Generic ClearSilver and GLib Module Build Support

Thomas J. Moore

Version 1.0 Revision 196
2010

Abstract

This document provides generic ClearSilver[1] and GLib[2] support for my generic build system. It also provides some useful generic template extensions.

This document was generated from the following sources, all of which are attached to the original electronic forms of this document:

```
$Id: build.nw 197 2012-12-06 05:02:53Z darktjm $
$Id: cs-glib.nw 196 2012-12-06 05:02:09Z darktjm $
```

Contents

1 Memory Allocation

There are a number of dynamic array facilities and memory allocation techniques that I tend to reimplement with every project. However, the GLib library provides many of the facilities I most commonly use, like dynamic arrays, dynamic strings, linked lists of buffers, and linked list of buffers with hash table duplication removal. Although some implementations are not as efficient as they could be, using them reduces the overall size of code that needs to be maintained. Support is provided here for that library — in particular, the ones provided with Red Hat Enterprise Linux 4 (2.4.7) and Red Hat Enterprise Linux 5 (2.12.3).

[1] http://www.clearsilver.net/; version 0.10.5+ is required. The 0.10.3 and 0.10.4 versions from EPEL are buggy and missing the automatic escape mechanisms.
[2] http://www.gtk.org/

1 ⟨*(build.nw) Common C Includes* 1⟩≡ (46a) 2d▷

```
#include <glib.h>
```

2a ⟨*(build.nw) makefile.vars* 2a⟩≡ 2b▷

```
GLIB_CFLAGS:=$(shell pkg-config --silence-errors gthread-2.0 --cflags || \
                     pkg-config glib-2.0 "--cflags")
GLIB_LDFLAGS:=$(shell pkg-config --silence-errors gthread-2.0 --libs || \
                      pkg-config glib-2.0 --libs)
```

2b ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁2a 7c▷

```
EXTRA_CFLAGS += $(GLIB_CFLAGS)
EXTRA_LDFLAGS += $(GLIB_LDFLAGS)
```

One thing that is missing that I have always found useful is a version of `fgets`(3) which supports dynamic strings. Such a function is easy enough to implement, so here it is. A generic version is provided as well for cases where `fgets`-equivalent functions are available.

When reading binary data that may contain NUL characters, the buffer length must be updated based not on `strlen`, but on the number of characters read. This can be obtained from `ftell` only for seekable streams, so the only option with `fgets` is to fill the memory with non-NUL characters and search for the last added NUL. Another option would be to use a different reader function. The CPU overhead for using `getc` for every character is pretty high, though, and `fscanf` has issues with NUL characters as well. Implementing a wrapper around `fread` would require that all reads from the file be done using this function in order to take care of backlog.

In order to avoid the processing overhead for NULs for most reads, the wrapper has two names, and the generic function takes a parameter to enable this.

2c ⟨*Library* `glib-supt` *Members* 2c⟩≡

```
g_string_fgets.o
```

2d ⟨*(build.nw) Common C Includes* 1⟩+≡ (46a) ◁1 4c▷

```
#include "g_string_fgets.h"
```

2e ⟨*g_string_fgets.h* 2e⟩≡

```
/*
   ⟨(build.nw) Common NoWeb Warning 3c⟩
*/
#ifndef _G_STRING_FGETS_H
#define _G_STRING_FGETS_H
⟨gets_fp definition 2f⟩
#endif /* _G_STRING_FGETS_H */
```

2f ⟨`gets_fp` *definition* 2f⟩≡ (2e)

```
typedef char *(*gets_fp)(char *buf, int len, void *f);
```

2g ⟨*(build.nw) Known Data Types* 2g⟩≡ 5c▷

```
gets_fp,%
```

2h ⟨*g_string_fgets.c* 2h⟩≡

⟨*(build.nw) Common C Header* 16c⟩

```
char *g_string_fgets(GString **_buf, guint offset, FILE *f)
{
  return g_string_gets_generic(_buf, offset, (gets_fp)fgets, f, FALSE);
}

char *g_string_fgets_bin(GString **_buf, guint offset, FILE *f)
{
  return g_string_gets_generic(_buf, offset, (gets_fp)fgets, f, TRUE);
}

char *g_string_gets_generic(GString **_buf, guint offset, gets_fp my_gets,
                            void *f, gboolean allow_zero)
{
  guint left, len;
  GString *buf = *_buf;
  gboolean read_some = FALSE;
  const char *ep;

  if(!buf)
    buf = *_buf = g_string_sized_new(offset + 80);
  else if(buf->allocated_len <= offset)
    g_string_set_size(buf, offset + 80);
  while(1) {
    left = buf->allocated_len - offset;
    if(allow_zero)
      memset(buf->str + offset, 255, left);
    else /* still need to detect end of line */
      buf->str[offset + left - 1] = 255;
    if(!(*my_gets)(buf->str + offset, left, f)) {
      buf->str[offset] = 0;
      break;
    }
    read_some = TRUE;
    if(allow_zero) {
      /* memrchr is GNU extension, so this is done manually */
      /* guaranteed to terminate */
      for(ep = buf->str + buf->allocated_len - 1; *ep; ep--);
      len = (int)(ep - buf->str) - offset;
    } else
      len = strlen(buf->str + offset);
    left -= len;
    offset += len;
    if((left > 1 && buf->str[offset + left - 1]) ||
       buf->str[offset - 1] == '\n')
      break;
    g_string_set_size(buf, buf->allocated_len * 2 - 1);
  }
  buf->len = offset;
  return read_some ? buf->str : NULL;
}
```

3a ⟨*(build.nw) Version Strings* 3a⟩≡

```
"$Id: cs-glib.nw 196 2012-12-06 05:02:09Z darktjm $\n"
```

3b ⟨*(build.nw) Sources* 3b⟩≡

```
$Id: cs-glib.nw 196 2012-12-06 05:02:09Z darktjm $
```

3c ⟨*(build.nw) Common NoWeb Warning* 3c⟩≡ (2e 47b)

```
# $Id: cs-glib.nw 196 2012-12-06 05:02:09Z darktjm $
```

Its main disadvantage for server-like processes is that it aborts the entire program on memory failures rather than returning an error so that the caller can recover gracefully. If this is an issue, a process monitor, such as `init`(8), can take care of this problem. Another memory allocation issue is that GLib uses 32-bit integers in some places, and does not check for integer overflow, so some errors are never caught except by segmentation fault[3]. This can be avoided by running these processes with "`ulimit -v <size>`" in effect on Linux, or "`ulimit -d <size>`" in effect on AIX, where <*size*> is at most a small (≤ 3) multiple of 1048576 (1GB). The following script can be used to support starting processes in `init`(8). It checks to see if a file in `/tmp` has been touched before continuing, so that the file can be touched before killing a process for maintentance.

4a ⟨*runit* 4a⟩≡

```
#!/bin/sh

prog_to_run="$1"
type "$prog_to_run" >/dev/null 2>&1 || exit 1
shift
prog_to_run_base="${prog_to_run##*/}"
while [ -f "/tmp/stop_${prog_to_run_base}" ]; do
  sleep 60
done

# Use -v on Linux, -d on AIX
ulimit -v $((2*1024*1024)) 2>/dev/null || ulimit -d $((2*1024*1024))
if type gawk >/dev/null 2>&1; then
  exec "$prog_to_run" "$@" 2>&1 | \
        gawk '{ print strftime("%Y-%m-%d %H:%M:%S"), $0; fflush(); }'
else
  exec "$prog_to_run" "$@"
fi
```

4b ⟨*(build.nw) Script Executables* 4b⟩≡

```
runit \
```

2 Logging

One way to deal with logs is to not deal with them, and instead just dump errors to standard error. While a simple `perror`(3) would be enough (with some extra text from application libraries), the slightly more complex ClearSilver[4] error facility could be used instead. This provides a partial traceback with little extra effort. As with any method that passes around error messages rather than error codes, though, it will be hard to translate things into a different language. It will also be hard to distinguish between different errors unless separate error codes are used.

[3] Beyond this, there are some problems in the way UNIX handles too-large requests for memory. UNIX will allocate from swap space, and even (at least on Linux) over-allocate beyond available memory without returning errors. Using `ulimit -v/-d` will cure this as well.

[4] `http://www.clearsilver.net/`; at least version 0.10.5 is required. Older versions are buggy and missing the automatic escape mechanisms.

4c ⟨*(build.nw) Common C Includes* 1⟩+≡ (46a) ◁2d 22c▷

```
#include <time.h> /* for -Wshadow */
#define timezone time_zone /* for -Wshadow */
#include <ClearSilver.h>
#include "cs_supt.h"
```

5a ⟨*cs_supt.h* 5a⟩≡

```
#ifndef _CS_SUPT_H
#define _CS_SUPT_H
⟨ClearSilver Support Globals 5b⟩
#endif /* _CS_SUPT_H */
```

5b ⟨*ClearSilver Support Globals* 5b⟩≡ (5a) 5h▷

```
extern NERR_TYPE GENERIC_ERR;
#ifdef __GNUC__
#define unused_attr __attribute__((__unused__))
#else
#define unused_attr
#endif
```

5c ⟨*(build.nw) Known Data Types* 2g⟩+≡ ◁2g 11e▷

```
unused_attr,%
```

5d ⟨*Library* `cs-supt` *Members* 5d⟩≡ 17a▷

```
cs_supt.o
```

5e ⟨*cs_supt.c* 5e⟩≡ 7f▷

```
⟨(build.nw) Common C Header 16c⟩

NERR_TYPE GENERIC_ERR = 0;
```

5f ⟨*Common Mainline Variables* 5f⟩≡ (46a)

```
NEOERR *nerr unused_attr = STATUS_OK;
int ret unused_attr = 0;
```

5g ⟨*Initialize logging* 5g⟩≡ (8g)

```
{
  const char *progname = strrchr(argv[0], '/');

  if(!progname)
    progname = argv[0];
  else
    progname++;
  g_set_prgname(progname);
}
nerr = nerr_init();
die_if_err(nerr); /* out of memory - pretty fatal */
nerr_register(&GENERIC_ERR, g_get_prgname());
```

5h ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁5b 6c▷

```
#define die_if_err(err) do { \
  NEOERR *_err = err; \
 \
  if(_err != STATUS_OK) { \
    nerr_log_error(_err); \
    exit(1); \
  } \
} while(0)

#define nerr_raise_msg(msg) \
  nerr_raise(GENERIC_ERR, "%s", msg)

#define die_msg(msg) \
  die_if_err(nerr_raise_msg(msg))

#define nerr_raise_msg_errno(msg) \
  nerr_raise_errno(GENERIC_ERR, "%s", msg)

#define die_errno(msg) \
  die_if_err(nerr_raise_msg_errno(msg))

#define nerr_raise_errno_local(msg, err) \
  nerr_raise(GENERIC_ERR, "%s: [%d] %s", msg, err, strerror(err))

#define die_errno_local(msg, err) \
  die_if_err(nerr_raise_errno_local(msg, err))

#define nerr_op(op) do { \
  if(nerr == STATUS_OK) \
    nerr = nerr_pass(op); \
} while(0)

#define nerr_op_ok(op) (nerr == STATUS_OK && \
                        (nerr = nerr_pass(op)) == STATUS_OK)
```

6a ⟨*(build.nw) C Prototypes* 6a⟩≡ 7a▷

```
void die_if_err(NEOERR *err);
NEOERR *nerr_raise_msg(const char *msg);
void die_msg(const char *msg);
NEOERR *nerr_raise_msg_errno(const char *msg);
void die_errno(const char *msg);
NEOERR *nerr_raise_errno_local(const char *msg, int err);
void die_errno_local(const char *msg, int err);
void nerr_op(NEOERR *op);
gboolean nerr_op_ok(NEOERR *op);
```

6b ⟨*Convert* `nerr` *to* `err_str` 6b⟩≡

```
STRING err_str;

string_init(&err_str);
nerr_error_traceback(nerr, &err_str);
nerr_ignore(&nerr);
```

A few GLib functions actually do return error codes (`GError *`), in which case a macro can be used to convert that into a ClearSilver error.

6c ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁5h 7d▷

```
#define gerr_to_nerr(gerr) \
  ((gerr) ? \
```

6c ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁5h 7d▷

```
    nerr_raise(GENERIC_ERR, "%s/%d: %s", \
               g_quark_to_string((gerr)->domain), (gerr)->code, \
                                 (gerr)->message) : \
    STATUS_OK)
```

7a ⟨*(build.nw) C Prototypes* 6a⟩+≡ ◁6a 7e▷

```
NEOERR *gerr_to_nerr(const GError *gerr);
```

Actually building with ClearSilver requires knowledge of the library location. This cannot be obtained by pkg-config, so it is an overridable make variable.

7b ⟨*(build.nw) makefile.config* 7b⟩≡ 8b▷

```
# Where to find ClearSilver
CLEARSILVER_CFLAGS=-I/usr/include/ClearSilver -I/usr/local/include/ClearSilver
CLEARSILVER_LDFLAGS=-L/usr/local/lib -lneo_cgi -lneo_cs -lneo_utl -lz
```

7c ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁2b 8c▷

```
EXTRA_CFLAGS += $(CLEARSILVER_CFLAGS)
EXTRA_LDFLAGS += $(CLEARSILVER_LDFLAGS)
```

3 Configuration

Since GLib doesn't support configuration files until 2.6, a different library is used. The ClearSilver library supports configuration files. It is also somewhat complex and poorly documented, but it is less complex and better documented than some, and provides easy ways to override variables and write out configuration files.

Configuration parameters for a program are always in a section of the configuration file named after the program. This allows a combined configuration file for multiple programs, and since the entire file can be prefixed with the program name fairly easily, it is no great burden on the user. If no configuration file is found, or an option has no entry in the configuration file, a hard-coded default is used.

7d ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁6c 8f▷

```
extern HDF *local_config;

#define getconf(name, def) hdf_get_value(local_config, name, def)
#define getconf_int(name, def) hdf_get_int64_value(local_config, name, def)
```

7e ⟨*(build.nw) C Prototypes* 6a⟩+≡ ◁7a 9b▷

```
const char *getconf(const char *name, const char *def);
gint64 getconf_int(const char *name, gint64 def);
```

7f ⟨*cs_supt.c* 5e⟩+≡ ◁5e 8a▷

```
HDF *local_config;

gint64 hdf_get_int64_value(HDF *hdf, const char *name, gint64 def)
{
  const char *val = hdf_get_value(hdf, name, NULL);
  char *s;
  gint64 ret;
```

7f ⟨*cs_supt.c* 5e⟩+≡ ◁5e 8a▷

```
  if(val && *val) {
    ret = strtoll(val, &s, 0);
    if(*s)
      die_if_err(nerr_raise(GENERIC_ERR, "Expected integer for %s; got %s", name, val));
    return ret;
  } else
    return def;
}
```

8a ⟨*cs_supt.c* 5e⟩+≡ ◁7f 11f▷

```
NEOERR *hdf_set_int64_value(HDF *hdf, const char *name, gint64 val)
{
  char nbuf[22]; /* 20 + sign */

  snprintf(nbuf, sizeof(nbuf), "%lld", (long long)val);
  return hdf_set_value(hdf, name, nbuf);
}
```

The configuration file's name and location are application-dependent. These support routines just provide a simple way to read the known name. Common environment variables are provided to override the name: `CONFIG_ROOT` just overrides the directory, and `CONFIG_FILE` overrides either the file name or the entire path. The default configuration path is determined by the project name, although the default need not be used.

8b ⟨*(build.nw) makefile.config* 7b⟩+≡ ◁7b 20c▷

```
# Installation directory for configuration files
ETC_DIR=/etc/$(PROJECT_NAME)
```

8c ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁7c 10a▷

```
EXTRA_CFLAGS += -DETC_DIR='"$(ETC_DIR)"' -DPROJECT_NAME='"$(PROJECT_NAME)"'
```

8d ⟨*Default* `config_root` 8d⟩≡

```
const char *config_root = ETC_DIR;
```

8e ⟨*Default* `config_path` 8e⟩≡

```
const char *config_path = PROJECT_NAME ".conf";
```

8f ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁7d 9a▷

```
/* define in mainline */
extern const char *config_root, *config_path;
```

8g ⟨*Read configuration* 8g⟩≡ (46a) 13a▷

```
⟨Initialize logging 5g⟩
env_override("CONFIG_ROOT", config_root);
env_override("CONFIG_FILE", config_path);
prepend_dir_if_relative(config_root, config_path);
{
  HDF *config_file;

  nerr = nerr_pass(hdf_init(&config_file));
  die_if_err(nerr); /* out of memory - pretty fatal */
```

8g ⟨*Read configuration* 8g⟩≡ (46a) 13a▷

```
  {
    char *p = strdup(config_path), *s;

    if(!p)
      die_errno("initial setup");
    s = strrchr(p, '/');
    if(s) {
      *s = 0;
      s = p;
    } else
      s = (char *)"."; /* not modified any more */
    nerr = nerr_pass(hdf_set_value(config_file, "hdf.loadpaths.conf", s));
    free(p);
    die_if_err(nerr);
  }
  nerr = nerr_pass(hdf_read_file(config_file, config_path));
  if(nerr != STATUS_OK) {
    /* ignore file not found -- no config file means use default */
    if(!nerr_match(nerr, NERR_NOT_FOUND))
      die_if_err(nerr);
    nerr_ignore(&nerr);
  }
  nerr_op(hdf_get_node(config_file, g_get_prgname(), &local_config));
  if(nerr == STATUS_OK) {
    HDF *lp = hdf_get_obj(config_file, "hdf.loadpaths");

    if(lp)
      nerr_op(hdf_copy(local_config, "hdf.loadpaths", lp));
  }
  die_if_err(nerr); /* out of memory */
}
```

9a ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁8f 11d▷

```
#define env_override(env, var) do { \
  const char *s = getenv((char *)env); \
 \
  if(s) \
    var = s; \
} while(0)

#define prepend_dir_if_relative(def_dir, fn) do { \
  if((fn)[0] != '/') { \
    char *s; \
 \
    if(!(s = malloc(strlen(def_dir) + strlen(fn) + 2))) \
      die_errno(fn); \
    sprintf(s, "%s/%s", def_dir, fn); \
    fn = s; \
  } \
} while(0)
```

9b ⟨*(build.nw) C Prototypes* 6a⟩+≡ ◁7e 12b▷

```
void env_override(const char *env, const char *&var);
void prepend_dir_if_relative(const char *def_dir, const char *&fn);
```

A sample commented configuration file should be provided. The ClearSilver documentation does not have a complete specification of the configuration file syntax, so instead of pointing a user at a URL, the relevant parts of the syntax are given as documentation at the top of the sample configuration file.

9c ⟨*ClearSilver Configuration Documentation* 9c⟩≡

```
# The purpose of this file is to assign values to keys.  The last assignment
# to any key overrides all previous assignments.  This file may provide
# configuration for multiple programs; each program's configuration is
# prefixed by its executable name.  Use links to allow multiple executable
# names to have the same configuration.
# File format:
#  Lines starting with # are comments, unless they are #include
#
#  #include <name> includes another file (<name> can have quotes around it)
#  If you want to use the include path, and want to use #include anywhere but
#  at the top level, add the following line in the same section as the
#  #include:
#    hdf : hdf
#
#  Keys are multiple level names, separated by dots:
#    Key level names are case-sensitive alphanumeric and underscores
#    A key level, followed by an open curly brace, starts a group
#      In the group, all entries are prefixed by the level before the brace
#      For example:
#        x {
#          y ...
#          z ...
#        }
#       is the same as
#        x.y ...
#        x.z ...
#
#  Values can either be literal values, inline data, or other keys
#    Literal values are assigned with <key> = <value> (no escapes)
#    Inline data is assigned by <key> << <token>
#      ... (literal data, no escapes, always includes a final newline)
#      <token>
#    A copy of another key's current value is created by <key> := <other key>
#    A link to another key that tracks changes is created by <key> : <other key>
#    Numeric values are C-style decimal ([1-9][0-9]*), octal (0[0-9]*) and
#    hexadecimal (0x[0-9a-fA-F]+).
#
#  Initial and trailing whitespace is ignored on all lines except inline data
#  Whitespace surrounding =, :=, :, <<, and { is ignored
#  Whenever key levels are sorted, they are interpreted as numbers first, if
#  possible.  Numbers always come before alphanumerics.
```

In order to support the multi-line literal syntax when using `notangle`, the end of each multi-line literal must be shifted all the way to the left margin. For this build procedure, only `EOV` is supported as an end token.

10a ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁8c 14a▷

```
HDF_FILES=⟨HDF Files 11b⟩
```

10b ⟨*(build.nw) Build Source* 10b⟩≡ 15f▷

```
$(HDF_FILES) \
```

10c ⟨*(build.nw) makefile.rules* 10c⟩≡ 11a▷

```
misc: $(HDF_FILES)
$(HDF_FILES): $(NOWEB_ORDER)
        notangle -R$@ $^ | sed 's/^ *EOV/EOV/' > $@
distclean: cleanhdf
cleanhdf:
```

10c ⟨*(build.nw) makefile.rules* 10c⟩≡ 11a▷

```
        rm -f $(HDF_FILES)
```

11a ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁10c 15e▷

```
misc: $(HDF_FILES)
```

11b ⟨*HDF Files* 11b⟩≡ (10a)

```
\
```

11c ⟨*(build.nw) Install other files* 11c⟩≡

```
$(if $(HDF_FILES), \
mkdir -p $(DESTDIR)$(ETC_DIR); \
for x in $(HDF_FILES); do \
  cp -p $$x $(DESTDIR)$(ETC_DIR)/$${x}.sample; \
  test -f $(DESTDIR)$(ETC_DIR)/$${x} || cp -p $$x $(DESTDIR)$(ETC_DIR); \
done)
```

The sorting mentioned in the commentary above is implemented by a comparison function for `hdf_sort_obj`.

11d ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁9a 12a▷

```
/* These type names are too long, so here's shorter versions */
typedef long long llong;
typedef unsigned long long ullong;
```

11e ⟨*(build.nw) Known Data Types* 2g⟩+≡ ◁5c 15b▷

```
ullong,llong,%
```

11f ⟨*cs_supt.c* 5e⟩+≡ ◁8a 12c▷

```
int comp_hdf_name(const void *_a, const void *_b)
{
  HDF * const *a = _a, * const *b = _b;
  const char *na = hdf_obj_name(*a), *nb = hdf_obj_name(*b);
  char *ea, *eb;
  ullong la, lb;
  int c;

  /* empty should never happen, but comes first */
  if(!*na && !*nb)
    return 0;
  if(!*na)
    return -1;
  if(!*nb)
    return 1;
  /* next come pure numbers (C-style dec, hex, oct) */
  la = strtoll(na, &ea, 0);
  lb = strtoll(nb, &eb, 0);
  if(!*ea && *eb)
    return -1;
  if(!*eb && *ea)
    return 1;
  /* can't just return la - lb, since they're not int */
  /* also, don't want different number formats to sort randomly */
```

11f ⟨*cs_supt.c* 5e⟩+≡ ◁8a 12c▷

```
  if(!*ea) {
    if(la < lb)
      return -1;
    else if(la > lb)
      return 1;
  }
  /* then comes strcasecmp() (somewhat locale-sensitive) */
  c = strcasecmp(na, nb);
  if(c)
    return c;
  /* finally, strcmp ensures that the case-sensitive names are not */
  /* shuffled by strcasecmp */
  return strcmp(na, nb);
}
```

A few additional convenience functions for hierarchical parameters are provided, as well.

12a ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁11d 14c▷

```
#define getconf_first(name) hdf_first_child(local_config, name)
#define getconf_first_sorted(name) hdf_first_child_sort(local_config, name)
```

12b ⟨*(build.nw) C Prototypes* 6a⟩+≡ ◁9b

```
HDF *getconf_first(const char *name);
HDF *getconf_first_sorted(const char *name);
```

12c ⟨*cs_supt.c* 5e⟩+≡ ◁11f 14b▷

```
HDF *hdf_first_child(HDF *hdf, const char *name)
{
  if(!(hdf = hdf_get_obj(hdf, name)))
    return hdf;
  return hdf_obj_child(hdf);
}

HDF *hdf_first_child_sort(HDF *hdf, const char *name)
{
  if(!(hdf = hdf_get_obj(hdf, name)))
    return hdf;
  hdf_sort_obj(hdf, comp_hdf_name);
  return hdf_obj_child(hdf);
}

gint64 getconf_hier_int(const char *parent, HDF *obj, const char *name,
                        gint64 def)
{
  const char *val = hdf_get_value(obj, name, NULL);
  char *s;
  gint64 ret;

  if(val && *val) {
    ret = strtoll(val, &s, 0);
    if(*s)
      die_if_err(nerr_raise(GENERIC_ERR, "Expected integer for %s.%s.%s; got %s",
                                         parent, hdf_obj_name(obj), name, val));
    return ret;
  } else
    return def;
}
```

Command-line options of the form *var=val* can be used to override configuration options from the file and defaults, where *var* is the configuration parameter name under the local section, and *val* is the value. Additional configuration files, which, like the command-line overrides, are overrides for the command-specific options, can be given on the command-line by prefixing the file name with a colon (`:`). Standard input can be specified using a dash (`:-`). Invalid options cause the command-line help to appear.

13a ⟨*Read configuration* 8g⟩+≡ (46a) ◁8g 14d▷

```
while(nerr == STATUS_OK && argc > 1) {
  const char *s;
  char *vn;

  argc--;
  s = *++argv;
  if(*s == ':' && s[1]) {
    s++;
    if(*s == '-' && !s[1]) {
      GString *cfile = NULL;

      while(g_string_fgets(&cfile, cfile ? cfile->len : 0, stdin));
      nerr_op(hdf_read_string(local_config, cfile->str));
      g_string_free(cfile, TRUE);
    } else
      nerr_op(hdf_read_file(local_config, s));
    if(nerr != STATUS_OK) {
      nerr_log_error(nerr);
      help();
    }
    continue;
  }
  if(!isalpha(*s))
    help();
  while(isalnum(*s) || *s == '_' || *s == '.')
    s++;
  if(*s != '=')
    help();
  vn = strdup(*argv);
  vn[(int)(s - *argv)] = 0;
  nerr_op(hdf_set_value(local_config, vn, s + 1));
  free(vn);
}
if(nerr != STATUS_OK) { /* out of memory */
  nerr_log_error(nerr);
  exit(1);
}
```

The default help consists mostly of the appropriate section from the default configuration file with the keyword/value pairs uncommented. This is inserted into the C file as string literals after being extracted from the NoWeb file. The actual text to extract consists of two chunks: ⟨`<program>` *Description* (imported)⟩, which is a one-line description, and ⟨`<program>` *Configuration* (imported)⟩, which is the contents of the program-specific configuration file section.

13b ⟨*Help Function* 13b⟩≡

```
void help(void)
{
  fprintf(stderr, "%s: __PROGDESC__\n\n", g_get_prgname());
  fprintf(stderr, "Usage: %s [<keyword>=<value>|:<file> ...]\n\n"
                  ":<file> specifies parameters to be read from a file "
                    "(- = standard input),\n"
                  "        one per line; see default configuration file "
                    "for additional syntax\n\n"
                  "Supported keywords and their built-in defaults:\n\n"
```

13b ⟨*Help Function* 13b⟩≡

```
                    __OPTIONS__, g_get_prgname());
  fprintf(stderr, "\nVersion ID:\n%s", version_id);
  exit(1);
}
```

14a ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁10a 15c▷

```
C_POSTPROCESS+= | \
sed "s~__PROGDESC__~` \
  notangle -R'[[$*]] Description' $(NOWEB_ORDER) 2>/dev/null || echo`~" | \
sed -e "/__OPTIONS__/{s/__OPTIONS__//;i\\` \
  (notangle -R'[[$*]] Configuration' $(NOWEB_ORDER) 2>/dev/null || echo) | \
    sed -e '/^ *\#[^ ].* =/{s/\#//;s/ = */=/;};' \
        -e '/^ *\#[^ ].* {$$/s/\#//;s/\#}$$/}/;' \
        -e 's/"/\\\\\\\\"/g;s/%/%%/g;s/^/\"/;s/\$$/\\\\\\\n"\\\\/' \
        -e '\$${s/\\\\\$$//;s/\$$/\\n}/;}'`"
```

The first parameter is actually not intended to be documented in the default configuration file. Setting `debug` is for debugging during development only.

14b ⟨*cs_supt.c* 5e⟩+≡ ◁12c 23a▷

```
int debug;
```

14c ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁12a 15a▷

```
extern int debug;
```

14d ⟨*Read configuration* 8g⟩+≡ (46a) ◁13a

```
debug = getconf_int("debug", 0);
```

One issue with the in-memory configuration file format is that string comparisons must be done repeatedly on parameter names. There is not really anything that can be done about it, though, other than to save the results of one lookup in a local variable. Along the same lines, no place is provided to store an integral interpretation of the string value, either, except for the internally reserved field for actual integers. However, these could still be stored elsewhere. In order to help convert strings to integers, either something like a constant `GStringChunk` could be used, which would unfortunately double the string storage requirements, or, if the list of possible strings is known at compile time, a perfect hash function could be used to just convert it to an enumeration.

The following uses `gperf`[5] to convert strings to numbers. This requires a few makefile modifications to generate the code appropriately. The strings to hash are specified in chunk names like *prefix*-gperf-*suffix*. Strings are separated by newlines or single spaces; commas and spaces are not allowed in the strings. The *prefix* is the base name of the output files; the C code is generated in *prefix*.c.gperf, and the enumerations are built in *prefix*.h.gperf. These should be included directly in the C and header files that use them. Each unique *suffix* generates a different hash function and enumeration type. The enumeration type is named *suffix*`_t`, and the hash lookup function is named *suffix*`_id`. The *suffix* must be a valid C identifier. The enumeration identifiers are called *SUFFIX_STRING*, where *SUFFIX* is *suffix* converted to upper-case, and *STRING* is the string with all alphabetic characters in upper-case and all non-alphanumeric characters converted to underscores. This means that strings differing only in case or special characters cannot coexist in the same hash table. One additional enumeration literal, `UNKNOWN_`*SUFFIX*, is given the value zero, which is returned if the string is not found. The only option which may be set is case-sensitivity. A case-insensitive hash can be

[5] `http://www.gnu.org/software/gperf/`

generated by using -gperf-nc- instead of -gperf-; the nc- part is not considered to be part of the *suffix*. Note that the use of `\U` to do the upper-case conversions is a GNU sed extension.

15a ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁14c 22d▷

```
struct gperf_name_id {
  int name, id;
};
```

15b ⟨*(build.nw) Known Data Types* 2g⟩+≡ ◁11e 28a▷

```
gperf_name_id,%
```

15c ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁14a 19c▷

```
GPERF_PREFIX:=$(shell $(NOROOTS) $(NOWEB_ORDER) | tr -d '<>' | \
                        fgrep -- -gperf- | sed 's/-gperf-.*//' | sort -u)
GPERF_C := $(GPERF_PREFIX:%=%.c.gperf)
GPERF_H := $(GPERF_PREFIX:%=%.h.gperf)
GPERF_FILES := $(GPERF_H) $(GPERF_C)
```

15d ⟨*(build.nw) Clean temporary files* 15d⟩≡ 48c▷

```
rm -f $(GPERF_FILES)
```

15e ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁11a 16a▷

```
$(GPERF_C): $(NOWEB_ORDER)
        :; root=$(@:%.c.gperf=%); \
        noroots $(NOWEB_ORDER) | tr -d '<>' | grep \^$${root}-gperf- | \
          while read r; do \
            n=$${r#$${root}-gperf-}; \
            nc=; case $$n in nc-*) nc=--ignore-case; n=$${n#nc-} ;; esac ; \
            (notangle -Rgperf-prefix $(NOWEB_ORDER) ; \
             notangle -R$$r $(NOWEB_ORDER) | tr \  \\n | sort -u | \
               sed -e "h;s/^/$${n}_/;s/.*/\\U\\0/;s/[^A-Z0-9\\n]/_/g;s/^/,/;H;g" \
                   -e "s/\\n//"; \
             echo %%; \
             notangle -R'Actual gperf lookup function' $(NOWEB_ORDER)) | \
               gperf $$nc| sed "s/__GPERF_NAME__/$${n}/g"; \
          done >$@

$(GPERF_H): $(NOWEB_ORDER)
        :; root=$(@:%.h.gperf=%); \
        noroots $(NOWEB_ORDER) | tr -d '<>' | grep \^$${root}-gperf- | \
          while read r; do \
            n=$${r#$${root}-gperf-}; n=$${n#nc-}; \
            echo typedef enum \{; \
            echo UNKNOWN_`echo $$n | tr '[a-z]-' '[A-Z]_'`; \
            notangle -R$$r $(NOWEB_ORDER) | tr \  \\n | sort -u | \
              sed "s/^/$${n}_/;s/.*/\\U\\0/;s/[^A-Z0-9\\n]/_/g;s/^/,/"; \
            echo \} $${n}_t\;; \
          done >$@
```

15f ⟨*(build.nw) Build Source* 10b⟩+≡ ◁10b 19d▷

```
$(GPERF_FILES) \
```

The prototypes for the lookup functions are generated separately, because the gperf files need to be handled differently from normal C files. The fact that they are included in files that have already had their

prototypes generated does not matter, because cproto will not generate prototypes for included files. Reinserting prototypes from the separate prototype file is not possible, so ⟨*(build.nw) C Prototypes* 6a⟩ must be used, instead.

16a ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁15e 19f▷

```
gperf-proto.h: $(GPERF_FILES)
        echo '#include <string.h>' | cat - $(GPERF_FILES) | \
          cproto -E "$(CC) $(CFLAGS) $(EXTRA_CFLAGS) -x c -E" - >$@

$(COFILES): gperf-proto.h
```

16b ⟨*(build.nw) C Headers* 16b⟩≡

```
gperf-proto.h \
```

16c ⟨*(build.nw) Common C Header* 16c⟩≡ (2h 5e 17b)

```
#include "gperf-proto.h"
```

The options are sent to gperf via the header, except for the case-insensitive option sent on the command line above.

16d ⟨*gperf-prefix* 16d⟩≡

```
%pic
%struct-type
%omit-struct-type
%define hash-function-name __GPERF_NAME___hash
%define lookup-function-name __GPERF_NAME___name_id
%define string-pool-name __GPERF_NAME___strs
%compare-lengths
%readonly-tables
%enum
%define word-array-name __GPERF_NAME___words
struct gperf_name_id;
%%
```

The actual lookup function returns just the ID rather than the internal, otherwise useless structure. No equivalent function is provided for converting IDs back to strings, though. The internal lookup function is suppressed for GNU environments, because the functions are declared inline, and cproto ignores such functions.

16e ⟨*Actual gperf lookup function* 16e⟩≡

```
__GPERF_NAME___t __GPERF_NAME___id(const char *name, int len)
{
  const struct gperf_name_id *hv = __GPERF_NAME___name_id(name, len);
  return hv ? hv->id : 0; /* UNKNOWN__GPERF_NAME__ */
}
```

4 Templates

Templates can either be strings or files, and be dumped to either an open file descriptor or a string. Templates used for displaying to the user generally have a default encoding method of HTML, but templates used for configuration need to be evaluated raw. A flag allows selection of raw mode. The HDF must be provided, even though it will usually just be `local_config`. Since the output parameters are only used by one

function call, the input is processed by a generic function that takes the same callback parameters as the output function.

17a ⟨*Library* `cs-supt` *Members* 5d⟩+≡ ◁5d

```
tmpl.o
```

17b ⟨*tmpl.c* 17b⟩≡ 17c▷

```
⟨(build.nw) Common C Header 16c⟩
```

17c ⟨*tmpl.c* 17b⟩+≡ ◁17b 18a▷

```
NEOERR *tmpl_string_to(const char *tmpl, HDF *parms, CSOUTFUNC outf,
                       void *outp, gboolean raw)
{
  CSPARSE *cs = NULL;
  NEOERR *nerr = STATUS_OK;
  char *tdup;

  ⟨Set raw template parms 17d⟩
  nerr_op(prepare_template(&cs, parms));
  if(nerr != STATUS_OK) {
    ⟨Free raw template parms 17e⟩ /* memory probs, so don't try to restore */
    return nerr_pass_ctx(nerr, "%s", tmpl);
  }
  /* modified by cs_parse_string; freed by cs_destroy */
  tdup = strdup(tmpl);
  if(!tdup)
    nerr = nerr_raise_msg_errno("No memory for template");
  else
    nerr = nerr_pass(cs_parse_string(cs, tdup, strlen(tmpl)));
  nerr_op(cs_render(cs, outp, outf));
  cs_destroy(&cs);
  ⟨Restore non-raw template parms 17f⟩
  return nerr_pass_ctx(nerr, "%s", tmpl);
}
```

17d ⟨*Set raw template parms* 17d⟩≡ (17c 19a)

```
char *oescape = NULL;

if(raw) {
  nerr_op(hdf_get_copy(parms, "Config.VarEscapeMode", &oescape, NULL));
  nerr_op(hdf_set_value(parms, "Config.VarEscapeMode", "none"));
}
```

17e ⟨*Free raw template parms* 17e⟩≡ (17c 19a)

```
if(oescape)
  free(oescape);
```

17f ⟨*Restore non-raw template parms* 17f⟩≡ (17c 19a)

```
if(raw) {
  NEOERR *ignerr;

  if(oescape)
    ignerr = hdf_set_buf(parms, "Config.VarEscapeMode", oescape);
  else
    ignerr = hdf_remove_tree(parms, "Config.VarEscapeMode");
  if(ignerr != STATUS_OK)
```

17f ⟨*Restore non-raw template parms* 17f⟩≡ (17c 19a)

```
    nerr_log_error(ignerr);
}
```

18a ⟨*tmpl.c* 17b⟩+≡ ◁17c 18b▷

```
NEOERR *prepare_template(CSPARSE **_cs, HDF *hdf)
{
  NEOERR *nerr = nerr_pass(cs_init(_cs, hdf));
  CSPARSE *cs = *_cs;

  ⟨Initialize templates 22a⟩
  return nerr;
}
```

18b ⟨*tmpl.c* 17b⟩+≡ ◁18a 18c▷

```
NEOERR *cs_to_gstring(void *user, char *str)
{
  GString *out = user;

  g_string_append(out, str);
  return STATUS_OK;
}

NEOERR *tmpl_string(const char *tmpl, HDF *parms, GString *out, gboolean raw)
{
  return nerr_pass(tmpl_string_to(tmpl, parms, cs_to_gstring, out, raw));
}
```

18c ⟨*tmpl.c* 17b⟩+≡ ◁18b 18e▷

```
NEOERR *cs_to_file(void *user, char *str)
{
  FILE *f = user;

  return fputs(str, f) < 0 ? nerr_raise_msg_errno("Writing") : STATUS_OK;
}

NEOERR *tmpl_string_to_file(const char *tmpl, HDF *parms, FILE *out,
                            gboolean raw)
{
  return nerr_pass(tmpl_string_to(tmpl, parms, cs_to_file, out, raw));
}
```

Template files are loaded from the default directory if relative.

18d ⟨*Template Parameter Configuration* 18d⟩≡

```
# Default directory for templates (relative to $CONFIG_ROOT)
#tmpl_dir = tmpl
```

18e ⟨*tmpl.c* 17b⟩+≡ ◁18c 19a▷

```
const char *tmpl_dir = "tmpl";
```

18f ⟨*Set up templates* 18f⟩≡

```
{
  extern const char *tmpl_dir;
```

18f ⟨*Set up templates* 18f⟩≡

```
  tmpl_dir = getconf("tmpl_dir", "tmpl");
  prepend_dir_if_relative(config_root, tmpl_dir);
  die_if_err(hdf_set_value(local_config, "hdf.loadpaths.tmpl_dir", tmpl_dir));
}
```

19a ⟨*tmpl.c* 17b⟩+≡ ◁18e 19b▷

```
NEOERR *tmpl_file_to(const char *tmpl, HDF *parms, CSOUTFUNC outf,
                     void *outp, gboolean raw)
{
  CSPARSE *cs = NULL;
  NEOERR *nerr = STATUS_OK;

  ⟨Set raw template parms 17d⟩
  nerr_op(prepare_template(&cs, parms));
  if (nerr != STATUS_OK) {
    ⟨Free raw template parms 17e⟩ /* memory probs, so don't try to restore */
    return nerr_pass_ctx(nerr, "%s", tmpl);
  }
  nerr_op(cs_parse_file(cs, (char *)tmpl));
  nerr_op(cs_render(cs, outp, outf));
  cs_destroy(&cs);
  ⟨Restore non-raw template parms 17f⟩
  return nerr_pass_ctx(nerr, "%s", tmpl);
}
```

19b ⟨*tmpl.c* 17b⟩+≡ ◁19a 22b▷

```
NEOERR *tmpl_file(const char *tmpl, HDF *parms, FILE *out, gboolean raw)
{
  return nerr_pass(tmpl_file_to(tmpl, parms, cs_to_file, out, raw));
}

NEOERR *tmpl_file_to_string(const char *tmpl, HDF *parms, GString *out,
                            gboolean raw)
{
  return nerr_pass(tmpl_file_to(tmpl, parms, cs_to_gstring, out, raw));
}
```

One problem with providing a macro library in ClearSilver is that the final newline of a file will appear in the final output. There is no extremely efficient way to deal with this, so a simple awk program is used.

19c ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁15c 20d▷

```
CS_FILES=⟨CS files 47a⟩
```

19d ⟨*(build.nw) Build Source* 10b⟩+≡ ◁15f

```
$(CS_FILES) \
```

19e ⟨*(build.nw) Clean built files* 19e⟩≡

```
rm -f $(CS_FILES)
```

19f ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁16a 20h▷

```
# strip trailing \n - there has to be an easier way
$(CS_FILES): $(NOWEB_ORDER)
        notangle -R$@ $^ | awk '⟨Strip final newline 20a⟩' >$@
misc: $(CS_FILES)
```

20a ⟨*Strip final newline* 20a⟩≡ (19f)

```
{  if(prline) print line; line = $$0; prline = 1 } END { ORS = ""; print line; }
```

Templates are provided in this document, but if a program does not use them, there is no point in installing them. For this reason, a chunk is created for the installation instructions which can be added to ⟨*(build.nw) Install other files* 11c⟩ if desired. Since HTML files are generally templates as well, they are also added in this rule.

20b ⟨*Install ClearSilver templates* 20b⟩≡ 20g▷

```
mkdir -p $(DESTDIR)$(ETC_DIR)/tmpl
cp -p $(CS_FILES) $(HTML_FILES) $(DESTDIR)$(ETC_DIR)/tmpl
```

However, some HTML files may be just plain HTML that needs to be installed in the web server's document directory. For consistency, these are also copied into the template directory, but they are soft linked to the copy in the template directory. While the configuration option will always be visible, users can just ignore the option if there is no plain HTML.

20c ⟨*(build.nw) makefile.config* 7b⟩+≡ ◁8b

```
# Install directory for plain HTML files
HTML_DIR=/var/www/html
```

20d ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁19c 47d▷

```
PLAIN_HTML=⟨Plain HTML 20f⟩
```

20e ⟨*(build.nw) Plain Files* 20e⟩≡

```
⟨Plain HTML 20f⟩
```

20f ⟨*Plain HTML* 20f⟩≡ (20)

```
\
```

20g ⟨*Install ClearSilver templates* 20b⟩+≡ ◁20b

```
$(if $(PLAIN_HTML), \
mkdir -p $(DESTDIR)$(HTML_DIR); \
relup=; edir=$(ETC_DIR); hdir=$(HTML_DIR); \
edir=$${edir%/}/tmpl/; hdir=$${hdir%/}/; \
while [ $${edir#$${hdir}} = $$edir ]; do \
  relup=../$${relup}; \
  hdir=$${hdir%/*/}/; \
done; \
ltarget=$${relup}$${edir#/}; \
for x in $(PLAIN_HTML); do \
  ln -sf $${ltarget}/$$x $(DESTDIR)$(HTML_DIR)/$$x; \
done)
```

A check rule is added for running `htmltidy` on plain HTML files, since they should be complete and error-free.

Format	Example
%Y-%m-%d %T	2008-06-13 18:07:10
%Y-%m-%d %R	2008-06-13 18:07
%Y-%m-%d	2008-06-13
%m/%d/%Y %T	6/13/2008 18:07:10
%m/%d/%Y %R	6/13/2008 18:07
%m/%d/%Y	6/13/2008
%d-%m-%Y %T	13-06-2008 18:07:10
%d-%m-%Y %R	13-06-2008 18:07
%d-%m-%Y	13-06-2008
%T	18:07:10
%R	18:07
%a %T	Friday 18:07:10
%a %R	Friday 18:07
%a	Friday
%b %Y	June 2008
%b	June
%c	Fri Jun 13 18:07:10 2008
%x	06/13/08

Table 1: Default formats for `parse_date`

20h ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁19f 47e▷

```
ifneq ($(strip $(PLAIN_HTML)),)
check: checkhtml
checkhtml: $(PLAIN_HTML)
        for x in $(PLAIN_HTML); do \
          tidy -q -asxhtml -asxhtml -e $$x; \
        done
endif
```

There are a number of common things a user might want to do in a template, that are either not supported by ClearSilver or are unnecessarily inefficient and ugly in ClearSilver. ClearSilver has a rudimentary mechanism for adding string functions to the template expression language. It also has a method for adding more general functions, but that method is currently marked as experimental and likely to change. This can probably be worked around using version detection, although the lack of convenient version macros in ClearSilver will probably require the use of external definitions. The version has remained at 0.10.5 for more than a year now, so this might not be an issue. String functions may only take one string as input [6], and produce one string as output. Any functions which do not fit into that pattern use the generic extension facility. Generic functions are also the only method that gives access to the HDF being used for variable substitutions. Even generic functions are limited, though, in that they can only take a fixed number of arguments.

The following functions are added to support the common transformations needed by templates:

parse_date(*formats*, *date*) Parse an absolute date in server's current locale using `strptime`(3) or a relative date using a custom format and return as seconds since UNIX epoch. The *formats* string consists of zero or more formats separated by vertical bars (`|`). The formats in the "Default formats for `parse_date`" table are appended to the given formats, in order (examples are for Friday, June 13, 2008, 18:07:10, with adjustments described below). The first format which matches the *date* will be used.

Since `strptime`(3) does not support time zones, the time zone is always the server's local time zone. This can be changed for subsequent formats by using the format `TZ=`*zone*. Note that the format of

[6] Care must be taken to ensure that the parameter really is a string. Once an expression becomes a number, concatenation becomes addition (even if one side cannot be converted to a number), and these string conversion functions become no-ops. The only way to convert an expression back into a string is to call the built-in string function `string.slice` with zero as the start and an extremely large end.

the *zone* string is operating system-dependent. See `tzset`(3) for details. If any of hours, minutes, or seconds are not present, they are set to zero. If the month is specified, but not the day, the first of the month is used. If only the year is specified, January 1 is used. These formats also support some relative times. If the year is unspecified, it is the current year, unless the day of year is specified, either by month or day of year, and that day is on or after today, in which case it is last year. If only the weekday is specified, then it is the last such weekday, excluding today. In the previous examples, the %a format only works between Saturday, June 14, and Friday, June 20, and the %b format only works between July 1, 2008, and June 30, 2009.

For specifically relative dates, the date must consist of a star (*), implying the current date, optionally followed by an offset. The offset is an optional sign (+ or -; - is default), followed by an optional count (1 is the default), followed by an optional case-insensitve interval length indicator for seconds (S), minutes (’), hours (H), days (D) (the default), months (M), or years (Y). For a valid offset, one or the other of the count or the interval length indicator must be present. All parts of the date finer than the specified interval are set to their lowest value. For example, if the interval is in months, then the day is set to the first of the month, and the time is set to midnight.

Finally, a blank input returns 0, always.

22a ⟨*Initialize templates* 22a⟩≡ (18a) 26a▷

```
nerr_op(cs_register_function(cs, (char *)"parse_date", 2, cs_parse_date));
```

22b ⟨*tmpl.c* 17b⟩+≡ ◁19b 23b▷

```
static const char const * date_fmts[] = {
  "%Y-%m-%d %T",
  "%Y-%m-%d %R",
  "%Y-%m-%d",
  "%m/%d/%Y %T",
  "%m/%d/%Y %R",
  "%m/%d/%Y",
  "%d-%m-%Y %T",
  "%d-%m-%Y %R",
  "%d-%m-%Y",
  "%T",
  "%R",
  "%a %T",
  "%a %R",
  "%a",
  "%b",
  "%b %Y",
  "%c",
  "%x %X",
  "%x",
#if 0 /* these formats should just be specified in the argument instead */
  "%x %X",
  "%Ec",
  "%Ex",
  "%Ex %EX",
#endif
  NULL
};
```

22c ⟨*(build.nw) Common C Includes* 1⟩+≡ (46a) ◁4c 27a▷

```
#include <pthread.h>
```

22d ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁15a 27d▷

```
/* Lock this if modifying timezone information */
extern pthread_mutex_t tz_lock;
```

23a ⟨*cs_supt.c* 5e⟩+≡ ◁14b 28b▷

```
/* cheat: if it's already defined elsewhere, disable */
#ifndef TZ_LOCK_DEFINED
pthread_mutex_t tz_lock = PTHREAD_MUTEX_INITIALIZER;
#endif
```

23b ⟨*tmpl.c* 17b⟩+≡ ◁22b 25▷

```
time_t parse_date(char *formats, const char *date_str)
{
  char *s, *e = NULL;
  const char *fmt, **fmtp;
  struct tm tm_parsed, tm_ref, tm_now;
  time_t t;
  const char *old_tz = NULL; /* init to shut gcc up */
  gboolean set_tz = FALSE;
  gboolean got_err = FALSE;

  if(!date_str || !*date_str)
    return 0;
  memset(&tm_ref, 0xff, sizeof(tm_ref));
  t = time(NULL);
  localtime_r(&t, &tm_now);
  if(formats && *formats) {
    fmt = formats;
    while(1) {
      s = strchr(fmt, '|');
      if(s)
        *s++ = 0;
      if(!strncmp(fmt, "TZ=", 3)) {
        if(!set_tz) {
          pthread_mutex_lock(&tz_lock);
          old_tz = getenv("TZ");
          set_tz = TRUE;
        }
        setenv("TZ", fmt + 3, 1);
        tzset();
      } else {
        tm_parsed = tm_ref;
        if((e = strptime(date_str, fmt, &tm_parsed)) && !*e)
          break;
      }
      if(s)
        fmt = s;
      else
        break;
    }
  }
  for(fmtp = date_fmts; (!e || *e) && *fmtp; fmtp++) {
    tm_parsed = tm_ref;
    e = strptime(date_str, *fmtp, &tm_parsed);
  }
  if(!e || *e) {
    if(*date_str == '*') {
      gboolean add = TRUE;
      tm_parsed = tm_now;
      gint num_add = 1;
```

23b ⟨*tmpl.c* 17b⟩+≡ ◁22b 25▷

```
      if (*++date_str == '+')
        date_str++;
      else if (*date_str == '-') {
        date_str++;
        add = FALSE;
      }
      if (isdigit(*date_str)) {
        num_add = atoi(date_str);
        while (isdigit(*++date_str));
      }
      if (!add)
        num_add = -num_add;
      if (*date_str && date_str[1])
        got_err = TRUE; /* parse error */
      else
        switch (toupper(*date_str)) {
          case 'S':
            tm_parsed.tm_sec += num_add;
            break;
          case '\'':
            tm_parsed.tm_min += num_add;
            break;
          case 'H':
            tm_parsed.tm_hour += num_add;
            break;
          case 'D':
          case 0:
            tm_parsed.tm_mday += num_add;
            break;
          case 'M':
            tm_parsed.tm_mon += num_add;
            break;
          case 'Y':
            tm_parsed.tm_year += num_add;
            break;
          default:
            got_err = TRUE; /* parse error */
        }
      if (!got_err)
        /* all cases in following switch fall through */
        switch (toupper(*date_str)) {
          case 'Y':
            tm_parsed.tm_mon = 0;
          case 'M':
            tm_parsed.tm_mday = 1;
          case 'D':
          case 0:
            tm_parsed.tm_hour = 0;
          case 'H':
            tm_parsed.tm_min = 0;
          case '\'':
            tm_parsed.tm_sec = 0;
          case 'S':
            break;
        }
    } else
      got_err = TRUE; /* unknown */
  } else {
    /* adjust for partial specification; bias for the past */
    if (tm_parsed.tm_sec == tm_ref.tm_sec)
      tm_parsed.tm_sec = 0;
    if (tm_parsed.tm_min == tm_ref.tm_min)
      tm_parsed.tm_min = 0;
    if (tm_parsed.tm_hour == tm_ref.tm_hour)
```

23b ⟨*tmpl.c* 17b⟩+≡ ◁22b 25▷

```
      tm_parsed.tm_hour = 0;
    tm_parsed.tm_isdst = -1;
    if(tm_parsed.tm_year == tm_ref.tm_year) {
      tm_parsed.tm_year = tm_now.tm_year;
      if(tm_parsed.tm_mon != tm_ref.tm_mon) {
        if(tm_parsed.tm_mon > tm_now.tm_mon)
          tm_parsed.tm_year--;
        else if(tm_parsed.tm_mon == tm_now.tm_mon &&
                (tm_parsed.tm_mday == tm_ref.tm_mday ||
                 tm_parsed.tm_mday > tm_now.tm_mday))
          tm_parsed.tm_year--;
      } else if(tm_parsed.tm_yday != tm_ref.tm_yday) {
        if(tm_parsed.tm_yday > tm_now.tm_yday)
          tm_parsed.tm_year--;
        tm_parsed.tm_mon = 0;
        tm_parsed.tm_mday = tm_parsed.tm_yday;
      }
    }
    if(tm_parsed.tm_mday == tm_ref.tm_mday) {
      if(tm_parsed.tm_yday != tm_ref.tm_yday) {
        tm_parsed.tm_mday = tm_parsed.tm_yday;
        tm_parsed.tm_mon = 0;
      } else if(tm_parsed.tm_wday != tm_ref.tm_wday) {
        if(tm_parsed.tm_wday >= tm_now.tm_wday)
          tm_parsed.tm_wday -= 7;
        tm_parsed.tm_mday = tm_now.tm_mday;
        tm_parsed.tm_mday -= tm_now.tm_wday - tm_parsed.tm_wday;
      } else if(tm_parsed.tm_mon != tm_ref.tm_mon)
        tm_parsed.tm_mday = 1;
      else
        tm_parsed.tm_mday = tm_now.tm_mday;
    }
    if(tm_parsed.tm_mon == tm_ref.tm_mon)
      tm_parsed.tm_mon = tm_now.tm_mon;
  }
  t = mktime(&tm_parsed);
  if(set_tz) {
    if(old_tz)
      setenv("TZ", old_tz, 1);
    else
      unsetenv("TZ");
    tzset();
    pthread_mutex_unlock(&tz_lock);
  }
  if(got_err)
    return 0;
  else
    return t;
}
```

25 ⟨*tmpl.c* 17b⟩+≡ ◁23b 26b▷

```
NEOERR *cs_parse_date(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  NEOERR *nerr = STATUS_OK;
  char *formats = NULL, *date_str = NULL;
  result->op_type = CS_TYPE_NUM;
  result->s = NULL;
  nerr = cs_arg_parse(cs, args, "ss", &formats, &date_str);
  if(nerr == STATUS_OK)
    /* n is long, so this will fail in 2038 on 32-bit systems */
    result->n = parse_date(formats, date_str);
  if(formats)
    free(formats);
```

25 ⟨*tmpl.c* 17b⟩+≡ ◁23b 26b▷

```
  if(date_str)
    free(date_str);
  return nerr;
}
```

format_date(*format*, *timestamp*) Format date using `strftime`(3), with optional *format*. A blank *format* is replaced by %c. The *timestamp* is seconds since UNIX epoch.

26a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁22a 27b▷

```
nerr_op(cs_register_function(cs, (char *)"format_date", 2, cs_format_date));
```

26b ⟨*tmpl.c* 17b⟩+≡ ◁25 27c▷

```
NEOERR *cs_format_date(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  char *buf;
  guint buflen, ret;
  long t_arg;
  time_t t;
  char *fmt_arg = NULL;
  const char *fmt;
  struct tm tm_in;
  NEOERR *nerr;

  nerr = cs_arg_parse(cs, args, "si", &fmt_arg, &t_arg);
  if(nerr != STATUS_OK) {
    if(fmt_arg)
      free(fmt_arg);
    return nerr;
  }
  fmt = fmt_arg;
  /* 32-bit systems will fail in 2038 due to use of long */
  t = t_arg;
  if(!fmt || !*fmt)
    fmt = "%c";
  localtime_r(&t, &tm_in);
  buflen = strlen(fmt) * 4;
  buf = malloc(buflen);
  while(buf) {
    ret = strftime(buf, buflen, fmt, &tm_in);
    if(ret && ret < buflen)
      break;
    buflen *= 2;
    if(buflen >= 1024) {
      free(buf);
      buf = NULL;
    } else {
      char *tmp = realloc(buf, buflen);

      if(!tmp)
        free(buf);
      buf = tmp;
    }
  }
  result->op_type = CS_TYPE_STRING;
  result->n = 0;
  result->s = buf;
  result->alloc = 1;
  free(fmt_arg);
  if(!buf)
```

26b ⟨*tmpl.c* 17b⟩+≡ ◁25 27c▷

```
    return nerr_raise_msg_errno("No memory for conversion");
  return STATUS_OK;
}
```

url_unescape(*string*) URL-decode *string* (URL-encoding is provided by the CGI library as `url_escape(string)`)

27a ⟨*(build.nw) Common C Includes* 1⟩+≡ (46a) ◁22c 38d▷

```
#include <cgi/cgi.h>
```

27b ⟨*Initialize templates* 22a⟩+≡ (18a) ◁26a 31a▷

```
if(nerr == STATUS_OK) {
  nerr = cgi_register_strfuncs(cs);
  nerr_ignore(&nerr);
}
nerr_op(cs_register_strfunc(cs, (char *)"url_unescape", cs_url_unescape));
```

27c ⟨*tmpl.c* 17b⟩+≡ ◁26b 31b▷

```
NEOERR *cs_url_unescape(const char *in, char **out)
{
  *out = strdup(in);
  if(!*out)
    return nerr_raise_msg("No memory for conversion");
  cgi_url_unescape(*out);
  return STATUS_OK;
}
```

json_escape(*variable-name*) Escape *variable* and its children using JavaScript Object Notation (JSON). See `http://www.json.org` for the complete format specification. Nonexistent variables and variables without values or children are output as `null`. Variables with values are output as strings by default, but may be output as numbers by adding the `num` attribute, and may be output as booleans by adding the `bool` attribute; booleans with no value, blank values, or the value `0` are output as `false`. Variables with children are output as objects by default, but may be output as arrays (i.e., children names are sorted, but not output) by adding the `array` attribute; arrays with no value are output as an empty array, but there is no way to output an empty object (`{}`). String values must be UTF-8 encoded. If any variables have raw values which may conflict with UTF-8, they should be double-encoded using e.g. URL-encoding to ensure that their value is converted to ASCII first.

It would be nice to be able to pass in the variable rather than passing in the variable's name, but the ClearSilver function support only supports integer and string variables, and not object references. JSON encoding is not supported by any functions in the libraries being used, so a generic `GString` version is provided as well.

First, in order to support the different data types with indistinguishable internal representations, flags can be set on variables using HDF attributes. These attributes can be set using the API or within an HDF using the bracketed attribute name immeidately after the variable name. There is no way to set attributes within templates, so some sort of skeleton HDF file would need to be read in and used for that.

27d ⟨*ClearSilver Support Globals* 5b⟩+≡ (5a) ◁22d

```
typedef enum json_var_type {
  JSON_UNK, JSON_ARRAY, JSON_BOOL, JSON_NUM
} json_var_type;
```

28a ⟨*(build.nw) Known Data Types* 2g⟩+≡ ◁15b 58▷

```
json_var_type,%
```

28b ⟨*cs_supt.c* 5e⟩+≡ ◁23a 28c▷

```
json_var_type hdf_json_var_type(HDF *obj)
{
  HDF_ATTR *attrs = hdf_obj_attr(obj);

  while(attrs) {
    if(!strcmp(attrs->key, "num"))
      return JSON_NUM;
    else if(!strcmp(attrs->key, "bool"))
      return JSON_BOOL;
    else if(!strcmp(attrs->key, "array"))
      return JSON_ARRAY;
    attrs = attrs->next;
  }
  return JSON_UNK;
}
```

The conversion function recursively appends values to a string. If the top level has children, either an object (attribute-value pairs, which include the name of each child) or an array (values only; the names are just sorted before display) is appended.

28c ⟨*cs_supt.c* 5e⟩+≡ ◁28b 30b▷

```
GString *g_string_append_json(GString *buf, HDF *hdf)
{
  json_var_type vt = hdf_json_var_type(hdf);

  if(hdf_obj_child(hdf)) {
    if(vt == JSON_ARRAY) {
      char c = '[';
      hdf_sort_obj(hdf, comp_hdf_name);
      for(hdf = hdf_obj_child(hdf); hdf; hdf = hdf_obj_next(hdf)) {
        g_string_append_c(buf, c);
        c = ',';
        g_string_append_json(buf, hdf);
      }
      g_string_append_c(buf, ']');
    } else {
      char c = '{';
      for(hdf = hdf_obj_child(hdf); hdf; hdf = hdf_obj_next(hdf)) {
        g_string_append_c(buf, c);
        c = ',';
        append_unicode_quoted(buf, hdf_obj_name(hdf));
        g_string_append_c(buf, ':');
        g_string_append_json(buf, hdf);
      }
      g_string_append_c(buf, '}');
    }
    return buf;
  }
  ⟨Append JSON-encoded hdf value to buf 29a⟩
  return buf;
}
```

If there are no children, the value is appended. Technically, a `NULL` value could indicate either an empty object, an empty array, a boolean `false`, or `null`. If the array flag is set, an empty array is appended. If the

boolean flag is set, a boolean `false` is appended. There is no way to distinguish empty objects and `null`, so `null` is appended.

29a ⟨*Append JSON-encoded* `hdf` *value to* `buf` 29a⟩≡ (28c) 29b▷

```
const char *s = hdf_obj_value(hdf);

if (!s) {
  if (vt == JSON_BOOL)
    g_string_append(buf, "false");
  else if (vt == JSON_ARRAY)
    g_string_append(buf, "[]");
  else
    g_string_append(buf, "null");
  return buf;
}
```

For ClearSilver boooleans, empty strings and zero are false. Technically, if the zero is considered a string, it is not false, but there is no way to easily distinguish. There is also no real need to check for other forms of zero, such as hexadecimal zero or multiple zero digits.

29b ⟨*Append JSON-encoded* `hdf` *value to* `buf` 29a⟩+≡ (28c) ◁29a 29c▷

```
if (vt == JSON_BOOL) {
  if (!*s || !strcmp(s, "0"))
    g_string_append(buf, "false");
  else
    g_string_append(buf, "true");
  return buf;
}
```

Numbers are only appended as numbers if they are really valid numbers. JSON does not support octal or hexadecimal numbers, but this code does, and converts it to decimal before printing. If it is not a valid number, it falls through to string formatting.

29c ⟨*Append JSON-encoded* `hdf` *value to* `buf` 29a⟩+≡ (28c) ◁29b 30a▷

```
if (vt == JSON_NUM && (*s == '-' || isdigit(*s))) {
  const char *d = *s == '-' ? s + 1 : s;
  if ((*d >= '1' && *d <= '9') || (*d == '0' && (!d[1] || d[1] == '.'))) {
    while (isdigit(*++d));
    if (*d == '.' && isdigit(d[1])) {
      while (isdigit(*++d));
      if (*d == 'e' || *d == 'E') {
        if (d[1] == '+' || d[1] == '-')
          d++;
        if (isdigit(d[1]))
          while (isdigit(*++d));
      }
    }
    if (!*d) {
      g_string_append(buf, s);
      return buf;
    }
  } else if (*d == '0') {
    char *e;
    unsigned long long v = strtoull(s, &e, 0);
```

29c ⟨*Append JSON-encoded* `hdf` *value to* `buf` 29a⟩+≡ (28c) ◁29b 30a▷

```
    if(!*e) {
      g_string_append_printf(buf, "%lld", v);
      return buf;
    }
  }
}
```

Strings are only complicated in that they must be UTF-8 encoded. It is assumed that they already are; in most cases, this means that the string must be ASCII to begin with. As a fallback, if the string fails UTF-8 validation, it is printed as ASCII with special characters escaped as though they were unicode.

30a ⟨*Append JSON-encoded* `hdf` *value to* `buf` 29a⟩+≡ (28c) ◁29c

```
append_unicode_quoted(buf, s);
```

30b ⟨*cs_supt.c* 5e⟩+≡ ◁28c 31c▷

```
GString *append_unicode_quoted(GString *buf, const char *s)
{
  g_string_append_c(buf, '"');
  if(g_utf8_validate(s, -1, NULL))
    while(*s) {
      gunichar uc = g_utf8_get_char(s);
      s = g_utf8_next_char(s);
      ⟨Tack escaped uc to buf 30c⟩
    }
  else
    while(*s) {
      unsigned char uc = *s++;
      ⟨Tack escaped uc to buf 30c⟩
    }
  g_string_append_c(buf, '"');
  return buf;
}
```

30c ⟨*Tack escaped* `uc` *to* `buf` 30c⟩≡ (30b)

```
if(uc < 128 && isprint(uc) && uc != '"' && uc != '\\')
  g_string_append_c(buf, uc);
else switch(uc) {
  case '\b':
    g_string_append(buf, "\\b");
    break;
  case '\t':
    g_string_append(buf, "\\t");
    break;
  case '\f':
    g_string_append(buf, "\\f");
    break;
  case '\n':
    g_string_append(buf, "\\n");
    break;
  case '\r':
    g_string_append(buf, "\\r");
    break;
  case '\\':
  case '"':
  case '/':
    g_string_append_c(buf, '\\');
    g_string_append_c(buf, uc);
    break;
  default:
```

30c ⟨*Tack escaped* `uc` *to* `buf` 30c⟩≡ (30b)

```
    g_string_append_printf(buf, "\\u%04x", uc);
    break;
}
```

Finally, the template function is pretty simple, except that it needs to be a full callback in order to access the HDF.

31a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁27b 36a▷

```
nerr_op(cs_register_function(cs, (char *)"json_escape", 1, cs_json_escape));
```

31b ⟨*tmpl.c* 17b⟩+≡ ◁27c 36b▷

```
NEOERR *cs_json_escape(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  NEOERR *nerr = STATUS_OK;
  HDF *src;
  char *varname;
  GString *json;

  nerr = cs_arg_parse(cs, args, "s", &varname);
  result->op_type = CS_TYPE_STRING;
  if (nerr == STATUS_OK) {
    json = g_string_new("");
    src = hdf_get_obj(cs->hdf, varname);
    if (src)
      g_string_append_json(json, src);
    else
      g_string_assign(json, "null");
    result->s = g_string_free(json, FALSE);
  } else
   result->s = NULL;
  if (varname)
    free(varname);
  return nerr;
}
```

json_unescape(*variable-name*, *value*) Set *variable* to the result of parsing *value* using JavaScript Object Notation (JSON). See `http://www.json.org` for the complete format specification. Arrays will have their `array` attribute set and their children will receive numeric sequence names. Booleans will be set to `1` for `true` and blank for `false` and have their `bool` attribute set. Numbers will have their `num` attribute set. Unicode escapes in the string will be encoded as per UTF-8.

Reading JSON is probably not something C (or template) code will ever do, but nonetheless, here is the reverse operation. Just in case the string has more than one value encoded, an end pointer is provided. This is actually used in recursive calls, since the value is embedded in a structure. Reading from a JSON file might be useful as well, but it requires a character of lookahead, and is probably not useful enough to require a separate function in order to allow reading in large volumes of data.

31c ⟨*cs_supt.c* 5e⟩+≡ ◁30b 35b▷

```
NEOERR *json_from_string(const char *s, HDF *hdf)
{
  NEOERR *nerr = nerr_pass(embedded_json_from_string(s, hdf, &s));
  if (nerr == STATUS_OK) {
    while (isspace(*s))
      s++;
    if (*s)
```

31c ⟨*cs_supt.c* 5e⟩+≡ ◁30b 35b▷

```
      return nerr_raise_msg("Invalid JSON: Garbage after value");
  }
  return nerr;
}

NEOERR *embedded_json_from_string(const char *s, HDF *hdf, const char **ep)
{
  NEOERR *nerr = STATUS_OK;
  ⟨Convert JSON to HDF 32a⟩
  return nerr;
}
```

Any JSON value may have leading or trailing space.

32a ⟨*Convert JSON to HDF* 32a⟩≡ (31c) 32b▷

```
while (isspace(*s))
  s++;
if (!*s)
  return nerr_raise_msg("Invalid JSON: Expected value");
```

Objects start with curly braces. A recursive call is used to parse values. Names could be processed by doing a recursive call with a dummy HDF object as its target, but instead the string processing is done in a separate function.

32b ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁32a 33a▷

```
if (*s == '{') {
  gboolean gotone = FALSE;
  HDF *cn;
  GString *buf = NULL;

  s++;
  while (nerr == STATUS_OK) {
    while (isspace(*s))
      s++;
    if (*s == '}')
      break;
    if (gotone) {
      if (*s != ',') {
        nerr = nerr_raise_msg("Invalid JSON: expected , or }");
        break;
      }
      while (isspace(*++s));
    }
    if (*s != '"') {
      nerr = nerr_raise_msg("Invalid JSON: expected quoted name");
      break;
    }
    gotone = TRUE;
    if (!buf)
      buf = g_string_new("");
    else
      g_string_truncate(buf, 0);
    nerr = parse_json_string(s, buf, &s);
    if (nerr == STATUS_OK) {
      while (isspace(*s))
        s++;
      if (*s != ':') {
        nerr = nerr_raise_msg("Invalid JSON: expected :");
        break;
      }
```

32b ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁32a 33a▷

```
      s++;
    }
    nerr_op(hdf_get_node(hdf, buf->str, &cn));
    nerr_op(embedded_json_from_string(s, cn, &s));
  }
  if(buf)
    g_string_free(buf, TRUE);
  if(ep && nerr == STATUS_OK)
    *ep = s + 1;
  return nerr;
}
```

Arrays start with square brackets. A recursive call is used to parse values, and names are just sequential numbers. Even though the function is inteded to be called with an empty object, an attempt is made to avoid any existing children. This means that if values are not intended to be merged, they should first be destroyed.

33a ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁32b 33b▷

```
if(*s == '[') {
  gboolean gotone = FALSE;
  int idx = -1;
  char idx_buf[22];
  HDF *cn;

  s++;
  for(cn = hdf_obj_child(hdf); cn; cn = hdf_obj_next(cn)) {
    int cn_idx = atol(hdf_obj_name(cn));
    if(cn_idx >= idx)
      idx = cn_idx;
  }
  hdf_set_attr(hdf, NULL, "array", "1");
  while(nerr == STATUS_OK) {
    while(isspace(*s))
      s++;
    if(*s == ']')
      break;
    if(gotone) {
      if(*s != ',') {
        nerr = nerr_raise_msg("Invalid JSON: expected , or ]");
        break;
      }
      while(isspace(*++s));
    }
    gotone = TRUE;
    sprintf(idx_buf, "%d", ++idx);
    nerr = hdf_get_node(hdf, idx_buf, &cn);
    nerr_op(embedded_json_from_string(s, cn, &s));
  }
  if(ep && nerr == STATUS_OK)
    *ep = s + 1;
  return nerr;
}
```

Boolean values are unquoted literals.

33b ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁33a 34a▷

```
#define is_end(c) (!(c) || isspace(c) || (c) == ',' || (c) == ']' || (c) == '}')
if(!strncmp(s, "true", 4) && is_end(s[4])) {
  nerr = hdf_set_attr(hdf, NULL, "bool", "1");
  nerr_op(hdf_set_value(hdf, NULL, "1"));
  if(ep)
    *ep = s + 4;
```

33b ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁33a 34a▷

```
    return nerr;
  }
  if(!strncmp(s, "false", 5) && is_end(s[5])) {
    nerr = hdf_set_attr(hdf, NULL, "bool", "1");
    nerr_op(hdf_set_value(hdf, NULL, "0"));
    if(ep)
      *ep = s + 5;
    return nerr;
  }
```

An NULL value should probably clear out the entire value, if it already exists. Instead, it is simply ignored, on the assumption that the value is already NULL.

34a ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁33b 34b▷

```
  if(!strncmp(s, "null", 4) && is_end(s[4])) {
    if(ep)
      *ep = s + 4;
    return STATUS_OK;
  }
```

Numbers can be parsed using `strtol` or `strtod`, but their format needs to be validated first. Since all HDF values are basically strings, there is no need to really parse the number. For floating point values, precision would be lost by such an action, anyway. JSON also provides no range for valid values, so it is possible to have numbers outside of the range for even `strtoull`.

34b ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁34a 34c▷

```
  if(isdigit(*s) || (*s == '-' && isdigit(s[1]))) {
    const char *e = *s == '-' ? s + 1 : s;
    gboolean is_float;

    if(*e == '0' && isdigit(e[1])) /* octal; explicitly forbidden */
      return nerr_raise_msg("Invalid JSON: octal numbers not allowed");
    while(isdigit(*++e));
    is_float = *e == '.';
    if(is_float && isdigit(e[1])) {
      while(isdigit(*++e));
      if(*e == 'e' || *e == 'E') {
        if((e[1] == '+' || e[1] == '-') && isdigit(e[2]))
          ++e;
        if(!isdigit(e[1]))
          return nerr_raise_msg("Invalid JSON: invalid exponent");
        while(isdigit(*++e));
      }
    }
    if(!is_end(*e))
      return nerr_raise_msg("Invalid JSON: invalid numeric format");
    GString *buf = g_string_new_len(s, (int)(e - s));
    if(ep)
      *ep = e;
    nerr = hdf_set_attr(hdf, NULL, "num", "1");
    return hdf_set_buf(hdf, NULL, g_string_free(buf, FALSE));
  }
```

Finally, as mentioned above, strings get their own processing routine. This makes the parser portion pretty simple.

34c ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁34b 35a▷

```
if(*s == '"') {
  GString *buf = g_string_new("");
  nerr = parse_json_string(s, buf, ep);
  nerr_op(hdf_set_buf(hdf, NULL, g_string_free(buf, FALSE)));
  return nerr;
}
```

That covers all possible value formats.

35a ⟨*Convert JSON to HDF* 32a⟩+≡ (31c) ◁34c

```
return nerr_raise_msg("Invalid JSON: unrecognized value format");
```

35b ⟨*cs_supt.c* 5e⟩+≡ ◁31c 40a▷

```
NEOERR *parse_json_string(const char *s, GString *buf, const char **ep)
{
  if(*s != '"')
    return nerr_raise_msg("Invalid JSON: expected \"");
  for(s++; *s && *s != '"'; s++) {
    if(*s != '\\') {
      g_string_append_c(buf, *s);
      continue;
    }
    switch(*++s) {
      case '"':
      case '\\':
      case '/':
        g_string_append_c(buf, *s);
        break;
      case 'b':
        g_string_append_c(buf, '\b');
        break;
      case 'f':
        g_string_append_c(buf, '\f');
        break;
      case 'n':
        g_string_append_c(buf, '\n');
        break;
      case 'r':
        g_string_append_c(buf, '\r');
        break;
      case 't':
        g_string_append_c(buf, '\t');
        break;
      case 'u':
        if(!isxdigit(*++s) || !isxdigit(*++s) || !isxdigit(*++s) ||
           !isxdigit(*++s))
          return nerr_raise_msg("Invalid JSON");
        char unibuf[6];
        int len, c;

        sscanf(s - 4, "%04x", &c);
        len = g_unichar_to_utf8(c, unibuf);
        g_string_append_len(buf, unibuf, len);
        break;
      default:
        return nerr_raise_msg("Invalid JSON: invalid string format");
    }
  }
  if(*s == '"')
    s++;
  else
```

35b ⟨*cs_supt.c* 5e⟩+≡ ◁31c 40a▷

```
    return nerr_raise_msg("Invalid JSON: no terminating \" on string");
  if(ep)
    *ep = s;
  return STATUS_OK;
}
```

36a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁31a 36c▷

```
nerr_op(cs_register_function(cs, (char *)"json_unescape", 2, cs_json_unescape));
```

36b ⟨*tmpl.c* 17b⟩+≡ ◁31b 36d▷

```
NEOERR *cs_json_unescape(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args,
                         CSARG *result)
{
  NEOERR *nerr = STATUS_OK;
  HDF *target;
  char *varname, *val;

  nerr = cs_arg_parse(cs, args, "ss", &varname, &val);
  nerr_op(hdf_get_node(cs->hdf, varname, &target));
  nerr_op(json_from_string(val, target));
  /* no void functions, so return 1 on success (i.e., always) */
  result->op_type = CS_TYPE_NUM;
  result->s = NULL;
  result->n = nerr == STATUS_OK;
  if(varname)
    free(varname);
  if(val)
    free(val);
  return nerr;
}
```

to_hex(*bin*) Convert a URL-encoded binary blob to a string of lower-case hexadecimal digits. This is mostly for displaying checksums. The blob must be URL-encoded so that zeroes are handled properly in the string.

36c ⟨*Initialize templates* 22a⟩+≡ (18a) ◁36a 37a▷

```
nerr_op(cs_register_strfunc(cs, (char *)"to_hex", cs_to_hex));
```

36d ⟨*tmpl.c* 17b⟩+≡ ◁36b 37b▷

```
NEOERR *cs_to_hex(const char *in, char **out)
{
  guint8 c, *op;

  *out = malloc(strlen(in) * 2 + 1);
  if(!*out)
    return nerr_raise_msg("No memory for conversion");
  op = (guint8 *)*out;
  while(*in) {
    if(*in == '%' && isxdigit(in[1]) && isxdigit(in[2])) {
      c = *++in & 0xf;
      if(*in > '9')
        c += 10 - ('a' & 0xf);
      c <<= 4;
      c += *++in & 0xf;
      if(*in > '9')
```

36d ⟨*tmpl.c* 17b⟩+≡ ◁36b 37b▷

```
        c += 10 - ('a' & 0xf);
    } else if(*in == '+')
      c = ' ';
    else
      c = *in;
    *op = (c >> 4) + '0';
    if(c > 0x9f)
      *op += 'a' - '0' - 10;
    *++op = (c &= 0xf) + '0';
    if(c > 9)
      *op += 'a' - '0' - 10;
    op++;
    in++;
  }
  *op = 0;
  return STATUS_OK;
}
```

from_hex(*hex*) Convert a string of hexadecimal digits to a URL-encoded binary blob. This is mostly for converting checksums from their usual format to that expected by these utilities. The return must be URL-encoded to prevent issues with zeroes in string returns.

37a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁36c 37c▷

```
nerr_op(cs_register_strfunc(cs, (char *)"from_hex", cs_from_hex));
```

37b ⟨*tmpl.c* 17b⟩+≡ ◁36d 37d▷

```
NEOERR *cs_from_hex(const char *in, char **out)
{
  char *op;

  *out = op = malloc(strlen(in) * 3 / 2 + 1);
  if(!*out)
    return nerr_raise_msg("No memory for conversion");
  while(isxdigit(*in) && isxdigit(in[1])) {
    *op++ = '%';
    *op++ = toupper(*in++);
    *op++ = toupper(*in++);
  }
  if(*in)
    return nerr_raise_msg("Invalid hex string");
  *op = 0;
  return STATUS_OK;
}
```

to_lower(*string*) Convert *string* to lower-case in the server's current locale. This is done a byte at a time using the `ctype.h` function. This means that the string will probably be mangled in UTF-8 locales or if the input locale does not match the server's locale.

37c ⟨*Initialize templates* 22a⟩+≡ (18a) ◁37a 38a▷

```
nerr_op(cs_register_strfunc(cs, (char *)"to_lower", cs_to_lower));
```

37d ⟨*tmpl.c* 17b⟩+≡ ◁37b 38b▷

```
NEOERR *cs_to_lower(const char *in, char **out)
{
  char *op;

  *out = op = malloc(strlen(in) + 1);
  if(!*out)
    return nerr_raise_msg("No memory for conversion");
  while(*in)
    *op++ = tolower(*in++);
  *op = 0;
  return STATUS_OK;
}
```

to_upper(*string*) Convert *string* to upper-case. The comments for `to_lower()` apply here as well.

38a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁37c 38c▷

```
nerr_op(cs_register_strfunc(cs, (char *)"to_upper", cs_to_upper));
```

38b ⟨*tmpl.c* 17b⟩+≡ ◁37d 38e▷

```
NEOERR *cs_to_upper(const char *in, char **out)
{
  char *op;

  *out = op = malloc(strlen(in) + 1);
  if(!*out)
    return nerr_raise_msg("No memory for conversion");
  while(*in)
    *op++ = toupper(*in++);
  *op = 0;
  return STATUS_OK;
}
```

pw_uid(*id*) Return the password file entry for the given user ID on the local system. Since this uses the system function, it may not be a real file entry, but it is returned in the format of one.

38c ⟨*Initialize templates* 22a⟩+≡ (18a) ◁38a 39b▷

```
nerr_op(cs_register_function(cs, (char *)"pw_uid", 1, cs_pw_uid));
```

38d ⟨*(build.nw) Common C Includes* 1⟩+≡ (46a) ◁27a 39e▷

```
#include <pwd.h>
```

38e ⟨*tmpl.c* 17b⟩+≡ ◁38b 39c▷

```
/* not reentrant, but not for threaded programs */
NEOERR *cs_pw_uid(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  long uid;
  NEOERR *nerr = cs_arg_parse(cs, args, "i", &uid);
  if(nerr != STATUS_OK)
    return nerr;
  result->op_type = CS_TYPE_STRING;
  result->n = 0;
  char *s;
```

38e ⟨*tmpl.c* 17b⟩+≡ ◁38b 39c▷

```
  struct passwd *pw = getpwuid((uid_t)uid);
  ⟨Set s to passwd file entry 39a⟩
  result->s = s;
  result->alloc = 1;
  return STATUS_OK;
}
```

39a ⟨*Set* s *to passwd file entry* 39a⟩≡ (38e 39c)

```
GString *res;

if(!pw)
  return nerr_raise_msg_errno("unknown user");
res = g_string_new("");
g_string_printf(res, "%s:*:%ld:%ld:%s:%s:%s",
                pw->pw_name, /* pw->pw_passwd, */ (long)pw->pw_uid,
                (long)pw->pw_gid, pw->pw_gecos, pw->pw_dir, pw->pw_shell);
s = g_string_free(res, FALSE);
```

pw_nam(*name*) Return the password file entry for the given user name on the local system. Since this uses the system function, it may not be a real file entry, but it is returned in the format of one.

39b ⟨*Initialize templates* 22a⟩+≡ (18a) ◁38c 39d▷

```
nerr_op(cs_register_strfunc(cs, (char *)"pw_nam", cs_pw_nam));
```

39c ⟨*tmpl.c* 17b⟩+≡ ◁38e 39f▷

```
NEOERR *cs_pw_nam(const char *in, char **out)
{
  struct passwd *pw = getpwnam(in);
  char *s;
  ⟨Set s to passwd file entry 39a⟩
  *out = s;
  return STATUS_OK;
}
```

re_match(*re*, *string*) Return integer flag 1 if URL-encoded POSIX extended regular expression *re* matches (non-URL-encoded) *string*.

39d ⟨*Initialize templates* 22a⟩+≡ (18a) ◁39b 40b▷

```
nerr_op(cs_register_function(cs, (char *)"re_match", 2, cs_re_match));
```

39e ⟨*(build.nw) Common C Includes* 1⟩+≡ (46a) ◁38d

```
#include <regex.h>
```

39f ⟨*tmpl.c* 17b⟩+≡ ◁39c 41▷

```
NEOERR *cs_re_match(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  regex_t re;
  int ret;
  char *in = NULL, *re_s = NULL;
  NEOERR *nerr;
```

39f ⟨*tmpl.c* 17b⟩+≡ ◁39c 41▷

```
  nerr = cs_arg_parse(cs, args, "ss", &re_s, &in);
  if(nerr != STATUS_OK) {
    if(in)
      free(in);
    if(re_s)
      free(re_s);
    return nerr;
  }
  cgi_url_unescape(re_s);
  ret = regcomp(&re, re_s, REG_EXTENDED | REG_NOSUB);
  if(ret) {
    nerr = nerr_pass(regerr_nerr(re_s, &re, ret));
    free(re_s);
    free(in);
    return nerr;
  }
  free(re_s);
  result->s = NULL;
  result->op_type = CS_TYPE_NUM;
  result->n = !regexec(&re, in, 0, NULL, 0);
  regfree(&re);
  free(in);
  return STATUS_OK;
}
```

40a ⟨*cs_supt.c* 5e⟩+≡ ◁35b 46b▷

```
NEOERR *regerr_nerr(const char *prefix, regex_t *re, int ret)
{
  size_t errlen = regerror(ret, re, NULL, 0);
  char *ebuf = malloc(errlen + 1);
  NEOERR *nerr;

  if(!ebuf)
    return nerr_raise_msg_errno("Regex Error");
  else {
    regerror(ret, re, ebuf, errlen + 1);
    nerr = nerr_raise(GENERIC_ERR, "%s: %s", prefix, ebuf);
    free(ebuf);
    return nerr;
  }
}
```

re_subst(*re*, *subst*, *string*) Substitute occurrences of URL-encoded POSIX extended regular expression *re* with URL-encoded *subst* string in (non-URL-encoded) *string*. In *subst*, backslash escapes are supported:

- \\ means backslash
- \0 means the entire matching string
- *n*, where *n* is a decimal number, means the n^{th} parenthesized subexpression
- \, means nothing (so you can separate *n* from literal decimal digits)
- \g also means nothing, but has the side effect that all occurrences of the regular expression are substituted. If this is not present, only the first occurrence is substituted.

The behavior of any other character following a backslash, or a backslash at the end of a string, is undefined.

40b ⟨*Initialize templates* 22a⟩+≡ (18a) ◁39d 42a▷

```
nerr_op(cs_register_function(cs, (char *)"re_subst", 3, cs_re_subst));
```

41 ⟨*tmpl.c* 17b⟩+≡ ◁39f 42b▷

```
NEOERR *cs_re_subst(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  regex_t re;
  int ret;
  char *in_p = NULL, *in, *subst = NULL, *re_s = NULL;
  char *s;
  NEOERR *nerr;
  int matchno, max_match;
  regmatch_t *matches;
  GString *subst_out;
  gboolean global = FALSE;
  int exec_opts;

  nerr = cs_arg_parse(cs, args, "sss", &re_s, &subst, &in_p);
  if(nerr != STATUS_OK) {
    if(in_p)
      free(in_p);
    if(re_s)
      free(re_s);
    if(subst)
      free(subst);
    return nerr;
  }
  cgi_url_unescape(subst);
  for(s = subst, max_match = 0; *s; s++)
    if(*s == '\\') {
      if(isdigit(s[1])) {
        matchno = atoi(s + 1);
        if(matchno > max_match)
          max_match = matchno;
      } else if(s[1]) {
        s++;
        if(*s == 'g')
          global = TRUE;
      }
    }
  matches = calloc(sizeof(*matches), max_match + 1);
  if(!matches) {
    free(re_s);
    free(subst);
    free(in_p);
    return nerr_raise_msg_errno("No memory for subst");
  }
  cgi_url_unescape(re_s);
  ret = regcomp(&re, re_s, REG_EXTENDED);
  if(ret) {
    free(matches);
    nerr = nerr_pass(regerr_nerr(re_s, &re, ret));
    free(re_s);
    return nerr;
  }
  free(re_s);
  subst_out = g_string_new("");
  exec_opts = 0;
  in = in_p;
  while(!regexec(&re, in, max_match + 1, matches, exec_opts)) {
    char *b;

    if(matches[0].rm_so)
```

41 ⟨*tmpl.c* 17b⟩+≡ ◁39f 42b▷

```
      g_string_append_len(subst_out, in, matches[0].rm_so);
    s = subst;
    while((b = strchr(s, '\\'))) {
      if(b != s)
        g_string_append_len(subst_out, s, (int)(b - s));
      if(isdigit(*++b)) {
        matchno = atoi(b);
        while(isdigit(*++b));
        if(matches[matchno].rm_so != -1 &&
           matches[matchno].rm_eo > matches[matchno].rm_so)
          g_string_append_len(subst_out, in + matches[matchno].rm_so,
                              matches[matchno].rm_eo - matches[matchno].rm_so);
      } else if(*b) {
        if(*b != 'g' && *b != ',')
          g_string_append_c(subst_out, *b);
        b++;
      }
      s = b;
    }
    if(*s)
      g_string_append(subst_out, s);
    /* non-overlapping global, like sed */
    in += matches[0].rm_eo;
    if(!global)
      break;
    exec_opts |= REG_NOTBOL;
    /* only allow empty patterns to match once */
    if(!matches[0].rm_so && !matches[0].rm_eo)
      break;
  }
  if(*in)
    g_string_append(subst_out, in);
  free(matches);
  free(subst);
  free(in_p);
  regfree(&re);
  result->n = 0;
  result->op_type = CS_TYPE_STRING;
  result->alloc = 1;
  result->s = g_string_free(subst_out, FALSE);
  return STATUS_OK;
}
```

re_extract(*variable* [+ ',' + *variable* ...], *re*, *string*) Extract substrings of (non-URL-encoded) *string* matching parenthesized subexpressions in URL-encoded POSIX extended regular expression *re*. The first such match is assigned to the first named *variable*, the second is assigned to the second named *variable*, and so forth. Blank variable names can be used to skip particular subexpressions. The return value is 1 if the regular expression matched. Otherwise, no variable assignments were made, and 0 is returned.

42a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁40b 44a▷

```
nerr_op(cs_register_function(cs, (char *)"re_extract", 3, cs_re_extract));
```

42b ⟨*tmpl.c* 17b⟩+≡ ◁41 44b▷

```
NEOERR *cs_re_extract(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  regex_t re;
  int ret;
```

42b ⟨*tmpl.c* 17b⟩+≡ ◁41 44b▷

```
char *vars = NULL, *re_s = NULL, *in_s = NULL;
char *s;
NEOERR *nerr = STATUS_OK;
int nmatch, matchno;
regmatch_t *matches;
gboolean matched;

nerr = cs_arg_parse(cs, args, "sss", &vars, &re_s, &in_s);
if(nerr != STATUS_OK) {
  if(vars)
    free(vars);
  if(re_s)
    free(re_s);
  if(in_s)
    free(in_s);
  return nerr;
}
for(nmatch = 0, s = vars; s && *s; nmatch++) {
  s = strchr(s, ',');
  if(s)
    *s++ = 0;
}
if(nmatch) {
  matches = calloc(sizeof(*matches), nmatch + 1);
  if(!matches) {
    free(vars);
    free(re_s);
    free(in_s);
    return nerr_raise_msg_errno("No memory for subst");
  }
} else
  matches = NULL;
cgi_url_unescape(re_s);
ret = regcomp(&re, re_s, REG_EXTENDED);
if(ret) {
  if(matches)
    free(matches);
  nerr = nerr_pass(regerr_nerr(re_s, &re, ret));
  free(vars);
  free(re_s);
  return nerr;
}
free(re_s);
matched = !regexec(&re, in_s, nmatch ? nmatch + 1 : 0, matches, 0);
if(matched) {
  for(s = vars, matchno = 1; matchno <= nmatch; matchno++, s += strlen(s) + 1) {
    if(*s) {
      int so = matches[matchno].rm_so, eo = matches[matchno].rm_eo;

      if(so == -1 || eo <= so)
        nerr = nerr_pass(hdf_set_value(cs->hdf, s, ""));
      else {
        char c = in_s[eo];

        in_s[eo] = 0;
        nerr = nerr_pass(hdf_set_value(cs->hdf, s, in_s + so));
        in_s[eo] = c;
      }
    }
    if(nerr != STATUS_OK) {
      free(vars);
      free(in_s);
      if(matches)
        free(matches);
```

42b ⟨*tmpl.c* 17b⟩+≡ ◁41 44b▷

```
        regfree(&re);
        return nerr;
      }
    }
  }
  free(vars);
  free(in_s);
  if(matches)
    free(matches);
  regfree(&re);
  result->op_type = CS_TYPE_NUM;
  result->s = NULL;
  result->n = !!matched;
  return STATUS_OK;
}
```

grep(*file*, *string*) Return the first line in *file* on the server matching POSIX extended regular expression *string*. If no lines match, a blank string is returned. The file name may not contain the vertical bar character. The primary intended purpose for this function is to look up password file entries in alternate password files.

44a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁42a 45a▷

```
nerr_op(cs_register_function(cs, (char *)"grep", 2, cs_grep));
```

44b ⟨*tmpl.c* 17b⟩+≡ ◁42b 45b▷

```
NEOERR *cs_grep(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  char *pat = NULL, *fname = NULL;
  FILE *f = NULL;
  GString *res = NULL;
  regex_t re;
  int ret;
  NEOERR *nerr;

  nerr = cs_arg_parse(cs, args, "ss", &fname, &pat);
  if(nerr != STATUS_OK) {
    if(fname)
      free(fname);
    if(pat)
      free(pat);
    return nerr;
  }
  f = fopen(fname, "r");
  if(!f) {
    nerr = nerr_raise_errno(NERR_IO, "%s", fname);
    free(fname);
    free(pat);
    return nerr;
  }
  free(fname);
  ret = regcomp(&re, pat, REG_EXTENDED | REG_NOSUB);
  if(ret) {
    fclose(f);
    nerr = nerr_pass(regerr_nerr(pat, &re, ret));
    free(pat);
    return nerr;
  }
  free(pat);
  result->op_type = CS_TYPE_STRING;
```

44b ⟨*tmpl.c* 17b⟩+≡ ◁42b 45b▷

```
  result->n = 0;
  result->alloc = 1;
  while (g_string_fgets(&res, 0, f)) {
    while (res->len > 0 && (res->str[res->len - 1] == '\n' ||
                            res->str[res->len - 1] == '\r'))
      res->len--;
    if (!regexec(&re, res->str, 0, NULL, 0)) {
      result->s = g_string_free(res, FALSE);
      fclose(f);
      return STATUS_OK;
    }
  }
  regfree(&re);
  g_string_free(res, TRUE);
  result->s = strdup("");
  return STATUS_OK;
}
```

eval_tmpl(*template*) Evaluate *template* as a template. Unlike `lvar` and `evar`, the results can be assigned to a variable for further processing. Note that the simple implementation of this function always crashes. The HDF is probably being shared, so it is duplicated and then destroyed. This means that any side effects are lost. This function also requires the use of the generic function interface; without it only the global HDF may be used by the template.

45a ⟨*Initialize templates* 22a⟩+≡ (18a) ◁44a

```
nerr_op(cs_register_function(cs, (char *)"eval_tmpl", 1, cs_eval_tmpl));
```

45b ⟨*tmpl.c* 17b⟩+≡ ◁44b

```
NEOERR *cs_eval_tmpl(CSPARSE *cs, CS_FUNCTION *csf, CSARG *args, CSARG *result)
{
  GString *out_str;
  NEOERR *nerr;
  char *in = NULL;

  nerr = cs_arg_parse(cs, args, "s", &in);
  if (nerr != STATUS_OK) {
    if (in)
      free(in);
    return nerr;
  }
  out_str = g_string_new("");
  nerr = nerr_pass(tmpl_string(in, cs->hdf, out_str, TRUE));
  free(in);
  if (nerr != STATUS_OK) {
    g_string_free(out_str, TRUE);
    return nerr;
  }
  result->op_type = CS_TYPE_STRING;
  result->n = 0;
  result->s = g_string_free(out_str, FALSE);
  result->alloc = 1;
  return nerr;
}
```

A simple program is provided to strip ClearSilver from templates, supporting these functions.

45c ⟨*(build.nw) C Executables* 45c⟩≡

```
display_template \
```

46a ⟨*display_template.c* 46a⟩≡

```
⟨(build.nw) Common C Includes 1⟩
#include "cproto.h"

void help(void)
{
}

const char *config_root = ".";
const char *config_path = "display_template.conf";

int main(int argc, const char **argv)
{
  ⟨Common Mainline Variables 5f⟩
  const char *tmpl;

  if(argc <= 1)
    return 0;
  argc--;
  tmpl = *++argv;
  ⟨Read configuration 8g⟩
  if(nerr == STATUS_OK) {
    GString *cwd = g_string_new("");

    nerr = nerr_pass(getcwd_gs(cwd));
    nerr_op(hdf_set_value(local_config, "hdf.loadpaths.cwd", cwd->str));
    g_string_free(cwd, TRUE);
  }
  nerr_op(hdf_set_value(local_config, "Config.VarEscapeMode", "html"));
  nerr_op(tmpl_file(tmpl, local_config, stdout, FALSE));
  ret = 0;
  if(nerr != STATUS_OK) {
    nerr_log_error(nerr);
    ret = 1;
  }
  return ret;
}
```

46b ⟨*cs_supt.c* 5e⟩+≡ ◁40a

```
NEOERR *getcwd_gs(GString *gs)
{
  if(!gs->allocated_len)
    g_string_set_size(gs, 32);
  while(1) {
    if(getcwd(gs->str, gs->allocated_len))
      return STATUS_OK;
    if(errno == EAGAIN || errno == EINTR)
      continue;
    if(errno != ERANGE)
      return nerr_raise_msg_errno("getting current directory");
    g_string_set_size(gs, gs->allocated_len * 2);
  }
}
```

5 HTML Templates

All of the HTML templates have to start with the wordy HTML document type header. Since heavy use of ClearSilver macros would invalidate most HTML files anyway, this is placed in a standard header file. It's also provided as a code chunk to make creation of plain HTML easier.

47a ⟨*CS files* 47a⟩≡ (19c)

```
html_prefix.cs \
```

47b ⟨*html_prefix.cs* 47b⟩≡

```
<?cs ⟨(build.nw) Common NoWeb Warning 3c⟩
?>⟨Standard HTML Prefix 47c⟩
```

47c ⟨*Standard HTML Prefix* 47c⟩≡ (47b)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
```

In addition to the diagrams, embedded HTML should have some sort of preview picture. The perl html2ps utility can be used for this, if file and button fields are converted to plain input fields. Also, ClearSilver escape sequences cause html2ps to produce bad output, so the `display_template` program (⟨*display_template.c* 46a⟩) is used to run the template through ClearSilver with any required extensions. Since these pages can get quite large, they can be split using `<div>` tags. The figures can then either be the entire HTML, just the HTML before the first `<div>`, or the HTML contained in a single `<div>`. Nested `<div>`s are not supported. The preview looks very little like the page would look in a browser. Rather than supporting arbitrary `<div>` names by extracting them from the NoWeb file, the only supported names are div1, div2, ... div9.

47d ⟨*(build.nw) makefile.vars* 2a⟩+≡ ◁20d

```
HTML_FILES=$(shell $(NOROOTS) $(NOWEB) | sed -n '/\.html>>/{s/<<//;s/>>//;p;}')
HTML2PS=html2ps -D -s 0.4 -S "@html2ps { prefilled:1; }" -o
HTML2PS_SEDFORM=s/\(<input[^>]*\) type="[fFbBsS][^""]*"/\1/g;
```

47e ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁20h 48b▷

```
%.eps: %.html display_template
        ./display_template $< | sed '$(HTML2PS_SEDFORM)' | $(HTML2PS) $@

%-main.eps: %.html display_template
        ./display_template $< | sed -n '$(HTML2PS_SEDFORM)/<div/,/<\/div>/!p' | \
                $(HTML2PS) $@

define eps_div
%-div$(1).eps: %.html display_template
        ./display_template $$< | sed -n -e '$(HTML2PS_SEDFORM)' \
                        -e '/<body/{p;:a /<div/bb;n;ba;:b;};' \
                        -e '/div.*id="div$(1)"/,/<\/div>/p;' \
                        -e '/<div/,/<\/div>/!p' | \
           $(HTML2PS) $$@
endef

$(foreach i,1 2 3 4 5 6 7 8 9,$(eval $(call eps_div,$i)))
```

Another way to reinclude the HTML would be to directly include the full HTML. This really only makes sense for text-only HTML; forms are dangerous to include in HTML and are difficult to convert to LaTeX/PDF. For PDF and TeX4ht, this can be done using the `typehtml` package[7]. This uses the `\htmlinput` command to import the HTML. Dependencies on the HTML need to be added manually to ⟨*(build.nw) makefile.rules* 10c⟩. In order to support templates as well as raw HTML, though, a rule is provided to support tacking `.nocs` onto the name to run the file through `display_template` first.

48a ⟨*(build.nw) preamble.tex* 48a⟩≡

```
\usepackage[hyperref]{typehtml}
```

48b ⟨*(build.nw) makefile.rules* 10c⟩+≡ ◁47e

```
%.nocs: % display_template
	./display_template $< >$@
```

48c ⟨*(build.nw) Clean temporary files* 15d⟩+≡ ◁15d

```
rm -f *.nocs
```

For l2h, the tag needs to be saved, and the post-processor needs to reinsert the HTML.

48d ⟨*(build.nw) preamble.l2h* 48d⟩≡

```
% l2h macro htmlinput 1 <htmlinput #1>
```

48e ⟨*(build.nw) Post-process HTML after weave* 48e⟩≡

```
(
  pl="cat"
  for hf in `sed -n '/^\\\\htmlinput[{]\\([^}]*\\)[}].*/\\1/;T;p' "$noweb"`; do
    pl="$pl|sed '/<htmlinput $hf>//;T;a\\`
       echo
       sed -n '/<body/{
           :a
            n
            /<\\/body>/b
            s/\\\\/\\\\\\\\/g
            s/'\\"/'\\"\\\\'\\"'\\"/g
            s/\$/\\\\/
            p
            ba
         }' $hf`
            '"
  done
  eval "$pl"
) | \
```

6 Usage

To use the facilities provided by this module, add a dependency to it. The only things provided by this package which may be of interest outside of the build system are the `g_string_fgets` routines and the `runit` utility.

The `g_string_fgets` routines include their own header file, `g_string_fgets.h`, and can be used as an object file or from the `libglib-supt.a` library. The header file does not include the prototypes for

[7] http://www.ctan.org/tex-archive/macros/latex/contrib/carlisle/

the provided functions, so `cproto` or something similar should be used to extract those first, like the build system normally does automatically. The following functions are provided:

```
char *g_string_fgets(GString **_buf,
                     guint offset,
                     FILE *f)
```

Retrieve a line into a `GString` buffer starting at the given `offset`, which may be initialized to NULL. The buffer will never be returned as NULL. The return value is a pointer to the string's buffer if there were no errors; use `ferror` and `feof` to determine the exact cause for a NULL return. Reading will terminate at the next newline. The string length will include the last newline, or the last character read, or the first ASCII NUL, whichever comes first.

```
char *g_string_fgets_bin(GString **_buf,
                         guint offset,
                         FILE *f)
```

Retrieve a line into a `GString` buffer starting at the given `offset`. Operation is identival to `g_string_fgets`, except that ASCII NUL characters in the input are not treated specially. This means that the result of `strlen` may differ from the string's `len` parameter.

```
char *g_string_gets_generic(GString **_buf,
                            guint offset,
                            gets_fp my_gets,
                            void *f,
                            gboolean allow_zero)
```

```
typedef char *(*gets_fp)(char *buf, int len, void *f);
```

Retrieve a line into a `GString` buffer starting at the given `offset`. The `my_gets` parameter replaces the standard `fgets` routine for reading the line, and takes the `f` parameter in place of the file. The `allow_zero` parameter distinguishes between the above two methods for determining the return length. This function may be used for reading something other than a standard `FILE`.

The `runit` utility simply takes the program to run, and its arguments, as its own argument. A file in `/tmp` with the same base name as the program to run, except with a `stop_` prefix, may be touched to temporarily delay execution of the program until the file is removed.

For programs which use the build facility, a number of other routines and definitions are also provided. For dealing with errors, the following macros and code chunks are provided.

- ⟨*Common Mainline Variables* 5f⟩ defines `nerr` and `ret` for tracking return codes. `nerr` is a `NEOERR` pointer and `ret` is an `int`.
- ⟨*Initialize logging* 5g⟩ is intended to be used in `main`. It stores the program name and initializes the ClearSilver error processing.
- `GENERIC_ERR` is a preinitialized generic `NEOERR` type for situations where the type does not matter. It is used with all generic message routines.
-

```
void die_if_err(NEOERR *err)
```

Prints `err` to standard error and exits the program if `err` is not `STATUS_OK`.

- ```
  void die_msg(const char *msg)
  ```

  Prints `msg` in the form of a ClearSilver error to standard error and exits the program.

- ```
  void die_errno(const char *msg)
  ```

  Prints `msg` in the form of a ClearSilver error with system error information and exits the program.

- ```
  void die_errno_local(const char *msg,
                       int err)
  ```

  Prints `msg` in the form of a ClearSilver error with system error information using `err` instead of `errno` to standard error and exits the program.

- ```
  NEOERR *nerr_raise_msg(const char *msg)
  ```

  Generates an error with a fixed message.

- ```
  NEOERR *nerr_raise_msg_errno(const char *msg)
  ```

  Generates an error with a fixed message and system error information.

- ```
  NEOERR *nerr_raise_errno_local(const char *msg,
                                 int err)
  ```

  Generates an error with a fixed message and system error information using `err` instead of `errno`.

- ```
  void nerr_op(NEOERR *op)
  ```

  Only executes `op` if the local variable `nerr` is `STATUS_OK`, and if so, it updates `nerr` to `op`'s return code.

- ```
  gboolean nerr_op_ok(NEOERR *op)
  ```

  Only executes `op` if the local variable `nerr` is `STATUS_OK`, and then also updates `nerr` to `op`'s return code. If and only if `nerr` is `STATUS_OK` still after this, `TRUE` is returned.

- ⟨*Convert* `nerr` *to* `err_str` 6b⟩ creates a `STRING` called `err_str`, which contains an expanded string version of the error stored in the local variable `nerr`.

-

```
NEOERR *gerr_to_nerr(const GError *gerr)
```

  Converts a GLib error to a ClearSilver error. Note that this macro, unlike the others, evaluates `err` multiple times.

For dealing with configuration files in ClearSilver HDF format, the following macros, functions, variables, and code chunks are provided. In addition, configuration files which use the multi-line literal here-doc style syntax starting with a double-less-than may use `EOV` as their termination literal, and have the termination literal forced to the left margin even if the code chunk is indented.

- ⟨*Read configuration* 8g⟩ reads the primary configuration file, and processes keyword-value pairs and file references on the command line. It already includes the log initialization above. On failures, it calls `void help(void)`, which it assumes exits the program.
- ⟨*Help Function* 13b⟩ is a generic help function (`void help(void)`) which displays the command-line syntax, program description, and configuration parameters. In order to do so, the program description for a program called `<program>` would need to be a single-line chunk named ⟨`<program>` *Description* (imported)⟩, and its configuration parameters would need to be in a chunk named ⟨`<program>` *Configuration* (imported)⟩. The format of the configuration parameters should be one block of comments per configuration parameter, with the description above the parameter. The description comments each have a space before the text, and the parameter is an assignment to the default value, commented out, with no space after the comment character or surrounding the equals sign. Each configuration parameter should be followed by a blank line to separate it from the next.
- ⟨*ClearSilver Configuration Documentation* 9c⟩ may be prepended to a configuration file to provide basic documentation of the configuration file format. An additional blurb may be necessary at the start to indicate the default file name and environment overrides.
- `const char *config_root` is a global variable which must be defined in a program using these facilities. It is the default search path for configuration files, and may be overridden with the environment variable `CONFIG_ROOT`. A default path derived from the project name can be inserted using ⟨*Default* `config_root` 8d⟩.
- `const char *config_path` is a global variable which must be defined in a program using these facilities. It is the default configuration file name, and may be overridden with the environment variable `CONFIG_FILE`. A default name derived from the project name can be inserted using ⟨*Default* `config_path` 8e⟩.
- `HDF *local_config` is a global variable which contains the primary configuration for a program. This is read from the configuration file under a section named the same as the executable.
-

```
const char *getconf(const char *name,
                    const char *def)
```

  Returns the value of a configuration variable in `local_config`, or `def` if not found or the parameter has no value (i.e. it is only the parent of another configuration value).

-

```
gint64 getconf_int(const char *name,
                   gint64 def)
```

Returns the value of an integer configuration variable in `local_config`, or `def` if not found or the parameter has no value (i.e. it is only the parent of another configuration value).

- ```
  HDF *getconf_first(const char *name)
  ```

  Returns the first child of the given configuration parameter.

- ```
  HDF *getconf_first_sorted(const char *name)
  ```

  Returns the first child of the given configuration parameter after sorting the children alphanumerically.

- ```
  gint64 hdf_get_int64_value(HDF *hdf,
                             const char *name,
                             gint64 def)
  ```

  Returns the value of an integer variable in `hdf`, or `default` if not found or the parameter has no value (i.e. it is only the parent of another configuration value). It is the 64-bit integer equivalent of `hdf_get_int_value`, except that it kills the program, rather than using the default value, if the parameter exists, but is not an integer.

- ```
  gint64 getconf_hier_int(const char *parent,
                          HDF *obj,
                          const char *name,
                          gint64 def)
  ```

  Returns the same value as `hdf_get_int64_value(obj, name, default)`, except that the message it displays on failure adds `parent` to the failed node name displayed in the message.

- ```
  NEOERR *hdf_set_int64_value(HDF *hdf,
                              const char *name,
                              gint64 val)
  ```

  Sets the named parameter to the string form of the integral value. This is the 64-bit equivalent of `hdf_set_int_value`.

- ```
  HDF *hdf_first_child(HDF *hdf,
                       const char *name)
  ```

  Returns the first child of the named variable in `hdf`.

- ```
  HDF *hdf_first_child_sort(HDF *hdf,
                            const char *name)
  ```

  Returns the first child of the named variable in `hdf` after sorting the children alphanumerically.
  ```

-
```
void env_override(const char *env,
                  const char *&var)
```
Sets `var` to the value of environment variable `env` if a variable by that name exists.

-
```
void prepend_dir_if_relative(const char *def_dir,
                             const char *&fn)
```
Sets `fn` to a heap-allocated string consisting of `dir`, followed by the old contents of `fn`, separated by a slash, if `fn` does not begin with a slash.

-
```
int comp_hdf_name(const void *_a,
                  const void *_b)
```
A comparison function which may be used with `hdf_sort_obj` to achieve an alphanumeric sort with numbers sorted first, and numeric prefixes sorted as numbers even if followed by alphabetic characters.

In addition, support is provided for using gperf to convert string values to numeric enumeration constants. The strings to hash are specified in chunk names like *prefix*-gperf-*suffix*. Strings are separated by newlines or single spaces; commas and spaces are not allowed in the strings. The *prefix* is the base name of the output files; the C code is generated in *prefix*.c.gperf, and the enumerations are built in *prefix*.h.gperf. These should be included directly in the C and header files that use them. Each unique *suffix* generates a different hash function and enumeration type. The enumeration type is named *suffix*_t, and the hash lookup function is named *suffix*_id. The *suffix* must be a valid C identifier. The enumeration identifiers are called *SUFFIX*_*STRING*, where *SUFFIX* is *suffix* converted to upper-case, and *STRING* is the string with all alphabetic characters in upper-case and all non-alphanumeric characters converted to underscores. This means that strings differing only in case or special characters cannot coexist in the same hash table. One additional enumeration literal, UNKNOWN_*SUFFIX*, is given the value zero, which is returned if the string is not found. The only option which may be set is case-sensitivity. A case-insensitive hash can be generated by using -gperf-nc- instead of -gperf-; the nc- part is not considered to be part of the *suffix*.

The templates use the standard ClearSilver syntax (see your installed ClearSilver package, or http://www.clearsilver.net/docs/man_templates.hdf).

To support templates, the following functions, variables, and code chunks are defined. In addition, files added to the ⟨*CS Files* (imported)⟩ chunk are plain files which are also post-processed to remove the final newline. This gets around a NoWeb limitation wherein any code chunk has a terminating newline, and allows for pure macro libraries which do not affect output.

- `tmpl_dir` is a global variable which is a subdirectory of `config_root` which is the default search path for template files. Its default value is `tmpl`, and it can be overridden by the `tmpl_dir` configuration parameter, as described in ⟨*Template Parameter Configuration* 18d⟩, a chunk which should be included in your configuration files. To install the templates in the default location, the ⟨*Install ClearSilver templates* 20b⟩ chunk should be added to ⟨*(build.nw) Install other files* 11c⟩.

-
```
NEOERR *tmpl_string(const char *tmpl,
                    HDF *parms,
                    GString *out,
                    gboolean raw)
```

Evaluates the template `tmpl` and places the results in `out`. Variable values are retrieved from `parms`. The `raw` option may be `TRUE` to make the default conversion option for the template raw. Note that setting this will cause `parms` to be temporarily modified, in addition to any modifications the template may cause.

-

```
NEOERR *tmpl_string_to_file(const char *tmpl,
                            HDF *parms,
                            FILE *out,
                            gboolean raw)
```

Evaluates the template `tmpl` and places the results in `out`. Variable values are retrieved from `parms`. The `raw` option may be `TRUE` to make the default conversion option for the template raw. Note that setting this will cause `parms` to be temporarily modified, in addition to any modifications the template may cause.

-

```
NEOERR *tmpl_string_to(const char *tmpl,
                       HDF *parms,
                       CSOUTFUNC outf,
                       void *outp,
                       gboolean raw)
```

A generic template string evaluator, which sends its output via `outf`, which is passed `outp`. Variable values are retrieved from `parms`. The `raw` option may be `TRUE` to make the default conversion option for the template raw. Note that setting this will cause `parms` to be temporarily modified, in addition to any modifications the template may cause.

-

```
NEOERR *tmpl_file(const char *tmpl,
                  HDF *parms,
                  FILE *out,
                  gboolean raw)
```

Evaluates the template file `tmpl` and places the results in `out`. Variable values are retrieved from `parms`. The `raw` option may be `TRUE` to make the default conversion option for the template raw. Note that setting this will cause `parms` to be temporarily modified, in addition to any modifications the template may cause.

-

```
NEOERR *tmpl_file_to_string(const char *tmpl,
                            HDF *parms,
                            GString *out,
                            gboolean raw)
```

Evaluates the template `tmpl` and places the results in `out`. Variable values are retrieved from `parms`. The `raw` option may be `TRUE` to make the default conversion option for the template raw. Note that setting this will cause `parms` to be temporarily modified, in addition to any modifications the template may cause.

-

Format	Example
%Y-%m-%d %T	2008-06-13 18:07:10
%Y-%m-%d %R	2008-06-13 18:07
%Y-%m-%d	2008-06-13
%m/%d/%Y %T	6/13/2008 18:07:10
%m/%d/%Y %R	6/13/2008 18:07
%m/%d/%Y	6/13/2008
%d-%m-%Y %T	13-06-2008 18:07:10
%d-%m-%Y %R	13-06-2008 18:07
%d-%m-%Y	13-06-2008
%T	18:07:10
%R	18:07
%a %T	Friday 18:07:10
%a %R	Friday 18:07
%a	Friday
%b %Y	June 2008
%b	June
%c	Fri Jun 13 18:07:10 2008
%x	06/13/08

Table 2: Default formats for `parse_date`

```
NEOERR *tmpl_file_to(const char *tmpl,
                     HDF *parms,
                     CSOUTFUNC outf,
                     void *outp,
                     gboolean raw)
```

A generic template file evaluator, which sends its output via `outf`, which is passed `outp`. Variable values are retrieved from `parms`. The `raw` option may be `TRUE` to make the default conversion option for the template raw. Note that setting this will cause `parms` to be temporarily modified, in addition to any modifications the template may cause.

- `display_template` is a program which displays a ClearSilver template using the above functions wich a simplified search path of the current directory.

All of the above template evaluators also add a few ClearSilver functions which may be used by the template:

parse_date(*formats*, *date*) Parse an absolute date in server's current locale using `strptime`(3) or a relative date using a custom format and return as seconds since UNIX epoch. The *formats* string consists of zero or more formats separated by vertical bars (`|`). The formats in the "Default formats for `parse_date`" table are appended to the given formats, in order (examples are for Friday, June 13, 2008, 18:07:10, with adjustments described below). The first format which matches the *date* will be used.

Since `strptime`(3) does not support time zones, the time zone is always the server's local time zone. This can be changed for subsequent formats by using the format `TZ=`*zone*. Note that the format of the *zone* string is operating system-dependent. See `tzset`(3) for details. If any of hours, minutes, or seconds are not present, they are set to zero. If the month is specified, but not the day, the first of the month is used. If only the year is specified, January 1 is used. These formats also support some relative times. If the year is unspecified, it is the current year, unless the day of year is specified, either by month or day of year, and that day is on or after today, in which case it is last year. If only the weekday is specified, then it is the last such weekday, excluding today. In the previous examples, the

%a format only works between Saturday, June 14, and Friday, June 20, and the %b format only works between July 1, 2008, and June 30, 2009.

For specifically relative dates, the date must consist of a star (*), implying the current date, optionally followed by an offset. The offset is an optional sign (+ or -; - is default), followed by an optional count (1 is the default), followed by an optional case-insensitve interval length indicator for seconds (S), minutes (’), hours (H), days (D) (the default), months (M), or years (Y). For a valid offset, one or the other of the count or the interval length indicator must be present. All parts of the date finer than the specified interval are set to their lowest value. For example, if the interval is in months, then the day is set to the first of the month, and the time is set to midnight.

Finally, a blank input returns 0, always.

format_date(*format*, *timestamp*) Format date using `strftime`(3), with optional *format*. A blank *format* is replaced by %c. The *timestamp* is seconds since UNIX epoch.

url_unescape(*string*) URL-decode *string* (URL-encoding is provided by the CGI library as `url_escape(`*`string`*`)`)

json_escape(*variable-name*) Escape *variable* and its children using JavaScript Object Notation (JSON). See `http://www.json.org` for the complete format specification. Nonexistent variables and variables without values or children are output as `null`. Variables with values are output as strings by default, but may be output as numbers by adding the `num` attribute, and may be output as booleans by adding the `bool` attribute; booleans with no value, blank values, or the value `0` are output as `false`. Variables with children are output as objects by default, but may be output as arrays (i.e., children names are sorted, but not output) by adding the `array` attribute; arrays with no value are output as an empty array, but there is no way to output an empty object (`{}`). String values must be UTF-8 encoded. If any variables have raw values which may conflict with UTF-8, they should be double-encoded using e.g. URL-encoding to ensure that their value is converted to ASCII first.

json_unescape(*variable-name*, *value*) Set *variable* to the result of parsing *value* using JavaScript Object Notation (JSON). See `http://www.json.org` for the complete format specification. Arrays will have their `array` attribute set and their children will receive numeric sequence names. Booleans will be set to `1` for `true` and blank for `false` and have their `bool` attribute set. Numbers will have their `num` attribute set. Unicode escapes in the string will be encoded as per UTF-8.

to_hex(*bin*) Convert a URL-encoded binary blob to a string of lower-case hexadecimal digits. This is mostly for displaying checksums. The blob must be URL-encoded so that zeroes are handled properly in the string.

from_hex(*hex*) Convert a string of hexadecimal digits to a URL-encoded binary blob. This is mostly for converting checksums from their usual format to that expected by these utilities. The return must be URL-encoded to prevent issues with zeroes in string returns.

to_lower(*string*) Convert *string* to lower-case in the server’s current locale. This is done a byte at a time using the `ctype.h` function. This means that the string will probably be mangled in UTF-8 locales or if the input locale does not match the server’s locale.

to_upper(*string*) Convert *string* to upper-case. The comments for `to_lower()` apply here as well.

pw_uid(*id*) Return the password file entry for the given user ID on the local system. Since this uses the system function, it may not be a real file entry, but it is returned in the format of one.

pw_nam(*name*) Return the password file entry for the given user name on the local system. Since this uses the system function, it may not be a real file entry, but it is returned in the format of one.

re_match(*re*, *string*) Return integer flag 1 if URL-encoded POSIX extended regular expression *re* matches (non-URL-encoded) *string*.

re_subst(*re*, *subst*, *string*) Substitute occurrences of URL-encoded POSIX extended regular expression *re* with URL-encoded *subst* string in (non-URL-encoded) *string*. In *subst*, backslash escapes are supported:

- \\ means backslash
- \0 means the entire matching string
- *n*, where *n* is a decimal number, means the n^{th} parenthesized subexpression
- \, means nothing (so you can separate *n* from literal decimal digits)
- \g also means nothing, but has the side effect that all occurrences of the regular expression are substituted. If this is not present, only the first occurrence is substituted.

The behavior of any other character following a backslash, or a backslash at the end of a string, is undefined.

re_extract(*variable* [+ ',' + *variable* ...], *re*, *string*) Extract substrings of (non-URL-encoded) *string* matching parenthesized subexpressions in URL-encoded POSIX extended regular expression *re*. The first such match is assigned to the first named *variable*, the second is assigned to the second named *variable*, and so forth. Blank variable names can be used to skip particular subexpressions. The return value is 1 if the regular expression matched. Otherwise, no variable assignments were made, and 0 is returned.

grep(*file*, *string*) Return the first line in *file* on the server matching POSIX extended regular expression *string*. If no lines match, a blank string is returned. The file name may not contain the vertical bar character. The primary intended purpose for this function is to look up password file entries in alternate password files.

eval_tmpl(*template*) Evaluate *template* as a template. Unlike `lvar` and `evar`, the results can be assigned to a variable for further processing.

For HTML templates, `html_prefix.cs` can be included in any file to remove the need to declare the document type as xhtml. Also, the HTML template can be run through `display_template` to create a preview picture in the document using `\includegraphics` with the base name of the HTML file. If the HTML file is too large for a single preview, it can be split using `<div>` tags with an id of `div1`, `div2`, and so forth. The portion of the HTML outside of the `div`s is named the same as the base name of the HTML file, but with `-main` tacked on, and the divisions are similarly named with `-div1`, etc. tacked on. If a graphical preview is inappropriate, and the HTML is simple enough, the HTML can be included by converting it to raw LaTeX with the `\htmlinput` directive. The dependency this adds to the PDF on the HTML output must be added to ⟨*(build.nw) makefile.rules* 10c⟩ manually.

A few miscellaneous items are also defined.

- `debug` is a global variable set to the value of the `debug` integer parameter. It should not normally be documented in the default configuration file.
-

```
NEOERR *getcwd_gs(GString *gs)
```

Retrieves the full current working directory into a dynamic string.

-

```
GString *g_string_append_json(GString *buf,
                              HDF *hdf)
```

Appends the JSON enconding of `hdf` onto the string `buf`. See the ClearSilver function `json_escape` for details.

-

```
NEOERR *json_from_string(const char *s,
                         HDF *hdf)
```

Converts the JSON-encoded string `s` into values for the HDF hierarchy `hdf`. See the ClearSilver function `json_unescape` for details.

-

```
NEOERR *embedded_json_from_string(const char *s,
                                  HDF *hdf,
                                  const char **ep)
```

Converts the JSON-encoded string `s` into values for the HDF hierarchy `hdf`. If `ep` is not NULL, a pointer to the text after the parsed value is returned. Otherwise, the text after the parsed value is simply ignored. See the ClearSilver function `json_unescape` for details.

-

```
GString *append_unicode_quoted(GString *buf,
                               const char *s)
```

Converts the string `s` into a JSON-encoded string value, including quote delimiters.

7 Code Dependencies

This application depends on several tools and libraries not included in this document. Required build tools include the build document provided along with this document, NoWeb, GNU make, cproto, and obviously a working C compiler. If the string-to-enum support is used, gperf is needed as well. Required libraries (with headers, so "development" versions) include GLib and ClearSilver. See the first mention of each of these in the document for where to get it (if not from your OS vendor) and what version is required.

The data types provided by the used libraries need to be highlighted, as well.

58 ⟨*(build.nw) Known Data Types* 2g⟩+≡ ◁28a

```
% GLib
gboolean,gint8,guint8,gint16,guint16,gint32,guint32,gint64,guint64,%
gchar,guchar,gint,guint,glong,gulong,GString,GStringChunk,GArray,GByteArray,%
GPtrArray,gconstpointer,gpointer,GMemChunk,GDir,GError,%
% ClearSilver
HDF,NEOERR,NERR_TYPE,CGI,CSPARSE,STRING,CSOUTFUNC,%
```

8 Code Index