1. String handling.

2. First, we will need some stuff from other files.

```
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <assert.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>     /* for access */
#endif
#include <time.h>     /* for struct tm co */
#include "mpstrings.h"     /* internal header */
```

3. Then there is some stuff we need to prepare ourselves.

```
⟨ mpstrings.h 3 ⟩ ≡
#ifndef MPSTRINGS_H
#define MPSTRINGS_H 1
#include "mplib.h"
#include "mplibps.h"     /* external header */
#include "mplibsvg.h"     /* external header */
#include "mpmp.h"     /* internal header */
#include "mppsout.h"     /* internal header */
#include "mpsvgout.h"     /* internal header */
#include "mpmath.h"     /* internal header */
  ⟨ Definitions 4 ⟩;
#endif
```

4. Here are the functions needed for the avl construction.

```
⟨ Definitions 4 ⟩ ≡
  void *copy_strings_entry(const void *p);
```

See also sections 8, 13, 14, 18, 19, 21, 22, 23, 26, 27, 29, 31, and 33.

This code is used in section 3.

5. An earlier version of this function used *strncmp*, but that produces wrong results in some cases.

```
#define STRCMP_RESULT(a)  ((a) < 0 ? -1 : ((a) > 0 ? 1 : 0))
  static int comp_strings_entry(void *p, const void *pa, const void *pb)
  {
    const mp_lstring *a = (const mp_lstring *) pa;
    const mp_lstring *b = (const mp_lstring *) pb;
    size_t l;
    unsigned char *s, *t;
    (void) p;
    s = a->str;
    t = b->str;
    l = (a->len ≤ b->len ? a->len : b->len);
    while (l-- > 0) {
      if (*s ≠ *t) return STRCMP_RESULT(*s - *t);
      s++;
      t++;
    }
    return STRCMP_RESULT((int)(a->len - b->len));
  }
  void *copy_strings_entry(const void *p)
  {
    mp_string ff;
    const mp_lstring *fp;
    fp = (const mp_lstring *) p;
    ff = malloc(sizeof (mp_lstring));
    if (ff ≡ Λ) return Λ;
    ff->str = malloc(fp->len + 1);
    if (ff->str ≡ Λ) {
      return Λ;
    }
    memcpy((char *) ff->str, (char *) fp->str, fp->len + 1);
    ff->len = fp->len;
    ff->refs = 0;
    return ff;
  }
  static void *delete_strings_entry(void *p)
  {
    mp_string ff = (mp_string)p;
    mp_xfree(ff->str);
    mp_xfree(ff);
    return Λ;
  }
```

6. Actually creating strings is done by *make_string*, but in order to do so it needs a way to create a new, empty string structure.

7.
```
static mp_string new_strings_entry(MP mp)
{
  mp_string ff;
  ff = mp_xmalloc(mp, 1, sizeof (mp_lstring));
  ff→str = Λ;
  ff→len = 0;
  ff→refs = 0;
  return ff;
}
```

8. Some even more low-level functions are these:

```
⟨ Definitions 4 ⟩ +≡
  extern int mp_xstrcmp(const char *a, const char *b);
  extern char *mp_xstrdup(MP mp, const char *s);
  extern char *mp_xstrldup(MP mp, const char *s, size_t l);
  extern char *mp_strdup(const char *p);
  extern char *mp_strldup(const char *p, size_t l);
```

9.
```
char *mp_strldup(const char *p, size_t l)
{
  char *r, *s;
  if (p ≡ Λ) return Λ;
  r = malloc((size_t)(l * sizeof(char) + 1));
  if (r ≡ Λ) return Λ;
  s = memcpy(r, p, (size_t)(l));
  *(s + l) = '\0';
  return s;
}
char *mp_strdup(const char *p)
{
  if (p ≡ Λ) return Λ;
  return mp_strldup(p, strlen(p));
}
```

10. **int** *mp_xstrcmp*(**const char** ∗*a*, **const char** ∗*b*)
```
{
  if (a ≡ Λ ∧ b ≡ Λ) return 0;
  if (a ≡ Λ) return −1;
  if (b ≡ Λ) return 1;
  return strcmp(a, b);
}
char ∗mp_xstrldup(MP mp, const char ∗s, size_t l)
{
  char ∗w;
  if (s ≡ Λ) return Λ;
  w = mp_strldup(s, l);
  if (w ≡ Λ) {
    mp_fputs("Out␣of␣memory!\n", mp→err_out);
    mp→history = mp_system_error_stop;
    mp_jump_out(mp);
  }
  return w;
}
char ∗mp_xstrdup(MP mp, const char ∗s)
{
  if (s ≡ Λ) return Λ;
  return mp_xstrldup(mp, s, strlen(s));
}
```

11. **void** *mp_initialize_strings*(MP *mp*)
```
{
  mp→strings = avl_create(comp_strings_entry, copy_strings_entry, delete_strings_entry, malloc, free, Λ);
  mp→cur_string = Λ;
  mp→cur_length = 0;
  mp→cur_string_size = 0;
}
```

12. **void** *mp_dealloc_strings*(MP *mp*)
```
{
  if (mp→strings ≠ Λ) avl_destroy(mp→strings);
  mp→strings = Λ;
  mp_xfree(mp→cur_string);
  mp→cur_string = Λ;
  mp→cur_length = 0;
  mp→cur_string_size = 0;
}
```

13. Here are the definitions:

⟨ Definitions 4 ⟩ +≡
```
extern void mp_initialize_strings(MP mp);
extern void mp_dealloc_strings(MP mp);
```

14. Most printing is done from **char** ∗s, but sometimes not. Here are functions that convert an internal string into a **char** ∗ for use by the printing routines, and vice versa.

```
⟨ Definitions 4 ⟩ +≡
  char *mp_str(MP mp, mp_string s);
  mp_string mp_rtsl(MP mp, const char *s, size_t l);
  mp_string mp_rts(MP mp, const char *s);
  mp_string mp_make_string(MP mp);
```

15.
```
  char *mp_str(MP mp, mp_string ss)
  {
    (void) mp;
    return (char *) ss->str;
  }
```

16.
```
  mp_string mp_rtsl(MP mp, const char *s, size_t l)
  {
    mp_string str, nstr;
    str = new_strings_entry(mp);
    str->str = (unsigned char *) mp_xstrldup(mp, s, l);
    str->len = l;
    nstr = (mp_string)avl_find(str, mp->strings);
    if (nstr ≡ Λ) {       /* not yet known */
      assert(avl_ins(str, mp->strings, avl_false) > 0);
      nstr = (mp_string)avl_find(str, mp->strings);
    }
    (void) delete_strings_entry(str);
    add_str_ref(nstr);
    return nstr;
  }
```

17.
```
  mp_string mp_rts(MP mp, const char *s)
  {
    return mp_rtsl(mp, s, strlen(s));
  }
```

18. Strings are created by appending character codes to *cur_string*. The *append_char* macro, defined here, does not check to see if the buffer overflows; this test is supposed to be made before *append_char* is used.

To test if there is room to append l more characters to *cur_string*, we shall write *str_room*(l), which tries to make sure there is enough room in the *cur_string*.

```
⟨ Definitions 4 ⟩ +≡
#define EXTRA_STRING 500
#define append_char(A) do
  {
    str_room(1);
    *(mp→cur_string + mp→cur_length) = (unsigned char)(A);
    mp→cur_length++;
  }
  while (0)
#define str_room(wsize) do
  {
    size_t nsize;
    if ((mp→cur_length + (size_t) wsize) > mp→cur_string_size) {
      nsize = mp→cur_string_size + mp→cur_string_size/5 + EXTRA_STRING;
      if (nsize < (size_t)(wsize)) {
        nsize = (size_t) wsize + EXTRA_STRING;
      }
      mp→cur_string = (unsigned char *) mp_xrealloc(mp, mp→cur_string, (unsigned)
          nsize, sizeof(unsigned char));
      memset(mp→cur_string + mp→cur_length, 0, (nsize - mp→cur_length));
      mp→cur_string_size = nsize;
    }
  }
  while (0)
```

19. At the very start of the metapost run and each time after *make_string* has stored a new string in the avl tree, the *cur_string* variable has to be prepared so that it will be ready to start creating a new string. The initial size is fairly arbitrary, but setting it a little higher than expected helps prevent *reallocs*.

```
⟨ Definitions 4 ⟩ +≡
  void mp_reset_cur_string(MP mp);
```

20.

```
  void mp_reset_cur_string(MP mp)
  {
    mp_xfree(mp→cur_string);
    mp→cur_length = 0;
    mp→cur_string_size = 63;
    mp→cur_string = (unsigned char *) mp_xmalloc(mp, 64, sizeof(unsigned char));
    memset(mp→cur_string, 0, 64);
  }
```

21. METAPOST's string expressions are implemented in a brute-force way: Every new string or substring that is needed is simply stored into the string pool. Space is eventually reclaimed using the aid of a simple system system of reference counts.

The number of references to string number s will be $s\rightarrow refs$. The special value $s\rightarrow refs = \texttt{MAX\_STR\_REF} = 127$ is used to denote an unknown positive number of references; such strings will never be recycled. If a string is ever referred to more than 126 times, simultaneously, we put it in this category.

```
⟨ Definitions 4 ⟩ +≡
#define MAX_STR_REF 127 /* "infinite" number of references */
#define add_str_ref(A)
  {
    if ((A)→refs < MAX_STR_REF) ((A)→refs)++;
  }
```

22. Here's what we do when a string reference disappears:

```
⟨ Definitions 4 ⟩ +≡
#define delete_str_ref(A) do
  {
    if ((A)→refs < MAX_STR_REF) {
      if ((A)→refs > 1) ((A)→refs)--;
      else mp_flush_string(mp,(A));
    }
  }
  while (0)
```

23. ⟨ Definitions 4 ⟩ +≡

```
void mp_flush_string(MP mp, mp_string s);
```

24.

```
void mp_flush_string(MP mp, mp_string s)
{
  if (s→refs ≡ 0) {
    mp→strs_in_use--;
    mp→pool_in_use = mp→pool_in_use − (integer)s→len;
    (void) avl_del(s, mp→strings, Λ);
  }
}
```

25. Some C literals that are used as values cannot be simply added, their reference count has to be set such that they can not be flushed.

```
mp_string mp_intern(MP mp, const char *s)
{
  mp_string r;
  r = mp_rts(mp, s);
  r→refs = MAX_STR_REF;
  return r;
}
```

26. ⟨ Definitions 4 ⟩ +≡

```
mp_string mp_intern(MP mp, const char *s);
```

27. Once a sequence of characters has been appended to *cur_string*, it officially becomes a string when the function *make_string* is called. This function returns a pointer to the new string as its value.

```
⟨ Definitions 4 ⟩ +≡
  mp_string mp_make_string(MP mp);
```

28.
```
mp_string mp_make_string(MP mp)
{  /* current string enters the pool */
  mp_string str;
  mp_lstring tmp;
  tmp.str = mp→cur_string;
  tmp.len = mp→cur_length;
  str = (mp_string) avl_find(&tmp, mp→strings);
  if (str ≡ Λ) {  /* not yet known */
    str = mp_xmalloc(mp, 1, sizeof (mp_lstring));
    str→str = mp→cur_string;
    str→len = tmp.len;
    assert(avl_ins(str, mp→strings, avl_false) > 0);
    str = (mp_string) avl_find(&tmp, mp→strings);
    mp→pool_in_use = mp→pool_in_use + (integer) str→len;
    if (mp→pool_in_use > mp→max_pl_used) mp→max_pl_used = mp→pool_in_use;
    mp→strs_in_use ++;
    if (mp→strs_in_use > mp→max_strs_used) mp→max_strs_used = mp→strs_in_use;
  }
  add_str_ref(str);
  mp_reset_cur_string(mp);
  return str;
}
```

29. Here is a routine that compares two strings in the string pool, and it does not assume that they have the same length. If the first string is lexicographically greater than, less than, or equal to the second, the result is respectively positive, negative, or zero.

```
⟨ Definitions 4 ⟩ +≡
  integer mp_str_vs_str(MP mp, mp_string s, mp_string t);
```

30.
```
integer mp_str_vs_str(MP mp, mp_string s, mp_string t)
{
  (void) mp;
  return comp_strings_entry(Λ, (const void *) s, (const void *) t);
}
```

31.
```
⟨ Definitions 4 ⟩ +≡
  mp_string mp_cat(MP mp, mp_string a, mp_string b);
```

32.
```
mp_string mp_cat(MP mp, mp_string a, mp_string b)
{
  mp_string str;
  size_t needed;
  size_t saved_cur_length = mp→cur_length;
  unsigned char *saved_cur_string = mp→cur_string;
  size_t saved_cur_string_size = mp→cur_string_size;
  needed = a→len + b→len;
  mp→cur_length = 0;      /* mp→cur_string = Λ; */      /* needs malloc, spotted by clang */
  mp→cur_string = (unsigned char *) mp_xmalloc(mp, needed + 1, sizeof(unsigned char));
  mp→cur_string_size = 0;
  str_room(needed + 1);
  (void) memcpy(mp→cur_string, a→str, a→len);
  (void) memcpy(mp→cur_string + a→len, b→str, b→len);
  mp→cur_length = needed;
  mp→cur_string[needed] = '\0';
  str = mp_make_string(mp);
  mp_xfree(mp→cur_string);      /* created by mp_make_string */
  mp→cur_length = saved_cur_length;
  mp→cur_string = saved_cur_string;
  mp→cur_string_size = saved_cur_string_size;
  return str;
}
```

33. ⟨ Definitions 4 ⟩ +≡
```
mp_string mp_chop_string(MP mp, mp_string s, integer a, integer b);
```

34. *mp_string mp_chop_string*(MP *mp*, *mp_string s*, *integer a*, *integer b*)

```
{
  integer l;      /* length of the original string */
  integer k;      /* runs from a to b */
  boolean reversed;      /* was a > b? */
  if (a ≤ b) reversed = false;
  else {
    reversed = true;
    k = a;
    a = b;
    b = k;
  }
  l = (integer)s→len;
  if (a < 0) {
    a = 0;
    if (b < 0) b = 0;
  }
  if (b > l) {
    b = l;
    if (a > l) a = l;
  }
  str_room((size_t)(b − a));
  if (reversed) {
    for (k = b − 1; k ≥ a; k−−) {
      append_char(*(s→str + k));
    }
  }
  else {
    for (k = a; k < b; k++) {
      append_char(*(s→str + k));
    }
  }
  return mp_make_string(mp);
}
```

⟨ Definitions 4, 8, 13, 14, 18, 19, 21, 22, 23, 26, 27, 29, 31, 33 ⟩ Used in section 3.
⟨ `mpstrings.h` 3 ⟩

MPSTRINGS