the course of day-to-day life. (Of course this fits into your day-to- day life a bit more if you have a job or hobby that regularly puts you into contact with data files.

Here are a few topics explored in more detail.

5.1 Rearranging Fields

When you do not use any reference files, `combine` still gives you the opportunity to create a new record layout based on the records you read.

This is an advantage over the `cut` command because cut allows you only to omit portions of the record, `combine` also allows you to reorder those fields you keep and to add a constant field somewhere in the order. In addition, `combine` gives you the chance to convert between fixed-width and delimited formats, where `cut` keeps the format you started with (although the GNU version does let you change delimiters).

Clearly, flexible tools like `awk` or `sed` or any programming language will also make this kind of thing (and with a little work anything `combine` can do) possible. It may be that they are a more efficient choice, but I prefer to have `combine`.

As an example, here is a fixed width file, which contains in its record layout some address book information. If I need to make a tab-delimited file of names and phone numbers to upload into my mobile phone, I can use the command that follows to get the output I want.

```
$ cat testadd.txt
2125551212Doe       John      123 Main StreetNew York  NY10001
2025551212Doe       Mary      123 Main StreetWashingtonDC20001
3015551212Doe       Larry     123 Main StreetLaurel    MD20707
6175551212Doe       Darryl    123 Main StreetBoston    MA02115
6035551212Doe       Darryl    123 Main StreetManchesterNH02020
```

Here is a command that graps the first and last name and the phone number and tacks the word "Home" on the end so that my phone marks the number with a little house.

Note that the statistics and the output all show up on the screen if you do not say otherwise. The statistics are on `stderr` and the output on `stdout`, so you can redirect them differently. You can also use the option ‘`--output-file`’ (or ‘`-t`’) to provide an output file, and you can suppress the statistics if you want with ‘`--no-statistics`’.

```
% combine --write-output --output-field-delimiter=" " \
          --output-fields=21-30,11-20,1-10 \
          --output-constant="Home" testadd.txt
Statistics for data file testadd.txt
  Number of records read:                            5
  Number of records dropped by filter:               0
  Number of records matched on key:                  5
  Number of records written:                         5
John Doe 2125551212 Home
Mary Doe 2025551212 Home
```

```
Larry Doe 3015551212 Home
Darryl Doe 6175551212 Home
Darryl Doe 6035551212 Home
```

For reference, here is a comparable SQL query that would select the same data assuming a table were set up containing the data in the file above.

```
SELECT First_Name, Last_Name, Phone_Number, 'Home'
  FROM Address_Book;
```

A comparable `awk` program would be something like this.

```
BEGIN {OFS = "\t"}
{
  print substr ($0, 21, 10), substr ($0, 11, 10), substr ($0, 1, 10), "Home";
  }
```

5.2 Aggregation

One feature of `combine` is aggregation. In other words, based on a set of keys in a data file, create a new file containing a summary record for each unique combination of values that appears in those keys.

This is a special case of reference-file based output in that the reference file and the data file are the same file. We are using it both to build the list of keys and to provide the values to summarize.

Here is a simple example. Suppose my 4-year-old were flipping Sacajawea dollar coins and recording the results, along with the time and the date. A portion of the resulting file might look like this:

```
Monday 27 Oct 2003 14:02:01 Head 1.00
Monday 27 Oct 2003 14:03:05 Head 1.00
Monday 27 Oct 2003 14:03:55 Tail 1.00
Monday 27 Oct 2003 14:04:30 Head 1.00
Monday 27 Oct 2003 14:06:12 Tail 1.00
Monday 27 Oct 2003 14:08:43 Head 1.00
Monday 27 Oct 2003 14:54:52 Head 1.00
Monday 27 Oct 2003 19:59:59 Tail 1.00
Tuesday 28 Oct 2003 08:02:01 Tail 5.00
Tuesday 28 Oct 2003 08:02:16 Tail 5.00
Tuesday 28 Oct 2003 08:02:31 Head 5.00
Tuesday 28 Oct 2003 08:02:46 Tail 5.00
Wednesday 29 Oct 2003 12:02:09 Head 10.00
```

Then if I wanted to measure her daily performance, I could count the number of coin flips per day, with the following command.

The option '`--data-is-reference`' tells the program to read the file only once, bulding the list of keys as it goes. This is useful for aggregations, and it requires that the key fields be the same and that we only keep one copy of each key (with the '`--unique`' option).

The records come out in an unspecified order. If you require a specific order, you can pipe the result through the `sort` command.

```
% combine --input-field-delimiter=" " --data-is-reference \
          --count --reference-file=testcoin.txt \
          --output-field-delimiter="," -D " " \
          --key-fields=2 --data-key-fields=2 --write-output \
          --output-fields=2 --unique testcoin.txt
28 Oct 2003,4
29 Oct 2003,1
27 Oct 2003,8
Statistics for reference file testcoin.txt
  Number of records read:                          13
  Number of records dropped by filter:              0
  Number of records stored:                         3
  Number of records matched on key:                 3
  Number of records matched fully:                  3
  Number of reference-data matches:                13
```

An equivalent SQL statement, assuming that the table has been created would be.

```
SELECT Date, COUNT (Result)
  FROM Coinflip
  GROUP BY Date;
```

If she wanted to count the number of heads and tails, and also the amount she bet on each, she could do the following. (I have shortened the options you have already seen to their 1-byte equivalents.)

The specification of the field to sum says to keep 2 decimal places. `combine` works with integers if you don't say otherwise. The option '`--counter-size`' tells the program to use at least that many bytes to present counters and sums.

```
% combine -D " " -M -n --sum-fields=5.2 --counter-size=8 \
          -r testcoin.txt -d "," -D " " -k 4 -m 4 -w -o 4 \
          -u testcoin.txt
Head,       7,   20.00
Tail,       6,   18.00
Statistics for reference file testcoin.txt
  Number of records read:                          13
  Number of records dropped by filter:              0
  Number of records stored:                         2
  Number of records matched on key:                 2
  Number of records matched fully:                  2
  Number of reference-data matches:                13
```

An equivalent SQL statement, assuming again that the table has been created would be.

```
SELECT Result, COUNT (Result), Sum (Wager)
  FROM Coinflip
  GROUP BY Result;
```

5.3 Finding Changes

If you get the same file again and again, and you want to see what's different, the canonical method is to use the diff command. It will list for you those records that are unique to each file, and those that appear to have changed. One drawback is that it requires the records in the two files to be in the same order. Another is that it does not pay attention to the key fields that may be important to you.

When I want to find the changes in a file, including additions, deletions, and any changes for the information about a key I normally use 4 applications of combine. One command is suffucuent to find the new keys. One is enough to find those that are no longer there. A third finds everything that is different, and the fourth finds the before and after image of keys whose information has changed.

As an example, suppose my bridge club's roster has a name and a phone number, separated by commas, for each member. The following four commands will identify the new members, the departed members, and those whose phone number has changed.

```
%# Reference file is optional, so unmatched data records will have
%# 2 blank fields at the end.
% combine -D , -d , -w -o 1-2 -r old_roster.txt -D , -k 1 -m 1 -p \
          -o 1-2 inew_roster.txt | grep -v ’ , $’ > new_member.txt
%# Data file is optional, so unmatched reference records will have
%# 2 blank fields at the beginning.
% combine -D , -d , -p -w -o 1-2 -r old_roster.txt -D , -k 1 -m 1 \
          -o 1-2 inew_roster.txt | grep -v ’^ , ’ > departed_member.txt
%# Both files are optional, so any unmatched records will have
%# blank fields at the beginning or end.  Here we match on the
%#entire record rather than the key as in the above two examples.
% combine -D , -d , -p -w -o 1-2 -r old_roster.txt -D , \
          -k 1-2 -m 1-2 -o 1-2 -p inew_roster.txt | grep -v ’ , ’ \
          > all_changes.txt
%# Match up the before and after versions of an entry to see changes.
% combine -D , -d , -w -o 1-2 -r all_changes.txt -D , -k 3 -m 1 -o 2 \
          all_changes.txt | grep -v ’ , ’ > changed_member.txt
```

TO BE CONTINUED

5.4 Hierarchies

A hierarchy tends to be an organization where there is a directional one-to-many relationship between parent records and their associated children. combine works with hierarchies within reference files when the file has a record for each node and each record points to its parent in the hierarchy.

Because the hierarchy is assumed to be stored in a reference file, it is accessed by matching to a data record. Once an individual reference record has been matched to the data record, its relationship to other records within the hierarchy is followed through the hierarchy until there is no further to go.

The standard process is to assume that the key that matched to the data file key is at the top of the hierarchy. When traversing the hierarchy, `combine` looks for the key on the current record in the hierarchy key of other reference records. This repeats until there are no further linkages from one record to the next. For each record that is linked to the hierarchy, that record is treated as a reference record that matched the data record.

In this section, we'll use the following hierarchy file. It is a simple hierarchy tree with '`Grandfather`' as the top node and 2 levels of entries below.

```
Grandfather,
Father,Grandfather
Uncle,Grandfather
Me,Father
Brother,Father
Cousin,Uncle
```

If my data file consisted only of a record with the key '`Grandfather`', then the following command would result in the records listed after it. Each record written includes the entry itself and its parent.

```
combine -D ',' -w -d ',' -r test1.tmp -k 1 -m 1 -a 2 -D ',' \
        -o 1-2 test2.tmp

Grandfather,
Father,Grandfather
Me,Father
Brother,Father
Uncle,Grandfather
Cousin,Uncle
```

If we are only interested in the endpoints (in this case all the lowest-level descendants of '`Grandfather`'), we can use the option '`-l`'.

```
combine -D ',' -w -d ',' -r test1.tmp -k 1 -m 1 -a 2 -D ',' -o 1 \
        -l test2.tmp

Me
Brother
Cousin
```

We can arrive at the same number of records, each containing the entire hierarchy traversed to get to the leaf nodes, by using the option '`--flatten-hierarchy`' ('`-F`'). This option takes a number as an argument, and then includes information from that many records found in traversing the hierarchy, starting from the record that matched the data record. This example tells `combine` to report three levels from the matching '`Grandfather`' record.

```
combine -D ',' -w -d ',' -r test1.tmp -k 1 -m 1 -a 2 -D ',' -o 1 \
        -F 3 test2.tmp

Grandfather,Father,Me
Grandfather,Father,Brother
Grandfather,Uncle,Cousin
```

As with other areas within `combine`, the hierarchy manipulation is extensible through Guile. The key fields can be modified as with any other fields. See Section 2.6 [Field Specifiers], page 13, for details. The matches within the hierarchy can be further filtered, using the '`h`' suboption of the option '`-x`'. (see Chapter 4 [Extending combine], page 16.) As with matches between reference records and data this filtering can allow you to perform fuzzy comparisons, to do more complex calculations to filter the match, or to decide when you have gone far enough and would like to stop traversing the hierarchy.

6 Future Work

Here is a short list of things that we think will be useful in the near future:

Indexed files. Add the ability to work with a reference file through an index rather than loading the whole thing into a hash table. Open questions are the format of the index and whether to put the entire index into memory, or work with it from disk.

Index

C

calendar functions . . . 17
calendar parsing . . . 19
calendar reference . . . 18

D

data file . . . 7

E

extension options . . . 16
extensions, calendar . . . 17

F

file options . . . 6
file structure options . . . 6
file, data . . . 7
file, reference . . . 8

G

general options . . . 5
guile modules . . . 17

M

modules, calendar . . . 17
modules, guile . . . 17

O

options, data file . . . 7
options, extension . . . 16
options, file . . . 6
options, file structure . . . 6
options, general . . . 5
options, reference file . . . 8

P

parsing, calendar . . . 19

R

reference file . . . 8
reference strings, calendar . . . 18

Table of Contents

combine Manual 1

1 Overview of combine 2
- 1.1 Introduction 2
  - 1.1.1 Ways to Use combine 2
    - 1.1.1.1 Encrypted Data 2
    - 1.1.1.2 Key Validation 3
- 1.2 Reporting Bugs 3

2 Invoking combine 5
- 2.1 General Options 5
- 2.2 File Options 5
- 2.3 Data Files 7
- 2.4 Reference Files 8
- 2.5 Output Files 10
- 2.6 Field Specifiers 13
- 2.7 Emulation 14

3 How combine Processes Files 15

4 Extending combine 16
- 4.1 Extension Options 16
- 4.2 Guile Modules 17
  - 4.2.1 Calendar Functions 17
  - 4.2.2 Calendar Reference 18
  - 4.2.3 Calendar Parsing 19

5 Applying combine 21
- 5.1 Rearranging Fields 21
- 5.2 Aggregation 22
- 5.3 Finding Changes 23
- 5.4 Hierarchies 24

6 Future Work 27

Index 28