1. METREP. This program reads BHT files to locate recent METAR reports for given station IDs. (The BHT files are written using the `metardb` program by means of the `LDM`.) The program decodes the report(s) and writes output in "*key*:*value*" format.

Report data is supplemented by station data (such as station location, city, timezone) which is retrieved from another file.

⟨ Includes 3 ⟩
⟨ Compile line Defines 5 ⟩
⟨ Structure Definitions 40 ⟩
⟨ Global Variables 24 ⟩
⟨ Forward Defs 13 ⟩
⟨ Function Definitions 4 ⟩
⟨ Main Program 30 ⟩

2. Useful symbolic definitions.

```
#define TRUE (1 ≡ 1)
#define FALSE (¬(TRUE))
```

3. ⟨ Includes 3 ⟩ ≡

```
#include <stdlib.h>
#include <sys/types.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include "bhtf.h"
```

See also sections 34, 150, 368, and 373.

This code is used in section 1.

4. Make sure the lookup string is uppercase.

⟨ Function Definitions 4 ⟩ ≡

```
void fixup(char *s)
{
  while (*s) {
    if (islower(*s)) *s ← toupper(*s);
    s++;
  }
}
```

See also sections 6, 7, 8, 12, 14, 16, 18, 20, 22, 42, 46, 47, 51, 58, 64, 66, 115, 117, 239, 293, 342, 349, 350, 353, 362, 382, and 383.

This code is used in section 1.

5. First, we need to identify the correct file. If we're not passed `METARDIR` on the compiler command line, we go with a default.

⟨ Compile line Defines 5 ⟩ ≡

```
#ifdef METARDIR
#define FILE_PREFIX  METARDIR
#else
#define FILE_PREFIX  "/rtwx/metar"
#endif
#define FILE_SUFFIX  ".bht"
```

See also section 35.

This code is used in section 1.

6. Create a file name string from the time t.

⟨ Function Definitions 4 ⟩ +≡

```
char *mk_fstr(time_t *t)
{
  char *fstr;
  struct tm dt;
  fstr ← malloc(strlen(FILE_PREFIX) + strlen(FILE_SUFFIX) + 22);
  gmtime_r(t, &dt);
  sprintf(fstr, "%s/%4d%02d%02d/%4d%02d%02d%02d%s", FILE_PREFIX, dt.tm_year + 1900, dt.tm_mon + 1,
      dt.tm_mday, dt.tm_year + 1900, dt.tm_mon + 1, dt.tm_mday, dt.tm_hour, FILE_SUFFIX);
  return fstr;
}
```

7. Since a single station return value may contain several reports, we need a way to step through each line.

⟨ Function Definitions 4 ⟩ +≡

```
char *find_next_line(char *s)
{
  if (¬s) return Λ;
  while (*s ∧ *s ≠ '\n') s++;
  return ++s;    /* skip past the newline character */
}
```

8. Look up METAR reports for station *stn* in the appropriate BHTF file(s) between (and including) time *ts* and time *te*. If we're only looking for the most recent report (if *last* ≡ `TRUE`), we'll start at the most recent time *te* and work backwards. Otherwise, we'll start from the oldest time *ts* and work forwards.

```
⟨ Function Definitions 4 ⟩ +≡
  void dolookup(char *stn, time_t *ts, time_t *te)
  {
    static int written ← 0;
    int atleastone ← 0;
    char *fstr, *value, *start;
    BHTFile bhtf;
    time_t t;
    DMptr dm;

    fixup(stn);
    if (last) t ← *te;
    else t ← *ts;
    do {
      fstr ← mk_fstr(&t);
      if (verbose) fprintf(stderr, "Trying␣file:␣%s␣for␣key␣%s\n", fstr, stn);
      if ((bhtf ← bht_open(fstr, BHT_READ, 0, 0, 2)) ≠ Λ) {
        value ← bht_getkey(bhtf, stn, 0);
        if (value) {
          ⟨ Process results returned in value 9 ⟩
          free(value);
        }
        bht_close(bhtf);
      }
      free(fstr);
      if (last ∧ atleastone) return;
      if (last) t −= (60 * 60);
      else t += (60 * 60);
    } while ((last ∧ t ≥ *ts) ∨ (¬last ∧ t ≤ *te));
  }
```

9. The *value* returned from the station lookup may contain several reports. Each report will end with a single newline `'\n'` character. The *decode_metar*( ) function will stop its work when it reaches a newline character, so we'll set *start* to point to the first report it should process. After parsing, we'll advance *start* to the beginning of the next report.

Depending on default or command-line options, we may filter the reports before output. Some report filters will work on the raw reports returned in *value*, and some filters may need to operate on the parsed report.

The variable *atleastone* will keep track of the count of reports that have been written for this run of *dolookup*( ). This is used to determine that we've found at least one report after filtering for this particular station lookup. The variable *written* will keep track of the total number of reports that have been written over the run of the program. It's only use at present is to add a separating newline between reports.

```
⟨ Process results returned in value 9 ⟩ ≡
  ⟨ Do initial report filtering 10 ⟩
  start ← value;
  while (*start) {
    dm ← decode_metar(start, &t);
    if (⟨ Post-parsing filtering 11 ⟩) {
      atleastone++;
      if (written++) printf("\n");
      write_metar(dm);
    }
    free_metar(dm);
    start ← find_next_line(start);
  }
```

This code is used in section 8.

10. These filters work on the raw reports (i.e., before parsing). These filters may remove certain reports from the *value* string, but they return no value themselves.

```
⟨ Do initial report filtering 10 ⟩ ≡
  if (metar_only) filter_rtype(value, METAR_ONLY);
  if (speci_only) filter_rtype(value, SPECI_ONLY);
  if (last) filter_last(value);
  if (best) filter_best(value);
```

This code is used in section 9.

11. Filters that make decisions based on a parsed report should be inserted here. These filters should return `TRUE` if the report should be written, or `FALSE` otherwise. This section is evaluated as the test expression of an **if**-statement, so the expression should be written to evaluate to `TRUE` or `FALSE`.

```
⟨ Post-parsing filtering 11 ⟩ ≡
  (¬present_wx_only ∨ filter_preswx(dm))
```

This code is used in section 9.

12. Filter reports so that only METARs or SPECIs are returned, as indicated.

```
#define METAR_ONLY 1
#define SPECI_ONLY 2
```

⟨ Function Definitions 4 ⟩ +≡

```
void filter_rtype(char *reps, int rtype)
{
  Cptr tmp ← (Cptr) malloc(strlen(reps) + 1), start, cmpstr ← "ZZZZZ";
  if (rtype ≡ METAR_ONLY) cmpstr ← "METAR";
  else if (rtype ← SPECI_ONLY) cmpstr ← "SPECI";
  tmp[0] ← '\0';
  start ← reps;
  while (*start) {
    if (strncmp(start, cmpstr, 5) ≡ 0) catline(tmp, start);
    start ← find_next_line(start);
  }
  strcpy(reps, tmp);
  free(tmp);
  return;
}
```

13. ⟨ Forward Defs 13 ⟩ ≡

```
void filter_rtype(char *, int);
```

See also sections 15, 17, 19, 21, 23, 48, 62, 65, 67, 116, 118, 294, 343, 352, 354, 363, and 388.

This code is used in section 1.

14. Filter reports so that only the most recent time is included. There could be multiple reports for the same time.

```
#define MAX_REPORTS 20
```

⟨ Function Definitions 4 ⟩ +≡

```
void filter_last(char *reps)
{
  Cptr r[MAX_REPORTS], tmp ← (Cptr) malloc(strlen(reps) + 1), start;
  int i, nreps ← 0, timeparsed, besttime ← −1;
  start ← reps;
  while (*start) {
    if (nreps < MAX_REPORTS) {
      r[nreps++] ← start;
      timeparsed ← parsetime(start);
      if (timeparsed > besttime) besttime ← timeparsed;
    } else break;
    start ← find_next_line(start);
  }
  tmp[0] ← '\0';
  for (i ← 0; i < nreps; i++)
    if (parsetime(r[i]) ≡ besttime) catline(tmp, r[i]);
  strcpy(reps, tmp);
  free(tmp);
  return;
}
```

15. ⟨ Forward Defs 13 ⟩ +≡

```
void filter_last(char *);
```

16. Concatenate source string s onto the end of destination string d, up to and including the first newline character (or Λ). The destination string d shall be Λ-terminated.

⟨ Function Definitions 4 ⟩ +≡

```
void catline(char *d, const char *s)
{
  while (*d) d++;
  while (*d++ ← *s++)
    if (*(s − 1) ≡ '\n') {
      *d ← '\0';
      break;
    }
}
```

17. ⟨ Forward Defs 13 ⟩ +≡

```
void catline(char *, const char *);
```

18. Extract the date/time string from a METAR report pointed to by s and calculate the number of minutes since the beginning of the month.

⟨ Function Definitions 4 ⟩ +≡

```
int parsetime(const char *s)
{
  int day, hr, min;
  sscanf(s + 11, "%02d%02d%02d", &day, &hr, &min);
  return ((day − 1) * 24 * 60 + hr * 60 + min);
}
```

19. ⟨ Forward Defs 13 ⟩ +≡

```
int parsetime(const char *);
```

20. From a collection of METAR reports, identify the best report for each time slot and eliminate the others. For our purposes, the best report for a given time will be the longest one.

```
#define MAX_BEST_REPS 30

⟨ Function Definitions 4 ⟩ +≡
  void filter_best(char *reps)
  {
    Cptr r[MAX_BEST_REPS], tmp ← (Cptr) malloc(strlen(reps) + 1), start;
    int t[MAX_BEST_REPS], l[MAX_BEST_REPS];
    int i, j, nreps ← 0, best_idx, best_len, currtime;

    start ← reps;
    while (*start) {
      if (nreps < MAX_BEST_REPS) {
        r[nreps] ← start;
        t[nreps] ← parsetime(start);
        l[nreps] ← linelen(start);
        nreps++;
      } else break;
      start ← find_next_line(start);
    }
    tmp[0] ← '\0';
    for (i ← 0; i < nreps; i++) {
      if (l[i]) {
        currtime ← t[i]; best_len ← l[i]; best_idx ← i;
        for (j ← i + 1; j < nreps; j++)
          if (t[j] ≡ currtime) {
            if (l[j] ≥ best_len) {
              l[best_idx] ← 0;
              best_idx ← j;
              best_len ← l[j];
            } else l[j] ← 0;
          }
        catline(tmp, r[best_idx]);
        l[best_idx] ← 0;
      }
    }
    strcpy(reps, tmp);
    free(tmp);
    return;
  }
```

21. ⟨ Forward Defs 13 ⟩ +≡

```
  void filter_best(char *);
```

22. Return the count of characters in a string up to and including the first newline character.

```
⟨ Function Definitions 4 ⟩ +≡
  int linelen(const char *s)
  {
    const char *t ← s;
    while (*t)
      if (*t++ ≡ '\n') break;
    return (t − s);
  }
```

23.
```
⟨ Forward Defs 13 ⟩ +≡
  int linelen(const char *);
```

24. This global variable keeps track of the command-line specification for more feedback during the report search process. Turned off by default, *verbose* ← `TRUE` when `"-v"` is seen on the command line.

```
⟨ Global Variables 24 ⟩ ≡
  int verbose ← FALSE;
```

See also sections 25, 26, 27, 28, 29, 36, 44, 151, 152, 165, 166, 167, and 168.

This code is used in section 1.

25. By default, we will only return the value for the most recent report for any station. If we want to see all the reports for the current time period, *last* ← `FALSE` when recognizing `"-a"` ("all") on the command line. It may be turned back on with a `"-l"` ("last") on the command line.

```
⟨ Global Variables 24 ⟩ +≡
  int last ← TRUE;
```

26. If a file contains multiple reports for the same time, the "best" report will be selected by default. The best report is just the longest one. This selection is disabled with a `"-r"` ("redundant") on the command line, and re-enabled with `"-b"` ("best").

```
⟨ Global Variables 24 ⟩ +≡
  int best ← TRUE;
```

27. Specify that only regular METARs shall be shown (i.e., SPECI reports are filtered).

```
⟨ Global Variables 24 ⟩ +≡
  int metar_only ← FALSE;
```

28. Specify that only SPECI reports are to be shown (i.e., METAR reports are filtered).

```
⟨ Global Variables 24 ⟩ +≡
  int speci_only ← FALSE;
```

29. Specify that only reports which include some sort of present weather shall be displayed. This is a more computationally expensive filter, since the report will need to be parsed before the filter can be applied.

```
⟨ Global Variables 24 ⟩ +≡
  int present_wx_only ← FALSE;
```

30. Very simple main loop. Command-line arguments are processed and looked up to retrieve their most recent METAR reports. For the future, since file I/O is relatively expensive, it might be better to aggregate station searches for each file, rather than doing an individual search through up to `MAXHOURS` files for each station.

```
#define MAXHOURS 36
#define SECS_IN_HR (60 * 60)
⟨ Main Program 30 ⟩ ≡
  int main(int argc, char *argv[])
  {
    time_t t, ts ← 0, te ← 0;
    char oneline[512];
    int exitval ← 0;
    ⟨ Initialize wx_regex 153 ⟩
    ⟨ Open station ID file or exit on error 37 ⟩
    t ← time(0);
    argc−−;
    argv++;
    while (argc−−) {
      ⟨ Process command-line options and station lookups 31 ⟩
      *argv++;
    }
  Exit: ⟨ Close station ID file 38 ⟩
    exit(exitval);
  }
```

This code is used in section 1.

31. ⟨ Process command-line options and station lookups 31 ⟩ ≡

```
  if (strcmp(*argv, "-v") ≡ 0) verbose ← TRUE;
  else if (strcmp(*argv, "-a") ≡ 0) last ← FALSE;
  else if (strcmp(*argv, "-l") ≡ 0) last ← TRUE;
  else if (strcmp(*argv, "-r") ≡ 0) best ← FALSE;
  else if (strcmp(*argv, "-b") ≡ 0) best ← TRUE;
  else if (strcmp(*argv, "-m") ≡ 0) {
    metar_only ← TRUE;
    speci_only ← FALSE;
  } else if (strcmp(*argv, "-s") ≡ 0) {
    speci_only ← TRUE;
    metar_only ← FALSE;
  } else if (strcmp(*argv, "-P") ≡ 0) present_wx_only ← TRUE;
  else if (**argv ≡ '-' ∨ **argv ≡ '+') {
    ⟨ Check for command-line time specs 32 ⟩
  } else {
    ⟨ Verify time range and perform station lookup 33 ⟩
  }
```

This code is used in section 30.

32. A command-line option that looks like `-`n or `+`m will limit the search for reports between the specified number of hours from the current time. A `-5` option, for example, will limit the search to within the past 5 hours, while a `+2` option will search for reports more than 2 hours ago. These options may be combined. An improper combination like `+5 -3` will generate a command-line error when the station lookup occurs.

```
⟨ Check for command-line time specs 32 ⟩ ≡
  if (¬isdigit(*((*argv) + 1))) {
    fprintf(stderr, "Unrecognized␣option:␣%s\n", *argv);
    exitval ← 1;
    goto Exit;
  } else {
    if (**argv ≡ '-') ts ← t − SECS_IN_HR * atoi((*argv) + 1);
    else te ← t − SECS_IN_HR * atoi((*argv) + 1);
  }
```

This code is used in section 31.

33.

```
⟨ Verify time range and perform station lookup 33 ⟩ ≡
  if (¬ts) ts ← t − (last ? MAXHOURS * SECS_IN_HR : SECS_IN_HR);
  if (¬te) te ← t;
  if (ts > te) {
    fprintf(stderr, "Command-line␣Error:␣Ending␣time␣comes␣before␣starting␣time.\n");
    exitval ← 1;
    goto Exit;
  } else dolookup(*argv, &ts, &te);
```

This code is used in section 31.

34. We want to include some additional information about each station, and we find the data in an external file in GDB format. We'll open the file once near the beginning of the program so that we can avoid opening/closing overhead for each station.

```
⟨ Includes 3 ⟩ +≡
#include <gdbm.h>
```

35. If we have been compiled with a command-line define for `STNFILE`, use that specification for which file we need to open to access station information.

```
⟨ Compile line Defines 5 ⟩ +≡
#ifdef STNFILE
#define GDB_FILE_NAME  STNFILE
#else
#define GDB_FILE_NAME  "master-lid.gdb"
#endif
```

36. To avoid having to pass the file handle around, we'll make it a global variable for convenience.

```
⟨ Global Variables 24 ⟩ +≡
  GDBM_FILE gdbf ← Λ;
```

37. Open the file for reading.

```
⟨ Open station ID file or exit on error 37 ⟩ ≡
  if ((gdbf ← gdbm_open(GDB_FILE_NAME, 0, GDBM_READER, °644, Λ)) ≡ Λ) {
    fprintf(stderr, "Couldn't␣open␣Station␣info␣file␣'%s'\n", GDB_FILE_NAME);
    exit(2);
  }
```

This code is used in section 30.

38. ⟨ Close station ID file 38 ⟩ ≡
if (*gdbf*) *gdbm_close*(*gdbf*);

This code is used in section 30.

39. Decoding. In order to parse the information out of a METAR report, we first need a place to put the data. As we develop means of parsing particular data types from reports, we'll add to the *decoded_metar* structure.

40. ⟨ Structure Definitions 40 ⟩ ≡
⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩
struct decoded_metar {
⟨ METAR storage variables 69 ⟩
⟨ Remark Storage 211 ⟩
};
typedef struct decoded_metar ∗**DMptr**;

This code is used in section 1.

41. ⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ ≡
typedef char ∗**Cptr**;

See also sections 43, 45, 73, 96, 104, 126, 186, 223, 307, and 336.

This code is used in section 40.

42. ⟨ Function Definitions 4 ⟩ +≡
⟨ Parse helpers 55 ⟩

43. As we're stepping through elements of the METAR report, it is helpful to know what we should be expecting next. Tokens in the body of the report will usually not be recognized out of order, so parser states listed here should be collected in the order in which they'll be recognized. The first state provided is `TS_UNKNOWN`.

⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ +≡
enum ParseStates {
`TS_UNKNOWN` ← 0, ⟨ Token-parsing states 76 ⟩
};

44. We keep the parser state in the global variable *parse_state*. Occasionally we may need to backtrack, and so we have a second variable *last_state* which records the last successful parsing state.

⟨ Global Variables 24 ⟩ +≡
enum ParseStates *parse_state* ← `TS_UNKNOWN`, *last_state* ← `TS_UNKNOWN`;

45. We need some token types.

⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ +≡
enum ttype {
`TOK_NONE` ← 0, ⟨ Token types 77 ⟩`TOK_END`, `TOK_UNKNOWN`
};

46. Create and initialize a **decoded_metar** structure. Any **int** or **float** that has not been explicitly set shall be initialized to the value `FLAG_MISSING`. Any pointer value should be set to Λ. Return Λ on error, or a pointer to the newly-allocated structure otherwise. Various **enum** types should be initialized to some explicit state that indicates unset.

```
#define FLAG_MISSING -999
```

```
⟨Function Definitions 4⟩ +≡
  DMptr init_metar(void)
  {
    DMptr dm ← (DMptr) malloc(sizeof(struct decoded_metar));
    int i;
    if (¬dm) return Λ;
    ⟨METAR initialization statements 70⟩
    return dm;
  }
```

47. And to deallocate the structure:

```
⟨Function Definitions 4⟩ +≡
  void free_metar(DMptr dm)
  {
    int i;
    if (¬dm) return;
    ⟨METAR deallocation statements 71⟩
    free(dm);
  }
```

48.

```
⟨Forward Defs 13⟩ +≡
  void free_metar(DMptr dm);
```

49. Parsing.

50. The first element we expect from a METAR report is the report type, and so we set *parse_state* appropriately.

⟨ Initialize parse state 50 ⟩ ≡
```
parse_state ← TS_EXP_REP;
```
This code is used in section 58.

51. Return the METAR token type pointed to by s.

⟨ Function Definitions 4 ⟩ +≡
```
enum ttype token(const char *s)
{
  const char *e ← s;
  ⟨ If end of METAR, return final token 53 ⟩
  ⟨ Move e to end of token 54 ⟩
ReTry:
  switch (parse_state) {
    ⟨ Analyze current token according to state 78 ⟩
  default: ⟨ Default token processing 52 ⟩
  }
  parse_state ← TS_UNKNOWN;
  return TOK_UNKNOWN;
}
```

52. ⟨ Default token processing 52 ⟩ ≡
```
parse_state ← TS_UNKNOWN;
return TOK_UNKNOWN;
```
This code is used in section 51.

53. The end of the report will either be a newline character, an equal sign character (`'='`), or a null character.

⟨ If end of METAR, return final token 53 ⟩ ≡
```
if (¬*s ∨ (*s ≡ '=') ∨ (*s ≡ '\n')) {
  parse_state ← TS_UNKNOWN;
  return TOK_END;
}
```
This code is used in section 51.

54. We set e to point just past the last character of the current token.

⟨ Move e to end of token 54 ⟩ ≡
```
while (*e ∧ ¬isspace(*e) ∧ *e ≠ '=') e++;
```
This code is used in section 51.

55. Return a pointer to the first token, skipping past any leading space. Return Λ if no token was to be found.

```
⟨ Parse helpers 55 ⟩ ≡
  inline const char *first_token(const char *s)
  {
    if (¬s) return Λ;
    while (*s ∧ isspace(*s)) s++;
    if (*s) return s;
    return Λ;
  }
```

See also sections 56, 57, 169, 170, 266, 267, and 268.

This code is used in section 42.

56. Return a pointer to the next token after the one pointed to by s.

```
⟨ Parse helpers 55 ⟩ +≡
  inline const char *advance_token(const char *s)
  {
    while (*s ∧ ¬isspace(*s) ∧ *s ≠ '=') s++;
    while (*s ∧ isspace(*s) ∧ *s ≠ '\n') s++;
    return s;
  }
```

57. Scan to the end of the current token pointed to by s.

```
⟨ Parse helpers 55 ⟩ +≡
  inline const char *find_token_end(const char *s)
  {
    while (*s ∧ ¬isspace(*s) ∧ *s ≠ '=') s++;
    return s;
  }
```

58. Decode a METAR report string into a structure.

```
#define TMPBUF_SZ 512
⟨ Function Definitions 4 ⟩ +≡
  DMptr decode_metar(const char *s, time_t *t)
  {
    char rembuf[TMPBUF_SZ], unkbuf[TMPBUF_SZ], rawbuf[TMPBUF_SZ];
    int remcntr ← 0, unkcntr ← 0;
    enum ttype thetoken;
    const char *e;
    DMptr dm;
    if (¬(s ← first_token(s))) return Λ;
    if (¬(dm ← init_metar())) return Λ;
    ⟨ Save a copy of the raw report 59 ⟩
    ⟨ Initialize parse state 50 ⟩
    ⟨ Process the tokens 61 ⟩
    ⟨ Transfer accumulated miscellaneous tokens 60 ⟩
    ⟨ Retrieve info from other sources 341 ⟩
    return dm;
  }
```

59. For debugging and comparison purposes, it is useful to see the original report string.

```
⟨ Save a copy of the raw report 59 ⟩ ≡
  rawbuf[0] ← '\0';
  catline(rawbuf, s);
  dm→rawrep ← strdup(rawbuf);
```

This code is used in section 58.

60.
```
⟨ Transfer accumulated miscellaneous tokens 60 ⟩ ≡
  if (remcntr) dm→remarks ← strdup(rembuf);
  if (unkcntr) dm→unknown ← strdup(unkbuf);
```

This code is used in section 58.

61. First, identify the current token pointed to by s, and then find the end of the token e. Normally the end of token is white space. However, a few tokens conditionally (or unconditionally) include white space, so the end of token marker e may be advanced while the token is parsed. The next token is found relative to the end of the current token *after* parsing.

```
⟨ Process the tokens 61 ⟩ ≡
  while ((thetoken ← token(s)) ≠ TOK_END) {
    e ← find_token_end(s);
    ⟨ Update successful token search parsing state 63 ⟩
    switch (thetoken) {
      ⟨ Parse token according to type 79 ⟩
    default: fprintf(stderr, "Unknown␣token␣in␣unknown␣state:␣%.*s", (int)(e − s), s);
    }
    s ← advance_token(e);
  }
```

This code is used in section 58.

62.
```
⟨ Forward Defs 13 ⟩ +≡
  DMptr decode_metar(const char *, time_t *);
```

63. If we have successfully recognized a token, then we want to update *last_state* to point to the current parsing search state. If the token following the current token is not recognized, we'll end up skipping that token and restarting at *last_state* for the succeeding token. Before we preserve token search states, however, we must have at least successfully parsed the report type, station ID, and report time.

```
⟨ Update successful token search parsing state 63 ⟩ ≡
  if ((thetoken ≠ TOK_UNKNOWN) ∧ (parse_state > TS_EXP_TIME)) last_state ← parse_state;
```

This code is used in section 61.

64.
```
⟨ Function Definitions 4 ⟩ +≡
  int filter_preswx(DMptr dm)
  {
    if (dm→wxcntr) return TRUE;
    return FALSE;
  }
```

65.
```
⟨ Forward Defs 13 ⟩ +≡
  int filter_preswx(DMptr);
```

66. Translate the **decoded_metar** into output. Return the pointer to the structure as a convenience.

```
⟨ Function Definitions 4 ⟩ +≡
  DMptr write_metar(DMptr dm)
  {
    int i;
    if (¬dm) return Λ;
    ⟨ Output statements 72 ⟩
    if (dm→remarks) printf("Remarks:␣%s\n", dm→remarks);
    if (dm→unknown) printf("Unknown:␣%s\n", dm→unknown);
    return dm;
  }
```

67. ⟨ Forward Defs 13 ⟩ +≡
  DMptr *write_metar*(**DMptr**);

68. METAR elements. For each kind of thing that we want to parse from the METAR report, we'll have a series of entries. These should be addressed in the order in which we expect them to occur. We'll create a variable in the structure, an initialization, a de-allocation (if required), a token, a parser state, instructions for parsing the values into the structure, and instructions for translating the conversions into output.

69. Raw report storage. This isn't a parsed item, for convenience it's merely a copy of the raw report. It's useful to have it around for debugging purposes.

```
⟨ METAR storage variables 69 ⟩ ≡
  Cptr rawrep;
```

See also sections 74, 81, 89, 97, 105, 119, 127, 136, 147, 171, 179, 187, 195, 199, 339, 344, 355, 359, 366, 371, and 377.

This code is used in section 40.

70. ⟨ METAR initialization statements 70 ⟩ ≡
  dm→rawrep ← Λ;

See also sections 75, 82, 90, 98, 106, 120, 128, 137, 148, 172, 180, 188, 196, 200, 212, 218, 225, 231, 242, 251, 270, 276, 282, 288, 296, 302, 309, 315, 323, 330, 340, 345, 356, 360, 367, 372, and 378.

This code is used in section 46.

71. ⟨ METAR deallocation statements 71 ⟩ ≡
  if (*dm→rawrep*) *free*(*dm→rawrep*);

See also sections 83, 138, 149, 173, 197, 201, 252, 324, 331, 346, 361, and 379.

This code is used in section 47.

72. ⟨ Output statements 72 ⟩ ≡
  if (*dm→rawrep*) *printf*("Raw:␣%s", *dm→rawrep*);

See also sections 80, 88, 95, 103, 114, 135, 143, 158, 178, 185, 194, 216, 222, 229, 237, 238, 240, 248, 249, 260, 261, 262, 263, 264, 265, 274, 280, 286, 292, 300, 306, 313, 320, 321, 328, 335, 348, 358, 365, 370, 375, and 381.

This code is used in section 66.

73. Report type. The first element of a METAR report is the report type (required). A report will either be a regular METAR, or a special report.

```
⟨ enums and typedefs for certain METAR values 41 ⟩ +≡
  enum RepType {
    TYPE_NONE, TYPE_METAR, TYPE_SPECI
  };
```

74. ⟨ METAR storage variables 69 ⟩ +≡
enum RepType *report_type*;

75. ⟨ METAR initialization statements 70 ⟩ +≡
dm→*report_type* ← TYPE_NONE;

76. ⟨ Token-parsing states 76 ⟩ ≡
TS_EXP_REP ,

See also sections 84, 91, 99, 108, 122, 130, 140, 154, 175, 182, 190, 203, and 208.

This code is used in section 43.

77. ⟨ Token types 77 ⟩ ≡
TOK_REPTYPE_SPECI, TOK_REPTYPE_METAR ,

See also sections 85, 92, 100, 107, 121, 129, 139, 155, 174, 181, 189, 202, 207, 213, 219, 226, 232, 243, 253, 271, 277, 283, 289, 297, 303, 310, 316, 325, and 332.

This code is used in section 45.

78. This is a required token. If we are not able to recognize the string, there's little point in going forward.

⟨ Analyze current token according to state 78 ⟩ ≡
case TS_EXP_REP:
parse_state++;
if (*strncmp*(*s*, "METAR", *e* − *s*) ≡ 0) **return** TOK_REPTYPE_METAR;
if (*strncmp*(*s*, "SPECI", *e* − *s*) ≡ 0) **return** TOK_REPTYPE_SPECI;
parse_state ← TS_UNKNOWN;
return TOK_UNKNOWN;

See also sections 86, 93, 101, 109, 123, 131, 141, 156, 176, 183, 191, 204, and 209.

This code is used in section 51.

79. ⟨ Parse token according to type 79 ⟩ ≡
case TOK_REPTYPE_SPECI: *dm*→*report_type* ← TYPE_SPECI;
break;
case TOK_REPTYPE_METAR: *dm*→*report_type* ← TYPE_METAR;
break;

See also sections 87, 94, 102, 110, 111, 112, 113, 124, 132, 133, 134, 142, 157, 177, 184, 192, 193, 198, 206, 210, 215, 221, 228, 234, 235, 236, 245, 246, 247, 255, 256, 257, 258, 259, 273, 279, 285, 291, 299, 305, 312, 318, 319, 327, and 334.

This code is used in section 61.

80. ⟨ Output statements 72 ⟩ +≡
printf("Type:␣%s\n", *dm*→*report_type* ≡ TYPE_METAR ? "METAR" : "SPECI");

81. Station ID. The next element is a 4-character station ID, and it is also required.

⟨ METAR storage variables 69 ⟩ +≡
Cptr *station_id*;

82. ⟨ METAR initialization statements 70 ⟩ +≡
dm→*station_id* ← Λ;

83. ⟨ METAR deallocation statements 71 ⟩ +≡
if (*dm*→*station_id*) *free*(*dm*→*station_id*);

84. ⟨ Token-parsing states 76 ⟩ +≡
TS_EXP_ID ,

85. ⟨ Token types 77 ⟩ +≡
TOK_STATID ,

86. ⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_ID:
parse_state++;
if $(e - s \equiv 4)$ **return** TOK_STATID;
parse_state ← TS_UNKNOWN;
return TOK_UNKNOWN;

87. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_STATID:
dm→*station_id* ← *strndup*(*s*, 4);
break;

88. ⟨ Output statements 72 ⟩ +≡
printf("StID:␣%s\n", *dm*→*station_id*);

89. **Time spec.** The next required element is the report time spec.

⟨ METAR storage variables 69 ⟩ +≡
int *day*, *hour*, *min*;

90. ⟨ METAR initialization statements 70 ⟩ +≡
dm→*day* ← *dm*→*hour* ← *dm*→*min* ← FLAG_MISSING;

91. ⟨ Token-parsing states 76 ⟩ +≡
TS_EXP_TIME ,

92. ⟨ Token types 77 ⟩ +≡
TOK_TIME ,

93. ⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_TIME:
parse_state++;
if $((e - s \equiv 7) \wedge (*(e - 1) \equiv$ 'Z'$))$ **return** TOK_TIME;
parse_state ← TS_UNKNOWN;
return TOK_UNKNOWN;

94. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_TIME:
sscanf(*s*, "%02d%02d%02d", &*dm*→*day*, &*dm*→*hour*, &*dm*→*min*);
break;

95. ⟨ Output statements 72 ⟩ +≡
printf("Date:␣%d\n", *dm*→*day*);
printf("Time:␣%02d:%02d␣Z\n", *dm*→*hour*, *dm*→*min*);

96. Report modifier. The report modifier token is optional. If present, it will take on one of two values.

```
⟨ enums and typedefs for certain METAR values 41 ⟩ +≡
  enum RepMod {
    MOD_NONE, MOD_AUTO, MOD_COR
  };
```

97.
```
⟨ METAR storage variables 69 ⟩ +≡
  enum RepMod report_mod;
```

98.
```
⟨ METAR initialization statements 70 ⟩ +≡
  dm→report_mod ← MOD_NONE;
```

99.
```
⟨ Token-parsing states 76 ⟩ +≡
  TS_EXP_MOD ,
```

100.
```
⟨ Token types 77 ⟩ +≡
  TOK_REPMOD_AUTO, TOK_REPMOD_COR ,
```

101. Since this token is optional, we should just try the next token type if this test is not successful.

```
⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_MOD:
  parse_state++;
  if (strncmp(s, "AUTO", e − s) ≡ 0) return TOK_REPMOD_AUTO;
  if (strncmp(s, "COR", e − s) ≡ 0) return TOK_REPMOD_COR;
  goto ReTry;
```

102.
```
⟨ Parse token according to type 79 ⟩ +≡
case TOK_REPMOD_AUTO: dm→report_mod ← MOD_AUTO;
  break;
case TOK_REPMOD_COR: dm→report_mod ← MOD_COR;
  break;
```

103.
```
⟨ Output statements 72 ⟩ +≡
  if (dm→report_mod ≠ MOD_NONE) printf("Mod:␣%s\n", dm→report_mod ≡ MOD_AUTO ? "AUTO" : "COR");
```

104. Wind. Wind indications are next. European wind speeds may be recorded in meters per second rather than knots, so in addition to the storage for the values themselves, we also need to know what units have been specified.

```
⟨ enums and typedefs for certain METAR values 41 ⟩ +≡
  enum RepSpeed {
    SPD_KT, SPD_MPS
  };
```

105.
```
⟨ METAR storage variables 69 ⟩ +≡
  int wind_dir, wind_spd, wind_gust;
  enum RepSpeed speed_indicator;
```

106. The *speed_indicator* needs no explicit initialization since its value is not required unless there are values for the other wind parameters. However, we'll initialize it to SPD_KT and reset the value later if we discover our initial guess was mistaken.

```
⟨ METAR initialization statements 70 ⟩ +≡
  dm→wind_dir ← dm→wind_spd ← dm→wind_gust ← FLAG_MISSING;
  dm→speed_indicator ← SPD_KT;
```

107. There are several variations on the wind type.

```
⟨ Token types 77 ⟩ +≡
  TOK_WIND, TOK_GUST, TOK_WINDVRB, TOK_WINDVRBG ,
```

108.
```
⟨ Token-parsing states 76 ⟩ +≡
  TS_EXP_WIND ,
```

109.
```
⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_WIND:
  parse_state++;
  if ((strncmp(e − 2, "KT", 2) ≡ 0) ∨ (strncmp(e − 3, "MPS", 3) ≡ 0)) {
    if (strncmp(s, "VRB", 3) ≡ 0) {
      if (*(s + 5) ≡ 'G') return TOK_WINDVRBG;    /* Don't know if this is legal */
      else return TOK_WINDVRB;
    }
    if ((e − s ≥ 7) ∧ (e − s ≤ 9)) return TOK_WIND;
    if (*(s + 5) ≡ 'G' ∨ *(s + 6) ≡ 'G') return TOK_GUST;
  }
  parse_state++;    /* If wind token is missing, skip past the variable wind token also. */
  goto ReTry;
```

110.
```
⟨ Parse token according to type 79 ⟩ +≡
case TOK_WIND:
  sscanf(s, "%03d%d", &dm→wind_dir, &dm→wind_spd);
  if (strncmp(e − 3, "MPS", 3) ≡ 0) dm→speed_indicator ← SPD_MPS;
  break;
```

111.
```
⟨ Parse token according to type 79 ⟩ +≡
case TOK_GUST:
  sscanf(s, "%03d%dG%d", &dm→wind_dir, &dm→wind_spd, &dm→wind_gust);
  if (strncmp(e − 3, "MPS", 3) ≡ 0) dm→speed_indicator ← SPD_MPS;
  break;
```

112. If the wind direction is recorded as variable, set the state accordingly.

```
#define WIND_VRB −800
⟨ Parse token according to type 79 ⟩ +≡
case TOK_WINDVRB:
  dm→wind_dir ← WIND_VRB;
  sscanf(s + 3, "%d", &dm→wind_spd);
  if (strncmp(e − 3, "MPS", 3) ≡ 0) dm→speed_indicator ← SPD_MPS;
  break;
```

113. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_WINDVRBG:
  dm→wind_dir ← WIND_VRB;
  sscanf(s + 3, "%dG%d", &dm→wind_spd, &dm→wind_gust);
  if (strncmp(e − 3, "MPS", 3) ≡ 0) dm→speed_indicator ← SPD_MPS;
  break;
```

114. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→wind_spd ≠ FLAG_MISSING) {
    if (dm→wind_dir ≡ 0 ∧ dm→wind_spd ≡ 0) printf("Winds:␣Calm");
    else {
      if (dm→wind_dir ≡ WIND_VRB) printf("Winds:␣variable");
      else {
        printf("Winds:␣from␣the␣%s", sixteenpt_dir(dm→wind_dir));
        ⟨ Output wind variability 125 ⟩
      }
      printf("␣at␣%d␣%s", dm→wind_spd, dm→speed_indicator ≡ SPD_KT ? "kts" : "mps");
      if (dm→wind_gust ≠ FLAG_MISSING)
        printf("␣with␣gusts␣to␣%d␣%s", dm→wind_gust, dm→speed_indicator ≡ SPD_KT ? "kts" : "mps");
    }
    printf("\n");
  }
```

115. Divide the compass into eight points of 45° each, centered on the compass point starting at North = 0°, moving clockwise. The wind direction is specified by a METAR report rounded to the nearest 10°, so the units digit and smaller is immaterial for the tests below. Return the string.

```
#define CIRC_MOD 360
#define SZ_EIGHTH (360/8)    /* 45° */
#define SZ_HALF8 (SZ_EIGHTH/2)
```

⟨ Function Definitions 4 ⟩ +≡

```
  Cptr eightpt_dir(int d)
  {
    static Cptr dstr[] ← {"North", "Northeast", "East", "Southeast", "South", "Southwest", "West",
        "Northwest", "North"};
    return dstr[((d + SZ_HALF8) % CIRC_MOD)/SZ_EIGHTH];
  }
```

116. ⟨ Forward Defs 13 ⟩ +≡

```
  Cptr eightpt_dir(int);
```

117. Divide the circle into 16 compass points.

```
#define SZ_SIXTEENTH (360/16)    /* 45° */
#define SZ_HALFSXT (SZ_SIXTEENTH/2)
```

⟨ Function Definitions 4 ⟩ +≡

```
Cptr sixteenpt_dir(int d)
{
  static Cptr dstr[] ← {"North", "North-Northeast", "Northeast", "East-Northeast", "East",
      "East-Southeast", "Southeast", "South-Southeast", "South", "South-Southwest",
      "Southwest", "West-Southwest", "West", "West-Northwest", "Northwest",
      "North-Northwest", "North"};
  return dstr[((d + SZ_HALFSXT) % CIRC_MOD)/SZ_SIXTEENTH];
}
```

118. ⟨ Forward Defs 13 ⟩ +≡

```
Cptr sixteenpt_dir(int);
```

119. Wind Variability. If there is a wind group, the variability group (if present) will follow it immediately. There should not be a variability group if there is no wind group.

⟨ METAR storage variables 69 ⟩ +≡

```
int var_wind_dir1, var_wind_dir2;
```

120. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→var_wind_dir1 ← dm→var_wind_dir2 ← FLAG_MISSING;
```

121. ⟨ Token types 77 ⟩ +≡

```
TOK_VARIABLE ,
```

122. ⟨ Token-parsing states 76 ⟩ +≡

```
TS_EXP_VAR ,
```

123. ⟨ Analyze current token according to state 78 ⟩ +≡

```
case TS_EXP_VAR:
  parse_state++;
  if ((*(s + 3) ≡ 'V') ∧ (e − s ≡ 7)) return TOK_VARIABLE;
  goto ReTry;
```

124. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_VARIABLE:
  sscanf(s, "%03dV%03d", &dm→var_wind_dir1, &dm→var_wind_dir2);
  break;
```

125. ⟨ Output wind variability 125 ⟩ ≡

```
if (dm→var_wind_dir1 ≠ FLAG_MISSING ∧ dm→var_wind_dir2 ≠ FLAG_MISSING)
  printf(", variable from %s to %s,", sixteenpt_dir(dm→var_wind_dir1),
      sixteenpt_dir(dm→var_wind_dir2));
```

This code is used in section 114.

126. Visibility. Visibility is recorded in statute miles or meters. Fractions of miles may be recorded, but only whole meters.

```
⟨ enums and typedefs for certain METAR values 41 ⟩ +≡
  enum RepDist {
    DIST_SM, DIST_METERS
  };
```

127.
```
⟨ METAR storage variables 69 ⟩ +≡
  float visibility;
  enum RepDist distance_indicator;
```

128. US visibility reports end with 'SM' (statute miles), but European reports have no indicator, merely a 4-digit number. We'll initialize *distance_indicator* to DIST_METERS, and we'll reset the value if we guessed wrong.

```
⟨ METAR initialization statements 70 ⟩ +≡
  dm→visibility ← FLAG_MISSING;
  dm→distance_indicator ← DIST_METERS;
```

129.
```
⟨ Token types 77 ⟩ +≡
  TOK_VISIBILITY, TOK_SPLITVIS, TOK_FRACVIS ,
```

130.
```
⟨ Token-parsing states 76 ⟩ +≡
  TS_EXP_VIS ,
```

131.
```
⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_VIS:
  parse_state++;
  if ((e − s) ≡ 4 ∧ isdigit(*s) ∧ isdigit(*(s + 1)) ∧ isdigit(*(s + 2)) ∧ isdigit(*(s + 3)))
    return TOK_VISIBILITY;    /* European option */
  if ((e − s) ≤ 2 ∧ isdigit(*(e − 1)) ∧ isdigit(*(e + 1)))  e ← find_token_end(e + 1);
  if (strncmp(e − 2, "SM", 2) ≡ 0) {
    do {
      if (*s ≡ '␣') return TOK_SPLITVIS;
      if (*s ≡ '/') return TOK_FRACVIS;
    } while (++s < e);
    return TOK_VISIBILITY;
  }
  goto ReTry;
```

132.
```
⟨ Parse token according to type 79 ⟩ +≡
case TOK_VISIBILITY:
  sscanf(s, "%f", &dm→visibility);
  if (strncmp(e − 2, "SM", 2) ≡ 0)  dm→distance_indicator ← DIST_SM;
  break;
```

133. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_FRACVIS:
  {
    float num, denom;
    if (*s ≡ 'M') s++;
    sscanf(s, "%f/%fSM", &num, &denom);
    dm→visibility ← num/denom;
    dm→distance_indicator ← DIST_SM;
  }
  break;
```

134. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_SPLITVIS:
  {
    float awhole, num, denom;
    sscanf(s, "%f", &awhole);
    s ← e;
    e ← find_token_end(e + 1);
    sscanf(s, "%f/%fSM", &num, &denom);
    dm→visibility ← awhole + (num/denom);
    dm→distance_indicator ← DIST_SM;
  }
  break;
```

135. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→visibility ≠ FLAG_MISSING)
    printf("Vis:␣%g␣%s\n", dm→visibility, dm→distance_indicator ≡ DIST_SM ? "miles" : "meters");
```

136. **Runway visibility.** This portion is optional.

```
#define MAXRUNWAYS 10
⟨ METAR storage variables 69 ⟩ +≡
  int runcntr;
  Cptr runway[MAXRUNWAYS];
```

137. ⟨ METAR initialization statements 70 ⟩ +≡

```
  dm→runcntr ← 0;
  for (i ← 0; i < MAXRUNWAYS; dm→runway[i++] ← Λ) ;
```

138. ⟨ METAR deallocation statements 71 ⟩ +≡

```
  for (i ← 0; i < dm→runcntr; i++) free(dm→runway[i]);
```

139. ⟨ Token types 77 ⟩ +≡

```
  TOK_RUNWAY ,
```

140. ⟨ Token-parsing states 76 ⟩ +≡

```
  TS_EXP_RUN ,
```

141. ⟨ Analyze current token according to state 78 ⟩ +≡

```
case TS_EXP_RUN:
  if (*s ≡ 'R') {
    if (strncmp(e − 2, "FT", 2) ≡ 0) return TOK_RUNWAY;
    if (*(s + 3) ≡ '/' ∨ *(s + 4) ≡ '/') return TOK_RUNWAY;
    parse_state++;
    goto ReTry;
  } else {
    parse_state++;
    goto ReTry;
  }
```

142. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_RUNWAY:
  if (dm→runcntr < MAXRUNWAYS) dm→runway[dm→runcntr++] ← strndup(s, e − s);
  break;
```

143. Translate runway visibility statements into English.

⟨ Output statements 72 ⟩ +≡

```
  {
    char *s, *e;
    int minv, maxv;
    int greater1 ← 0, greater2 ← 0, less1 ← 0, less2 ← 0;
    for (i ← 0; i < dm→runcntr; i++) {
      s ← dm→runway[i] + 1;      /* Skip leading 'R' */
      e ← strchr(s, '/');
      printf("Runway␣%.*s:␣", (int)(e − s), s);
      s ← e + 1;
      ⟨ Handle first P/M height 144 ⟩
      e ← strchr(s, 'V');
      if (¬e) {
        ⟨ Scan one height 145 ⟩
      } else {
        ⟨ Scan two heights 146 ⟩
      }
    }
  }
```

144. ⟨ Handle first P/M height 144 ⟩ ≡

```
  if (*s ≡ 'P') {
    s++;
    greater1 ++;
  } else if (*s ≡ 'M') {
    s++;
    less1 ++;
  }
```

This code is used in section 143.

145. ⟨ Scan one height 145 ⟩ ≡

```
e ← s;
while (*e) e++;    /* find end of string */
sscanf(s, "%04d", &minv);
printf("visibility␣%s%d␣%s\n", greater1 ? "greater␣than␣" : (less1 ? "less␣than␣" : ""), minv,
    strncmp(e − 2, "FT", 2) ≡ 0 ? "ft" : "m");
```

This code is used in section 143.

146. ⟨ Scan two heights 146 ⟩ ≡

```
e++;    /* skip past 'V' */
if (*e ≡ 'P') greater2++;
else if (*e ≡ 'M') less2++;
sscanf(s, "%04d", &minv);
if (isdigit(*e)) s ← e;
else s ← e + 1;
while (*e) e++;    /* Find end of string */
sscanf(s, "%04d", &maxv);
printf("visibility␣varying␣from␣%s%d␣to␣%s%d␣%s\n",
    greater1 ? "greater␣than␣" : (less1 ? "less␣than␣" : ""), minv,
    greater2 ? "greater␣than␣" : (less2 ? "less␣than␣" : ""), maxv, strncmp(e − 2, "FT",
    2) ≡ 0 ? "ft" : "m");
```

This code is used in section 143.

147. Present weather.

```
#define MAXWEATHER 5
```

⟨ METAR storage variables 69 ⟩ +≡

```
int wxcntr;
Cptr wx[MAXWEATHER];
```

148. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→wxcntr ← 0;
for (i ← 0; i < MAXWEATHER; dm→wx[i++] ← Λ) ;
```

149. ⟨ METAR deallocation statements 71 ⟩ +≡

```
for (i ← 0; i < dm→wxcntr; i++) free(dm→wx[i]);
```

150. A present weather group in a METAR report, though well-defined, is not necessarily easy to parse. Regular expressions are much better suited to the task, and so we'll build a regex in pieces.

```
#define WX_INTENSITY "(\\-|\\+|VC)?"
#define WX_DESCRIPTOR "(MI|PR|BC|DR|BL|FZ)?"
#define WX_SHOWTS "(TS|SH)"
#define WX_PRECIP "(DZ|RA|SN|SG|IC|PL|GR|GS|UP)"
#define WX_OBSCURE "(BR|FG|FU|VA|DU|SA|HZ|PY)"
#define WX_OTHER "(PO|SQ|FC|SS|DS)"
```

⟨ Includes 3 ⟩ +≡

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

151. The *wx_regex* sentence below should capture all legal combinations of "present weather" strings (I hope). For our purposes here, it is not strictly necessary to reject all illegal string combinations, so we can afford to be a little more liberal in what we accept. Our goal is to translate the string into natural language, not to validate statements for a computer compiler. However, it is important that the following expression does not match the empty string.

```
⟨ Global Variables 24 ⟩ +≡
  char *wx_regex ← WX_INTENSITY"(("WX_DESCRIPTOR"|"WX_SHOWTS")"
  "("WX_PRECIP"|"WX_OBSCURE"|"WX_OTHER"))+|"WX_SHOWTS;
```

152. We'll need to allocate a **regex_t** structure to store the compiled regex.

```
⟨ Global Variables 24 ⟩ +≡
  regex_t present_weather;
```

153. We'll also need to compile the regex string initially. This will only need to be done once for the run of the program. This operation is performed on a static string, so we do it early in the program initialization process. If an error occurs in compilation, we will just exit here since it reveals a flaw in the program.

```
⟨ Initialize wx_regex 153 ⟩ ≡
  {
    int regcomp_ret;
    char errbuf[512];
    if ((regcomp_ret ← regcomp(&present_weather, wx_regex, REG_EXTENDED)) ≠ 0) {
      regerror(regcomp_ret, &present_weather, errbuf, sizeof (errbuf));
      fprintf(stderr, "regcomp␣error:␣%s\n", errbuf);
      fprintf(stderr, "Bad␣regexp:␣'%s'\n", wx_regex);
      exit(3);
    }
  }
```

This code is used in section 30.

154.
```
⟨ Token-parsing states 76 ⟩ +≡
  TS_EXP_WX ,
```

155.
```
⟨ Token types 77 ⟩ +≡
  TOK_WX ,
```

156.
```
⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_WX:
  {
    int retval;
    char *tmpstr ← strndup(s, e − s);
    retval ← regexec(&present_weather, tmpstr, 0, Λ, 0);
    free(tmpstr);
    if (retval ≡ 0) return TOK_WX;
    parse_state++;
    goto ReTry;
  }
```

157.
```
⟨ Parse token according to type 79 ⟩ +≡
case TOK_WX:
  if (dm→wxcntr < MAXWEATHER) dm→wx[dm→wxcntr++] ← strndup(s, e − s);
  break;
```

158. Break down a present weather string into its constituent parts to translate them into English.

```
#define MAXP 5
#define MAXOBS 5
#define MAXOTHR 5
```

⟨ Output statements 72 ⟩ +≡

```
{
  char *s, *e;
  int shower, thunderstorm;
  Cptr precip[MAXP], obs[MAXOBS], other[MAXOTHR];
  Cptr descp[MAXP], descobs[MAXOBS], descoth[MAXOTHR], desc;
  Cptr intensity, vicinity;
  int np, nobs, nother, w;
  for (w ← 0; w < dm→wxcntr; w++) {
    printf("%s", w ? "; " : "WX: ");
    s ← dm→wx[w];
    ⟨ Reset quantities 159 ⟩
    ⟨ Check for tornado and continue 160 ⟩
    ⟨ Check for vicinity or intensity 161 ⟩
    ⟨ Check for thunderstorms 162 ⟩
    ⟨ Match precip, obscuration, or other 163 ⟩
    ⟨ Write present weather output 164 ⟩
  }
  if (w) printf("\n");
}
```

159.

```
#define EMPTY_STR ""
```

⟨ Reset quantities 159 ⟩ ≡

```
vicinity ← intensity ← EMPTY_STR;
shower ← thunderstorm ← 0;
np ← nobs ← nother ← 0;
```

This code is used in section 158.

160. This element should be stand-alone, and since it changes the character of funnel cloud, it's handled separately. If this token is recognized, the rest of the processing loop is short-circuited.

⟨ Check for tornado and continue 160 ⟩ ≡

```
if (strcmp(s, "+FC") ≡ 0) {
  printf("tornado or waterspout");
  continue;
}
```

This code is used in section 158.

161. ⟨ Check for vicinity or intensity 161 ⟩ ≡

```
if (strncmp(s, "VC", 2) ≡ 0) {
  vicinity ← "␣in␣the␣vicinity";
  s += 2;
} else if (*s ≡ '-') {
  intensity ← "light␣";
  s++;
} else if (*s ≡ '+') {
  intensity ← "heavy␣";
  s++;
}
```

This code is used in section 158.

162. ⟨ Check for thunderstorms 162 ⟩ ≡

```
if (strncmp(s, "TS", 2) ≡ 0) {
  thunderstorm++;
  s += 2;
}
```

This code is used in section 158.

163. A descriptor may precede any of the weather types, and so it should be attached to it here.

⟨ Match precip, obscuration, or other 163 ⟩ ≡

```
while (*s) {
  if (s_is_a(s, wx_desc)) {
    desc ← translate_s_to(s, wx_desc);
    s += 2;
  } else desc ← EMPTY_STR;
  if (*s) {
    if (s_is_a(s, wx_precip)) {
      descp[np] ← desc;
      precip[np++] ← translate_s_to(s, wx_precip);
    } else if (s_is_a(s, wx_obscur)) {
      descobs[nobs] ← desc;
      obs[nobs++] ← translate_s_to(s, wx_obscur);
    } else if (s_is_a(s, wx_other)) {
      descoth[nother] ← desc;
      other[nother++] ← translate_s_to(s, wx_other);
    } else fprintf(stderr, "Unrecognized␣present␣weather␣string:␣%.2s\n", s);
    s += 2;
  } else {    /* handle just "showers" with no precip description */
    if (strlen(desc)) {
      descp[np] ← (desc[0] ≡ '␣') ? desc + 1 : desc;    /* skip leading space */
      precip[np++] ← EMPTY_STR;
    }
  }
}
```

This code is used in section 158.

164. ⟨ Write present weather output 164 ⟩ ≡

```
printf("%s%s%s", intensity, thunderstorm ? "thunderstorm" : "",
    (thunderstorm ∧ (np ∨ nobs ∨ nother)) ? "␣with␣" : "");
for (i ← 0; i < np; i++) {
  if (i) printf(",␣");
  printf("%s%s%s", descp[i][0] ≡ '␣' ? "" : descp[i], precip[i],
  descp[i][0] ≡ '␣' ? descp[i] : "");
}
for (i ← 0; i < nobs; i++) {
  if (i ∨ np) printf(",␣");
  printf("%s%s%s", descobs[i][0] ≡ '␣' ? "" : descobs[i], obs[i],
  descobs[i][0] ≡ '␣' ? descobs[i] : "");
}
for (i ← 0; i < nother; i++) {
  if (i ∨ np ∨ nobs) printf(",␣");
  printf("%s%s%s", descoth[i][0] ≡ '␣' ? "" : descoth[i], other[i],
  descoth[i][0] ≡ '␣' ? descoth[i] : "");
}
printf("%s", vicinity);
```

This code is used in section 158.

165. Rather than write a bunch of similar functions that do the same sort of processing, I'll write the functions once, and they can operate on different data. A one-dimensional array will contain a 2-character string which should be matched in order to be translated into the string that follows.

⟨ Global Variables 24 ⟩ +≡

```
Cptr wx_desc[] ← {"MI", "shallow␣",
"PR", "partial␣",
"BC", "patches␣of␣",
"DR", "low␣drifting␣",
"BL", "blowing␣",
"FZ", "freezing␣",
"SH", "␣showers",
Λ, Λ};
```

166. ⟨ Global Variables 24 ⟩ +≡

```
Cptr wx_precip[] ← {"DZ", "drizzle",
"RA", "rain",
"SN", "snow",
"SG", "snow␣grains",
"IC", "ice␣crystals",
"PL", "ice␣pellets",
"GR", "hail",
"GS", "small␣hail␣and/or␣snow␣pellets",
"UP", "unknown␣precipitation",
Λ, Λ};
```

167. ⟨ Global Variables 24 ⟩ +≡

```
Cptr wx_obscur[] ← {"BR", "mist",
"FG", "fog",
"FU", "smoke",
"VA", "volcanic␣ash",
"DU", "widespread␣dust",
"SA", "sand",
"HZ", "haze",
"PY", "spray",
Λ, Λ};
```

168. ⟨ Global Variables 24 ⟩ +≡

```
Cptr wx_other[] ← {"PO", "well-developed␣dust/sand␣swirls",
"SQ", "squalls",
"FC", "funnel␣cloud",
"SS", "sand␣storm",
"DS", "dust␣storm",
Λ, Λ};
```

169. ⟨ Parse helpers 55 ⟩ +≡

```
int s_is_a(const char *s, Cptr d[])
{
  int i;
  for (i ← 0; d[i]; i += 2)
    if (strncmp(s, d[i], strlen(d[i])) ≡ 0) return TRUE;
  return FALSE;
}
```

170. ⟨ Parse helpers 55 ⟩ +≡

```
Cptr translate_s_to(const char *s, Cptr d[])
{
  int i;
  for (i ← 0; d[i]; i += 2)
    if (strncmp(s, d[i], strlen(d[i])) ≡ 0) return d[i + 1];
  return Λ;
}
```

171. Sky conditions.

```
#define MAXSKY 10
```

⟨ METAR storage variables 69 ⟩ +≡

```
int skycntr;
Cptr sky[MAXSKY];
```

172. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→skycntr ← 0;
for (i ← 0; i < MAXSKY; dm→sky[i++] ← Λ) ;
```

173. ⟨ METAR deallocation statements 71 ⟩ +≡

```
for (i ← 0; i < dm→skycntr; i++) free(dm→sky[i]);
```

174. ⟨ Token types 77 ⟩ +≡
TOK_SKY ,

175. ⟨ Token-parsing states 76 ⟩ +≡
TS_EXP_SKY ,

176. ⟨ Analyze current token according to state 78 ⟩ +≡
case TS_EXP_SKY:
if $((strncmp(s, \texttt{"VV"}, 2) \equiv 0) \vee (strncmp(s, \texttt{"SKC"}, 3) \equiv 0) \vee$
$(strncmp(s, \texttt{"CLR"}, 3) \equiv 0) \vee (strncmp(s, \texttt{"FEW"}, 3) \equiv 0) \vee$
$(strncmp(s, \texttt{"SCT"}, 3) \equiv 0) \vee (strncmp(s, \texttt{"BKN"}, 3) \equiv 0) \vee$
$(strncmp(s, \texttt{"OVC"}, 3) \equiv 0) \vee (strncmp(s, \texttt{"NSC"}, 3) \equiv 0) \vee$
$(strncmp(s, \texttt{"NCD"}, 3) \equiv 0) \vee (strncmp(s, \texttt{"CAVOK"}, 5) \equiv 0))$ **return** TOK_SKY;
$parse\_state{+}{+}$;
goto $ReTry$;

177. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_SKY:
if $(dm{\to}skycntr < \texttt{MAXSKY})$ $dm{\to}sky[dm{\to}skycntr{+}{+}] \leftarrow strndup(s, e - s)$;
break;

178. ⟨ Output statements 72 ⟩ +≡

```
{
  char *s, *e, *height_units ← (dm→pressure_type ≡ PT_HPA) ? "␣m" : "␣ft";
  int height;
  for (i ← 0; i < dm→skycntr; i++) {
    e ← s ← dm→sky[i];
    while (*e) e++;
    if (i) printf(",␣");
    else if (strncmp(s, "VV", 2) ≡ 0) printf("VertVis:␣");
    else printf("Skies:␣");
    if (strncmp(s, "VV", 2) ≡ 0) {
      sscanf(s + 2, "%03d", &height);
      printf("%d%s", height * 100, height_units);
    } else if (strncmp(s, "FEW", 3) ≡ 0) {
      sscanf(s + 3, "%03d", &height);
      printf("Few␣clouds␣at␣%d%s", height * 100, height_units);
    } else if (strncmp(s, "SCT", 3) ≡ 0) {
      sscanf(s + 3, "%03d", &height);
      printf("Scattered␣clouds␣at␣%d%s", height * 100, height_units);
    } else if (strncmp(s, "BKN", 3) ≡ 0) {
      sscanf(s + 3, "%03d", &height);
      printf("Broken␣clouds␣at␣%d%s", height * 100, height_units);
    } else if (strncmp(s, "OVC", 3) ≡ 0) {
      sscanf(s + 3, "%03d", &height);
      printf("Overcast␣at␣%d%s", height * 100, height_units);
    } else if (strncmp(s, "SKC", 3) ≡ 0 ∨ strncmp(s, "CLR", 3) ≡ 0 ∨ strncmp(s, "CAVOK", 5) ≡ 0)
      printf("Clear");
    else if (strncmp(s, "NSC", 3) ≡ 0) printf("No␣significant␣cloud␣cover");
    else if (strncmp(s, "NCD", 3) ≡ 0) printf("No␣clouds␣detected");
    if (strcmp(e − 2, "CB") ≡ 0) printf("␣with␣cumulonimbus");
    else if (strcmp(e − 3, "TCU") ≡ 0) printf("␣with␣towering␣cumulus");
  }
  if (i) printf("\n");
}
```

179. Temperature and Dewpoint.

⟨ METAR storage variables 69 ⟩ +≡

```
int temperature, dewpoint;
```

180. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→temperature ← dm→dewpoint ← FLAG_MISSING;
```

181. ⟨ Token types 77 ⟩ +≡

```
TOK_TEMPDEW ,
```

182. ⟨ Token-parsing states 76 ⟩ +≡

```
TS_EXP_TEMP ,
```

183. ⟨ Analyze current token according to state 78 ⟩ +≡

```
case TS_EXP_TEMP:
  parse_state++;
  if (*s ≡ 'M') s++;
  if (isdigit(*s) ∧ *(s + 2) ≡ '/') return TOK_TEMPDEW;
  goto ReTry;
```

184. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_TEMPDEW:
  if (*s ≡ 'M') {
    sscanf(s + 1, "%02d", &dm→temperature);
    dm→temperature ← −dm→temperature;
    s += 4;
  } else {
    sscanf(s, "%02d", &dm→temperature);
    s += 3;
  }
  if (*s ≡ 'M') {
    sscanf(s + 1, "%02d", &dm→dewpoint);
    dm→dewpoint ← −dm→dewpoint;
  } else if (isdigit(*s)) sscanf(s, "%02d", &dm→dewpoint);
  break;
```

185. If the higher precision temperature and dewpoint values are available from the remarks section, use those values instead. Provide Fahrenheit values as well as Centigrade.

```
#define C2F(x) (float)((((x) * 9.0)/5.0) + 32.0)
```

⟨ Output statements 72 ⟩ +≡

```
  if (dm→precise_temp ≡ FLAG_MISSING ∨ dm→precise_dew ≡ FLAG_MISSING) {
    if (dm→temperature ≠ FLAG_MISSING) {
      printf("TempF:␣%.1f\nTempC:␣%d\n", C2F(dm→temperature), dm→temperature);
      if (dm→dewpoint ≠ FLAG_MISSING)
        printf("DewF:␣%.1f\nDewC:␣%d\n", C2F(dm→dewpoint), dm→dewpoint);
    }
  }
```

186. Altimeter or Pressure.

⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ +≡

```
  enum PresUnit {
    PT_INMG, PT_HPA
  };
```

187. ⟨ METAR storage variables 69 ⟩ +≡

```
  float altimeter;
  enum PresUnit pressure_type;
```

188. Pressure units will not need to be initialized since the unit type is not needed until the actual value is parsed.

⟨ METAR initialization statements 70 ⟩ +≡

```
  dm→altimeter ← FLAG_MISSING;
```

189. ⟨ Token types 77 ⟩ +≡
```
TOK_ALTIMETER, TOK_PRESSURE ,
```

190. ⟨ Token-parsing states 76 ⟩ +≡
```
TS_EXP_ALT ,
```

191. ⟨ Analyze current token according to state 78 ⟩ +≡
```
case TS_EXP_ALT:
  parse_state++;
  if ((e − s) ≡ 5) {
    if (*s ≡ 'A') return TOK_ALTIMETER;
    if (*s ≡ 'Q') return TOK_PRESSURE;
  }
  goto ReTry;
```

192. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_ALTIMETER:
  sscanf(s + 1, "%f", &dm→altimeter);
  dm→altimeter /= 100.0;
  dm→pressure_type ← PT_INMG;
  break;
```

193. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_PRESSURE:
  sscanf(s + 1, "%f", &dm→altimeter);
  dm→pressure_type ← PT_HPA;
  break;
```

194. ⟨ Output statements 72 ⟩ +≡
```
  if (dm→altimeter ≠ FLAG_MISSING) {
    if (dm→pressure_type ≡ PT_INMG) printf("Alt:␣%.2f\"\n", dm→altimeter);
    else printf("Pressure:␣%.0f␣hPa\n", dm→altimeter);
  }
```

195. Unknown. Do any special processing or recovery for unrecognized tokens here. For now, we'll just copy the unrecognized tokens into another buffer.

⟨ METAR storage variables 69 ⟩ +≡
```
  Cptr unknown;
```

196. ⟨ METAR initialization statements 70 ⟩ +≡
```
  dm→unknown ← Λ;
```

197. ⟨ METAR deallocation statements 71 ⟩ +≡
```
  if (dm→unknown) free(dm→unknown);
```

198. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_UNKNOWN:
  if (unkcntr++) strcat(unkbuf, "␣");
  else unkbuf[0] ← '\0';
  strncat(unkbuf, s, e − s);
  break;
```

199. Remark token. For those remark elements that are not explicitly recognized, we'll keep the raw strings in storage here.

⟨ METAR storage variables 69 ⟩ +≡
  Cptr *remarks*;

200. ⟨ METAR initialization statements 70 ⟩ +≡
  dm→*remarks* ← Λ;

201. ⟨ METAR deallocation statements 71 ⟩ +≡
  if (*dm*→*remarks*) *free*(*dm*→*remarks*);

202. ⟨ Token types 77 ⟩ +≡
  `TOK_RMK` ,

203. ⟨ Token-parsing states 76 ⟩ +≡
  `TS_EXP_RMK` ,

204. Once we arrive at the state where we're expecting the remark token, we've reached the end of the line, and we won't advance parsing states until we actually receive the `'RMK'` token string. If we did not recognize the current token in the present state, we'll reset the search state and return `TOK_UNKNOWN`.

⟨ Analyze current token according to state 78 ⟩ +≡
case `TS_EXP_RMK`:
  if (*strncmp*(*s*, `"RMK"`, *e* − *s*) ≡ 0) {
    parse_state++;
    return `TOK_RMK`;
  }
  ⟨ Reset search state to the last known successful state 205 ⟩
  return `TOK_UNKNOWN`;

205. Reset the token search state to the last known successful search. As an example of how this works, suppose that I start off in a state searching for sky coverage tokens. If I receive a malformed token, I will keep advancing search states until I come to the `TS_EXP_RMK` state. At this point, if I have not matched the token, I now know that it is hopeless, and I return `TOK_UNKNOWN`. However, when called upon again, I wish to resume attempting to recognize tokens from the present state.

⟨ Reset search state to the last known successful state 205 ⟩ ≡
  parse_state ← *last_state*;

This code is used in section 204.

206. We don't need to store any information about the '`RMK`' token: its presence merely tells us to switch parsing states to recognize elements within the remarks section of the report. However, we'll use this opportunity to initialize the temporary remark storage.

⟨ Parse token according to type 79 ⟩ +≡
case `TOK_RMK`:
  rembuf[0] ← `'\0'`;
  break;

207. Remark elements. Many additional kinds of information may be encoded in the remarks section of a METAR report. Remarks or plain language text that is not explicitly parsed shall remain in the *remarks* character string.

⟨ Token types 77 ⟩ +≡
  `TOK_REMARKS` ,

208. ⟨ Token-parsing states 76 ⟩ +≡
TS_IN_RMK ,

209. If a particular token is not recognized by ⟨ Long remark token analysis 220 ⟩ or ⟨ Short remark token analysis 214 ⟩, then the generic token type TOK_REMARKS is returned. Long token markers are looked at first since some of the short token markers may begin with the some of the same characters.

⟨ Analyze current token according to state 78 ⟩ +≡
case TS_IN_RMK:
⟨ Long remark token analysis 220 ⟩
⟨ Short remark token analysis 214 ⟩
return TOK_REMARKS;

210. Add the remark token to the generic remarks buffer *rembuf*. These will be stored in the structure at the end of decoding.

⟨ Parse token according to type 79 ⟩ +≡
case TOK_REMARKS:
if (*remcntr*++) *strcat*(*rembuf*, "␣");
strncat(*rembuf*, *s*, *e* − *s*);
break;

211. Precise temperature and dewpoint.

⟨ Remark Storage 211 ⟩ ≡
float *precise_temp*, *precise_dew*;

See also sections 217, 224, 230, 241, 250, 269, 275, 281, 287, 295, 301, 308, 314, 322, 329, and 337.

This code is used in section 40.

212. ⟨ METAR initialization statements 70 ⟩ +≡
dm→*precise_temp* ← *dm*→*precise_dew* ← FLAG_MISSING;

213. ⟨ Token types 77 ⟩ +≡
TOK_PRECTEMP ,

214. ⟨ Short remark token analysis 214 ⟩ ≡
if (∗*s* ≡ 'T' ∧ (∗(*s* + 1) ≡ '0' ∨ ∗(*s* + 1) ≡ '1') ∧ ((*e* − *s*) ≡ 5 ∨ (*e* − *s*) ≡ 9)) **return** TOK_PRECTEMP;

See also sections 227, 233, 244, 272, 290, 326, and 333.

This code is cited in section 209.

This code is used in section 209.

215. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_PRECTEMP:
s++;
if ((*e* − *s*) ≡ 8) *sscanf*(*s*, "%04f%04f", &*dm*→*precise_temp*, &*dm*→*precise_dew*);
else *sscanf*(*s*, "%04f", &*dm*→*precise_temp*);
if (*dm*→*precise_temp* ≥ 1000.0) *dm*→*precise_temp* ← −(*dm*→*precise_temp* − 1000.0)/10.0;
else *dm*→*precise_temp* /= 10.0;
if (*dm*→*precise_dew* > FLAG_MISSING) {
if (*dm*→*precise_dew* ≥ 1000.0) *dm*→*precise_dew* ← −(*dm*→*precise_dew* − 1000.0)/10.0;
else *dm*→*precise_dew* /= 10.0;
}
break;

216. ⟨ Output statements 72 ⟩ +≡

```
if (dm→precise_temp ≠ FLAG_MISSING)
  printf("TempF:␣%.1f\nTempC:␣%.1f\n", C2F(dm→precise_temp), dm→precise_temp);
if (dm→precise_dew ≠ FLAG_MISSING)
  printf("DewF:␣%.1f\nDewC:␣%.1f\n", C2F(dm→precise_dew), dm→precise_dew);
```

217. Sea-level pressure.

⟨ Remark Storage 211 ⟩ +≡

```
float sealevelp;
```

218. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→sealevelp ← FLAG_MISSING;
```

219. ⟨ Token types 77 ⟩ +≡

```
TOK_SEALEVEL ,
```

220. Explicitly recognize the string `'SLPNO'` as a generic remark so that it does not interfere with the recognition of an actual sea-level pressure token.

⟨ Long remark token analysis 220 ⟩ ≡

```
if (strncmp(s, "SLPNO", e − s) ≡ 0) return TOK_REMARKS;
if (strncmp(s, "SLP", 3) ≡ 0) return TOK_SEALEVEL;
```

See also sections 254, 278, 284, 298, 304, 311, and 317.

This code is cited in section 209.

This code is used in section 209.

221. Translating from "`SLP`*nnn*" to an actual value for millibars, we need some help from the altimeter reading, since we only get the tens, units, and tenths from the "*nnn*" part. The conversion factor from inches of mercury to millibars is given by `IN2MB`. If an altimeter reading after conversion to millibars is very near an "edge" (e.g., 999.8 mb or 1000.1 mb), then the conversion may not work correctly. The SLP is compared to the altimeter reading and adjusted to the closest value.

```
#define IN2MB (33.8638815)
```

⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_SEALEVEL:
  {
    float altim, magnitude;
    if (dm→altimeter ≡ FLAG_MISSING) magnitude ← 1000.0;
    else {
      if (dm→pressure_type ≡ PT_INMG) altim ← dm→altimeter ∗ IN2MB;
      else altim ← dm→altimeter;
      magnitude ← truncf(altim/100.0) ∗ 100.0;
    }
    s += 3;
    sscanf(s, "%f", &dm→sealevelp);
    dm→sealevelp /= 10.0;
    dm→sealevelp += magnitude;
    if (fabs((double) altim − dm→sealevelp) > 30.0) {
      if (fabs((double) altim − (dm→sealevelp − 100.0)) ≤ 30.0) dm→sealevelp −= 100.0;
      else if (fabs((double) altim − (dm→sealevelp + 100.0)) ≤ 30.0) dm→sealevelp += 100.0;
    }
  }
  break;
```

222. ⟨ Output statements 72 ⟩ +≡
```
if (dm→sealevelp ≠ FLAG_MISSING) printf("SLP:␣%.1f␣hPa\n", dm→sealevelp);
```

223. Automatic station type.

⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ +≡
```
enum AutoType {
  ST_TYPE_NONE, ST_TYPE_A01, ST_TYPE_A02, ST_TYPE_UNK
};
```

224. ⟨ Remark Storage 211 ⟩ +≡
```
enum AutoType station_type;
```

225. ⟨ METAR initialization statements 70 ⟩ +≡
```
dm→station_type ← ST_TYPE_NONE;
```

226. ⟨ Token types 77 ⟩ +≡
```
TOK_AUTOTYPE ,
```

227. ⟨ Short remark token analysis 214 ⟩ +≡
```
if (*s ≡ 'A' ∧ (e − s) ≡ 3) return TOK_AUTOTYPE;
```

228. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_AUTOTYPE:
  if (*(s + 2) ≡ '1') dm→station_type ← ST_TYPE_A01;
  else if (*(s + 2) ≡ '2') dm→station_type ← ST_TYPE_A02;
  else dm→station_type ← ST_TYPE_UNK;
  break;
```

229. ⟨ Output statements 72 ⟩ +≡
```
if (dm→station_type ≠ ST_TYPE_NONE)
  printf("StType:␣%s\n", dm→station_type ≡ ST_TYPE_A01 ? "A01" : "A02");
```

230. Precipitation reports.

⟨ Remark Storage 211 ⟩ +≡
```
float precip_hour, precip36, precip24;
```

231. ⟨ METAR initialization statements 70 ⟩ +≡
```
dm→precip_hour ← dm→precip36 ← dm→precip24 ← FLAG_MISSING;
```

232. ⟨ Token types 77 ⟩ +≡
```
TOK_PRECIP, TOK_3OR6P, TOK_24P ,
```

233. ⟨ Short remark token analysis 214 ⟩ +≡
```
if (*s ≡ 'P' ∧ (e − s) ≡ 5) return TOK_PRECIP;
if (*s ≡ '6' ∧ (e − s) ≡ 5) return TOK_3OR6P;
if (*s ≡ '7' ∧ (e − s) ≡ 5) return TOK_24P;
```

234. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_PRECIP:
  sscanf(s + 1, "%04f", &dm→precip_hour);
  dm→precip_hour /= 100.0;
  break;
```

235. If the 24-hour precip group has been coded `'7////'`, then more than a trace has occurred, but the amount is uncertain. We'll set a special flag for that.

```
#define PRECIP_UNK -1.0
⟨ Parse token according to type 79 ⟩ +≡
case TOK_24P:
  if (*(s + 1) ≡ '/') dm→precip24 ← PRECIP_UNK;
  else {
    sscanf(s + 1, "%04f", &dm→precip24);
    dm→precip24 /= 100.0;
  }
  break;
```

236.
```
⟨ Parse token according to type 79 ⟩ +≡
case TOK_3OR6P:
  if (*(s + 1) ≡ '/') dm→precip36 ← PRECIP_UNK;
  else {
    sscanf(s + 1, "%04f", &dm→precip36);
    dm→precip36 /= 100.0;
  }
  break;
```

237.
```
⟨ Output statements 72 ⟩ +≡
  if (dm→precip_hour ≠ FLAG_MISSING) {
    if (dm→precip_hour ≡ 0.0) printf("PreHr:␣Trace␣hourly␣precipitation\n");
    else printf("PreHr:␣%.2f\"␣hourly␣precipitation\n", dm→precip_hour);
  }
```

238.
```
⟨ Output statements 72 ⟩ +≡
  if (dm→precip36 ≠ FLAG_MISSING) {
    if (dm→precip36 ≡ 0.0) printf("PreHr%d:␣Trace␣%d-hourly␣precipitation\n", hour6(dm→hour,
          dm→min) ? 6 : 3, hour6(dm→hour, dm→min) ? 6 : 3);
    else if (dm→precip36 ≡ PRECIP_UNK)
      printf("PreHr%d:␣Indeterminable␣%d-hourly␣precipitation\n", hour6(dm→hour,
          dm→min) ? 6 : 3, hour6(dm→hour, dm→min) ? 6 : 3);
    else printf("PreHr%d:␣%.2f\"␣%d-hourly␣precipitation\n", hour6(dm→hour, dm→min) ? 6 : 3,
          dm→precip36, hour6(dm→hour, dm→min) ? 6 : 3);
  }
```

239. Determine if the time reflects a 6-hourly update, plus or minus some tolerance in minutes (given by `MIN_TOL`).

```
#define MIN_PER_HR (60)
#define MIN_IN_6HRS (6 * MIN_PER_HR)    /* number of minutes in 6 hours */
#define MIN_TOL (15)    /* report time should be within ±MIN_TOL */
⟨ Function Definitions 4 ⟩ +≡
  inline int hour6(int h, int m)
  {
    int val ← h * MIN_PER_HR + m;
    if (((val + MIN_TOL) % MIN_IN_6HRS) ≤ (2 * MIN_TOL)) return TRUE;
    else return FALSE;
  }
```

240. ⟨ Output statements 72 ⟩ +≡

```
if (dm→precip24 ≠ FLAG_MISSING) {
  if (dm→precip24 ≡ 0.0) printf("PreHr24:␣Trace␣24-hourly␣precipitation\n");
  else if (dm→precip24 ≡ PRECIP_UNK) printf("Indeterminable␣24-hourly␣precipitation\n");
  else printf("PreHr24:␣%.2f\"␣24-hourly␣precipitation\n", dm→precip24);
}
```

241. Temperature min/max.

⟨ Remark Storage 211 ⟩ +≡

```
float temp_min6, temp_max6, temp_min24, temp_max24;
```

242. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→temp_min6 ← dm→temp_max6 ← FLAG_MISSING;
dm→temp_min24 ← dm→temp_max24 ← FLAG_MISSING;
```

243. ⟨ Token types 77 ⟩ +≡

```
TOK_MAXT6HR, TOK_MINT6HR, TOK_MINMAXT24HR ,
```

244. ⟨ Short remark token analysis 214 ⟩ +≡

```
if (*s ≡ '1' ∧ (e − s) ≡ 5) return TOK_MAXT6HR;
if (*s ≡ '2' ∧ (e − s) ≡ 5) return TOK_MINT6HR;
if (*s ≡ '4' ∧ (e − s) ≡ 9) return TOK_MINMAXT24HR;
```

245. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_MAXT6HR:
  sscanf(s + 1, "%04f", &dm→temp_max6);
  if (dm→temp_max6 ≥ 1000.0) dm→temp_max6 ← −(dm→temp_max6 − 1000.0);
  dm→temp_max6 /= 10.0;
  break;
```

246. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_MINT6HR:
  sscanf(s + 1, "%04f", &dm→temp_min6);
  if (dm→temp_min6 ≥ 1000.0) dm→temp_min6 ← −(dm→temp_min6 − 1000.0);
  dm→temp_min6 /= 10.0;
  break;
```

247. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_MINMAXT24HR:
  sscanf(s + 1, "%04f%04f", &dm→temp_min24, &dm→temp_max24);
  if (dm→temp_min24 ≥ 1000.0) dm→temp_min24 ← −(dm→temp_min24 − 1000.0);
  dm→temp_min24 /= 10.0;
  if (dm→temp_max24 ≥ 1000.0) dm→temp_max24 ← −(dm→temp_max24 − 1000.0);
  dm→temp_max24 /= 10.0;
  break;
```

248. ⟨ Output statements 72 ⟩ +≡

```
if (dm→temp_min6 ≠ FLAG_MISSING)
  printf("TempHr6min:␣%.1fC␣(%.1fF)\n", dm→temp_min6, C2F(dm→temp_min6));
if (dm→temp_max6 ≠ FLAG_MISSING)
  printf("TempHr6max:␣%.1fC␣(%.1fF)\n", dm→temp_max6, C2F(dm→temp_max6));
```

249. ⟨ Output statements 72 ⟩ +≡

```
if (dm→temp_min24 ≠ FLAG_MISSING)
  printf("TempHr24min:␣␣%.1fC␣(%.1fF)\n", dm→temp_min24, C2F(dm→temp_min24));
if (dm→temp_max24 ≠ FLAG_MISSING)
  printf("TempHr24max:␣/␣%.1fC␣(%.1fF)\n", dm→temp_max24, C2F(dm→temp_max24));
```

250. Visibility.

⟨ Remark Storage 211 ⟩ +≡

```
float tower_vis, surface_vis, asos_vis;
float prevail_vis_min, prevail_vis_max;
float sector_vis, vis_loc2;
Cptr sect_vis_dir, vis_loc2id;
```

251. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→tower_vis ← dm→surface_vis ← dm→asos_vis ← FLAG_MISSING;
dm→prevail_vis_min ← dm→prevail_vis_max ← FLAG_MISSING;
dm→sector_vis ← dm→vis_loc2 ← FLAG_MISSING;
dm→sect_vis_dir ← dm→vis_loc2id ← Λ;
```

252. ⟨ METAR deallocation statements 71 ⟩ +≡

```
if (dm→sect_vis_dir) free(dm→sect_vis_dir);
if (dm→vis_loc2id) free(dm→vis_loc2id);
```

253. ⟨ Token types 77 ⟩ +≡

```
TOK_TOWERVIS, TOK_SFCVIS, TOK_SECVIS, TOK_ASOSVIS, TOK_VARVIS ,
```

254. ⟨ Long remark token analysis 220 ⟩ +≡

```
if (strncmp(s, "TWR␣VIS", 7) ≡ 0) return TOK_TOWERVIS;
if (strncmp(s, "SFC␣VIS", 7) ≡ 0) return TOK_SFCVIS;
if (strncmp(s, "ASOS␣VIS", 8) ≡ 0) return TOK_ASOSVIS;
if (strncmp(s, "VIS", 3) ≡ 0) {
  if (isdigit(*(s + 4))) return TOK_VARVIS;
  else return TOK_SECVIS;
}
```

255. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_TOWERVIS:
  s += 8;
  dm→tower_vis ← parse_fraction(&s);
  e ← s;
  break;
```

256. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_SFCVIS:
  s += 8;
  dm→surface_vis ← parse_fraction(&s);
  e ← s;
  break;
```

257. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_ASOSVIS:
  s += 9;
  dm→asos_vis ← parse_fraction(&s);
  e ← s;
  break;
```

258. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_SECVIS:
  s += 4; e ← find_token_end(s);
  dm→sect_vis_dir ← strndup(s, e − s);
  s ← e + 1;
  dm→sector_vis ← parse_fraction(&s);
  e ← s;
  break;
```

259. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_VARVIS:
  s += 4;
  if (dm→prevail_vis_min ≡ FLAG_MISSING) {
    dm→prevail_vis_min ← parse_fraction(&s);
    if (*s ≠ 'V') {      /* 2nd location? */
      s++; e ← find_token_end(s);
      dm→vis_loc2 ← dm→prevail_vis_min;
      dm→vis_loc2id ← strndup(s, e − s);
      dm→prevail_vis_min ← FLAG_MISSING;
    } else {
      s++;
      dm→prevail_vis_max ← parse_fraction(&s);
      e ← s;
    }
  } else {      /* 2nd location? */
    dm→vis_loc2 ← parse_fraction(&s);
    s++;
    e ← find_token_end(s);
    dm→vis_loc2id ← strndup(e, e − s);
  }
  break;
```

260. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→prevail_vis_min ≠ FLAG_MISSING)
    printf("VisVar:␣Visibility␣varying␣from␣%g␣to␣%g␣%s\n", dm→prevail_vis_min,
        dm→prevail_vis_max, dm→prevail_vis_max > 1.0 ? "miles" : "mile");
```

261. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→sector_vis ≠ FLAG_MISSING) printf("VisSec:␣Visibility␣in␣sector(s)␣%s␣is␣%g␣%s\n",
        dm→sect_vis_dir, dm→sector_vis, dm→sector_vis > 1.0 ? "miles" : "mile");
```

262. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→asos_vis ≠ FLAG_MISSING) printf("VisASOS:␣ASOS␣visibility␣is␣%g␣%s\n", dm→asos_vis,
        dm→asos_vis > 1.0 ? "miles" : "mile");
```

263. ⟨ Output statements 72 ⟩ +≡

```
if (dm→surface_vis ≠ FLAG_MISSING) printf("VisSurf:␣Surface␣visibility␣is␣%g␣%s\n",
      dm→surface_vis, dm→surface_vis > 1.0 ? "miles" : "mile");
```

264. ⟨ Output statements 72 ⟩ +≡

```
if (dm→tower_vis ≠ FLAG_MISSING) printf("VisTow:␣Tower␣visibility␣is␣%g␣%s\n", dm→tower_vis,
      dm→tower_vis > 1.0 ? "miles" : "mile");
```

265. ⟨ Output statements 72 ⟩ +≡

```
if (dm→vis_loc2 ≠ FLAG_MISSING) printf("VisLoc2:␣Visibility␣at␣%s␣is␣%g␣%s\n", dm→vis_loc2id,
      dm→vis_loc2, dm→vis_loc2 > 1.0 ? "miles" : "mile");
```

266. This function will return true if the string starting at s looks like it may be a fraction (i.e., it contains a digit, a slash, and more digits).

⟨ Parse helpers 55 ⟩ +≡

```
inline int isfrac(const char *s)
{
  if (isdigit(*s) ∧ *(s + 1) ≡ '/' ∧ isdigit(*(s + 2)) ∧ *s ≠ '4') return TRUE;
  return FALSE;
}
```

267. Find the end of a number or fraction string.

⟨ Parse helpers 55 ⟩ +≡

```
inline const char *find_frac_end(const char *s)
{
  while (*s ∧ (isdigit(*s) ∨ *s ≡ '/')) s++;
  return s;
}
```

268. Visibility may contain fractional values. This function attempts to parse such values. The fractional value is returned, and the pointer *start* is updated to point to the character beyond the last-parsed value.

⟨ Parse helpers 55 ⟩ +≡

```
float parse_fraction(const char **start)
{
  const char *s ← *start, *e ← find_frac_end(*start);
  float whole, num, denom, value ← FLAG_MISSING;
  if (isfrac(e + 1)) {    /* whole num/denom */
    sscanf(s, "%f␣%f/%f", &whole, &num, &denom);
    value ← whole + (num/denom);
    e ← find_frac_end(e + 1);
  } else if (isfrac(s)) {    /* num/denom */
    sscanf(s, "%f/%f", &num, &denom);
    value ← num/denom;
    e ← find_frac_end(s);
  } else if (isdigit(*s)) {    /* whole */
    sscanf(s, "%f", &value);
    e ← find_frac_end(s);
  }
  *start ← e;
  return value;
}
```

269. Snow depth.

⟨ Remark Storage 211 ⟩ +≡
```
int snow_depth;
```

270. ⟨ METAR initialization statements 70 ⟩ +≡
```
dm→snow_depth ← FLAG_MISSING;
```

271. ⟨ Token types 77 ⟩ +≡
```
TOK_SNOWDPTH ,
```

272. ⟨ Short remark token analysis 214 ⟩ +≡
```
if (*s ≡ '4' ∧ *(s + 1) ≡ '/') return TOK_SNOWDPTH;
```

273. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_SNOWDPTH:
  sscanf(s + 2, "%03d", &dm→snow_depth);
  break;
```

274. ⟨ Output statements 72 ⟩ +≡
```
if (dm→snow_depth ≠ FLAG_MISSING)
  printf("SnowDp:␣Snow␣on␣the␣ground:␣%d␣in\n", dm→snow_depth);
```

275. Snow increasing rapidly.

⟨ Remark Storage 211 ⟩ +≡
```
int snowinc_hour, snowinc_ground;
```

276. ⟨ METAR initialization statements 70 ⟩ +≡
```
dm→snowinc_hour ← dm→snowinc_ground ← FLAG_MISSING;
```

277. ⟨ Token types 77 ⟩ +≡
```
TOK_SNINCR ,
```

278. ⟨ Long remark token analysis 220 ⟩ +≡
```
if (strncmp(s, "SNINCR", (int)(e − s)) ≡ 0) return TOK_SNINCR;
```

279. ⟨ Parse token according to type 79 ⟩ +≡
```
case TOK_SNINCR:
  s += 7;
  e ← find_token_end(s);
  sscanf(s, "%d/%d", &dm→snowinc_hour, &dm→snowinc_ground);
  break;
```

280. ⟨ Output statements 72 ⟩ +≡
```
if (dm→snowinc_hour ≠ FLAG_MISSING)
  printf("SnowInc:␣Snow␣increasing␣rapidly:␣past␣hour␣%d␣in;␣on␣ground␣%d␣in\n",
      dm→snowinc_hour, dm→snowinc_ground);
```

281. Water equivalent.

⟨ Remark Storage 211 ⟩ +≡
```
float water_equiv;
```

282. ⟨ METAR initialization statements 70 ⟩ +≡
dm→water_equiv ← `FLAG_MISSING`;

283. ⟨ Token types 77 ⟩ +≡
`TOK_H2OEQUIV` ,

284. ⟨ Long remark token analysis 220 ⟩ +≡
if (*strncmp*(*s*, `"933"`, 3) ≡ 0) **return** `TOK_H2OEQUIV`;

285. ⟨ Parse token according to type 79 ⟩ +≡
case `TOK_H2OEQUIV`:
sscanf(*s* + 3, `"%f"`, &*dm→water_equiv*);
dm→water_equiv /= 10.0;
break;

286. ⟨ Output statements 72 ⟩ +≡
if (*dm→water_equiv* ≠ `FLAG_MISSING`)
printf(`"H2Oequiv:␣Water␣equivalent␣of␣snow:␣%.1f\n"`, *dm→water_equiv*);

287. Pressure tendency.
⟨ Remark Storage 211 ⟩ +≡
float *press_tend_amt*;
int *press_character*;

288. ⟨ METAR initialization statements 70 ⟩ +≡
dm→press_tend_amt ← `FLAG_MISSING`;
dm→press_character ← `FLAG_MISSING`;

289. ⟨ Token types 77 ⟩ +≡
`TOK_3HRPRTEND` ,

290. ⟨ Short remark token analysis 214 ⟩ +≡
if (∗*s* ≡ `'5'` ∧ (*e* − *s*) ≡ 5) **return** `TOK_3HRPRTEND`;

291. ⟨ Parse token according to type 79 ⟩ +≡
case `TOK_3HRPRTEND`:
if (*isdigit*(∗(*s* + 1))) *dm→press_character* ← (∗(*s* + 1) − `'0'`);
sscanf(*s* + 2, `"%03f"`, &*dm→press_tend_amt*);
dm→press_tend_amt /= 10.0;
break;

292. ⟨ Output statements 72 ⟩ +≡

```
if (dm→press_character ≠ FLAG_MISSING) {
  Cptr ptstr[] ← {"Increasing,␣then␣decreasing",
  "Increasing,␣then␣steady;␣or"    /* concat */
  "␣increasing␣then␣increasing␣more␣slowly",
  "Increasing␣steadily␣or␣unsteadily",
  "Decreasing␣or␣steady,␣then␣increasing;␣or"    /* concat */
  "␣increasing␣then␣increasing␣more␣rapidly",
  "Steady",
  "Decreasing,␣then␣increasing",
  "Decreasing,␣then␣steady;␣or"    /* concat */
  "␣decreasing␣then␣decreasing␣more␣slowly",
  "Decreasing␣steadily␣or␣unsteadily",
  "Steady␣or␣increasing,␣then␣decreasing;␣or"    /* concat */
  "␣decreasing␣then␣decreasing␣more␣rapidly",
  "", Λ};
  printf("Pres3hr:␣change␣of␣%s%.1f␣hPa;␣%s\n",
      near_zero(dm→press_tend_amt) ? "" : ((dm→press_character < 4) ? "+" : "-"), dm→press_tend_amt,
      ptstr[dm→press_character]);
}
```

293. Return TRUE if a floating-point value is within a small tolerance of 0.0. We're generally dealing with just tenths, so the tolerance can be fairly liberal.

#define F_EPSILON $1.0 \cdot 10^{-4}$

⟨ Function Definitions 4 ⟩ +≡

```
int near_zero(float f)
{
  if (fabs(f) < F_EPSILON) return TRUE;
  return FALSE;
}
```

294. ⟨ Forward Defs 13 ⟩ +≡

```
int near_zero(float);
```

295. Peak wind.

⟨ Remark Storage 211 ⟩ +≡

```
int peak_wnd_dir, peak_wnd_spd, peak_wnd_hr, peak_wnd_min;
```

296. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→peak_wnd_dir ← dm→peak_wnd_spd ← dm→peak_wnd_hr ← dm→peak_wnd_min ← FLAG_MISSING;
```

297. ⟨ Token types 77 ⟩ +≡

```
TOK_PKWND ,
```

298. ⟨ Long remark token analysis 220 ⟩ +≡

```
if (strncmp(s, "PK␣WND", 6) ≡ 0) return TOK_PKWND;
```

299. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_PKWND:
  s += 7;
  e ← find_token_end(s);
  sscanf(s, "%03d%d", &dm→peak_wnd_dir, &dm→peak_wnd_spd);
  s += 5;
  if (*s ≠ '/') s++;
  s++;
  if (e − s ≡ 2) {
    sscanf(s, "%02d", &dm→peak_wnd_min);
    dm→peak_wnd_hr ← dm→hour;
  } else sscanf(s, "%02d%02d", &dm→peak_wnd_hr, &dm→peak_wnd_min);
  break;
```

300. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→peak_wnd_spd ≠ FLAG_MISSING)
    printf("PkWind:␣from␣the␣%s␣at␣%d␣kts␣at␣%02d:%02d␣Z\n", sixteenpt_dir(dm→peak_wnd_dir),
        dm→peak_wnd_spd, dm→peak_wnd_hr, dm→peak_wnd_min);
```

301. Wind shift.

⟨ Remark Storage 211 ⟩ +≡

```
  int wshft_hr, wshft_min, wshft_front;
```

302. ⟨ METAR initialization statements 70 ⟩ +≡

```
  dm→wshft_hr ← dm→wshft_min ← FLAG_MISSING;
  dm→wshft_front ← FALSE;
```

303. ⟨ Token types 77 ⟩ +≡

```
  TOK_WSHFT ,
```

304. ⟨ Long remark token analysis 220 ⟩ +≡

```
  if (strncmp(s, "WSHFT", 5) ≡ 0) return TOK_WSHFT;
```

305. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_WSHFT:
  s += 6;
  e ← find_token_end(s);
  if (e − s ≡ 2) {
    sscanf(s, "%02d", &dm→wshft_min);
    dm→wshft_hr ← dm→hour;
  } else sscanf(s, "%02d%02d", &dm→wshft_hr, &dm→wshft_min);
  if (strncmp(e + 1, "FROPA", 5) ≡ 0) {
    dm→wshft_front ← TRUE;
    e ← find_token_end(e);
  }
  break;
```

306. ⟨ Output statements 72 ⟩ +≡

```
  if (dm→wshft_hr ≠ FLAG_MISSING) printf("WindShft:␣at␣%02d:%02d␣Z%s\n", dm→wshft_hr,
        dm→wshft_min, dm→wshft_front ? "␣(front␣passage)" : "");
```

307. Pressure tendency.

⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ +≡

```
enum PressStat {
  PST_NONE, PST_RAPRISE, PST_RAPFALL
};
```

308. ⟨ Remark Storage 211 ⟩ +≡

```
enum PressStat press_stat;
```

309. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→press_stat ← PST_NONE;
```

310. ⟨ Token types 77 ⟩ +≡

```
TOK_PRESSTEND ,
```

311. ⟨ Long remark token analysis 220 ⟩ +≡

```
if (strncmp(s, "PRES", 4) ≡ 0) return TOK_PRESSTEND;
```

312. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_PRESSTEND:
  if (strncmp(e − 2, "RR", 2) ≡ 0) dm→press_stat ← PST_RAPRISE;
  else if (strncmp(e − 2, "FR", 2) ≡ 0) dm→press_stat ← PST_RAPFALL;
  break;
```

313. ⟨ Output statements 72 ⟩ +≡

```
if (dm→press_stat ≠ PST_NONE)
  printf("PresTend:␣rapidly␣%s\n", dm→press_stat ≡ PST_RAPRISE ? "rising" : "falling");
```

314. Variable ceiling height and second location.

⟨ Remark Storage 211 ⟩ +≡

```
int ceil_ht_min, ceil_ht_max, ceil_ht2;
Cptr ceil_ht2_loc;
```

315. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→ceil_ht_min ← dm→ceil_ht_max ← dm→ceil_ht2 ← FLAG_MISSING;
dm→ceil_ht2_loc ← Λ;
```

316. ⟨ Token types 77 ⟩ +≡

```
TOK_VARCEIL, TOK_CEIL2 ,
```

317. ⟨ Long remark token analysis 220 ⟩ +≡

```
if (strncmp(s, "CIG", 3) ≡ 0) {
  e ← find_token_end(s + 4);
  if (e − s < 11) return TOK_CEIL2;
  return TOK_VARCEIL;
}
```

318. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_VARCEIL:
  sscanf(*s* + 4, "%03dV%03d", &*dm*→*ceil_ht_min*, &*dm*→*ceil_ht_max*);
  dm→*ceil_ht_min* ∗= 100;
  dm→*ceil_ht_max* ∗= 100;
  e ← *find_token_end*(*s* + 4);
  break;

319. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_CEIL2:
  sscanf(*s* + 4, "%03d", &*dm*→*ceil_ht2*);
  dm→*ceil_ht2* ∗= 100;
  s ← *advance_token*(*find_token_end*(*s* + 4));
  e ← *find_token_end*(*s*);
  dm→*ceil_ht2_loc* ← *strndup*(*s*, *e* − *s*);
  break;

320. ⟨ Output statements 72 ⟩ +≡
  if (*dm*→*ceil_ht_min* ≠ FLAG_MISSING)
    printf("VarCeil:␣from␣%d␣to␣%d␣ft\n", *dm*→*ceil_ht_min*, *dm*→*ceil_ht_max*);

321. ⟨ Output statements 72 ⟩ +≡
  if (*dm*→*ceil_ht2* ≠ FLAG_MISSING) *printf*("SectCeil:␣at␣%s␣%d␣ft\n", *dm*→*ceil_ht2_loc*, *dm*→*ceil_ht2*);

322. Beginning/Ending precipitation events.

⟨ Remark Storage 211 ⟩ +≡
  Cptr *precip_begend*;

323. ⟨ METAR initialization statements 70 ⟩ +≡
  dm→*precip_begend* ← Λ;

324. ⟨ METAR deallocation statements 71 ⟩ +≡
  if (*dm*→*precip_begend*) *free*(*dm*→*precip_begend*);

325. ⟨ Token types 77 ⟩ +≡
  TOK_PEVENT_BE ,

326. This token type is recognized in the short section so that the longer token types have a shot first.

⟨ Short remark token analysis 214 ⟩ +≡
  if ((*s_is_a*(*s*, *wx_precip*) ∧ (∗(*s* + 2) ≡ 'B' ∨ ∗(*s* + 2) ≡ 'E')) ∨ (*strncmp*(*s*, "SH", 2) ≡ 0 ∧ *s_is_a*(*s* + 2, *wx_precip*)) ∨ (*s_is_a*(*s*, *wx_desc*) ∧ *s_is_a*(*s* + 2, *wx_precip*))) **return** TOK_PEVENT_BE;

327. ⟨ Parse token according to type 79 ⟩ +≡
case TOK_PEVENT_BE:
  dm→*precip_begend* ← *strndup*(*s*, *e* − *s*);
  break;

328. ⟨ Output statements 72 ⟩ +≡

```
if (dm→precip_begend) {
  Cptr s ← dm→precip_begend;
  int hr, min, written ← 0;
  printf("Events:␣");
  while (*s) {
    if (written++) printf(";␣");
    if (strncmp(s, "SH", 2) ≡ 0) {
      printf("%s␣showers", translate_s_to(s + 2, wx_precip));
      s += 4;
    } else if (s_is_a(s, wx_desc)) {
      printf("%s", translate_s_to(s, wx_desc));
      s += 2;
      if (s_is_a(s, wx_precip)) printf("%s", translate_s_to(s, wx_precip));
      else if (s_is_a(s, wx_obscur)) printf("%s", translate_s_to(s, wx_obscur));
      else if (s_is_a(s, wx_other)) printf("%s", translate_s_to(s, wx_other));
      s += 2;
    } else {
      if (s_is_a(s, wx_precip)) printf("%s", translate_s_to(s, wx_precip));
      else if (s_is_a(s, wx_obscur)) printf("%s", translate_s_to(s, wx_obscur));
      else if (s_is_a(s, wx_other)) printf("%s", translate_s_to(s, wx_other));
      s += 2;
    }
    if (*s ≡ 'B') printf("␣began␣at␣");
    else if (*s ≡ 'E') printf("␣ended␣at␣");
  ScanTime: s++;
    if (isdigit(*(s + 2))) {    /* 4-digit time */
      sscanf(s, "%02d%02d", &hr, &min);
      s += 4;
    } else {    /* 2-digit time */
      sscanf(s, "%02d", &min);
      hr ← dm→hour;
      s += 2;
    }
    printf("%02d:%02d␣Z", hr, min);
    if (*s ≡ 'B') {
      printf(",␣began␣at␣");
      goto ScanTime;
    } else if (*s ≡ 'E') {
      printf(",␣ended␣at␣");
      goto ScanTime;
    }
  }
  printf("\n");
}
```

329. Thunderstorm begin/end.

⟨ Remark Storage 211 ⟩ +≡

```
Cptr tsbegend;
```

330. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→tsbegend ← Λ;
```

331. ⟨ METAR deallocation statements 71 ⟩ +≡

```
if (dm→tsbegend) free(dm→tsbegend);
```

332. ⟨ Token types 77 ⟩ +≡

```
TOK_TSBEGEND ,
```

333. ⟨ Short remark token analysis 214 ⟩ +≡

```
if (strncmp(s, "TS", 2) ≡ 0 ∧ (*(s + 2) ≡ 'B' ∨ *(s + 2) ≡ 'E')) return TOK_TSBEGEND;
```

334. ⟨ Parse token according to type 79 ⟩ +≡

```
case TOK_TSBEGEND:
  dm→tsbegend ← strndup(s, e − s);
  break;
```

335. ⟨ Output statements 72 ⟩ +≡

```
if (dm→tsbegend) {
  Cptr s ← dm→tsbegend + 2;
  int hr, min;
  printf("TStorm:␣");
  if (*s ≡ 'B') printf("␣began␣at␣");
  else if (*s ≡ 'E') printf("␣ended␣at␣");
RescanTime: s++;
  if (isdigit(*(s + 2))) {    /* 4-digit time */
    sscanf(s, "%02d%02d", &hr, &min);
    s += 4;
  } else {    /* 2-digit time */
    sscanf(s, "%02d", &min);
    hr ← dm→hour;
    s += 2;
  }
  printf("%02d:%02d␣Z", hr, min);
  if (*s ≡ 'E') {
    printf(",␣ended␣at␣");
    goto RescanTime;
  }
  printf("\n");
}
```

336. Other remarks.

⟨ **enum**s and **typedef**s for certain METAR values 41 ⟩ +≡

```
enum TornAct {
  TACT_NONE, TACT_TORNADO, TACT_FUNNEL, TACT_WATERSP
};
```

337. ⟨ Remark Storage 211 ⟩ +≡

```
#if 0
  enum TornAct tornadic_act;
  char *light_freq, *light_type, *light_loc;
  char *ts_begend, *ts_loc, *ts_movdir;
  float hail_sz;
  char *ob_type, *ob_cover;
  int ob_height;
  char *vsky_type_a, *vsky_type_b;
  int vsky_ht;
#endif
```

338. Derived Information. Not all of the information we will need is included within the report itself. We will supplement with useful information from other sources.

339. Timestamp. Using time of the file in which the report was located, adjust the time and date according to the parsed information from the report to create a new UNIX-style timestamp. This makes it easier to translate the report time/date into other timezones later.

⟨ METAR storage variables 69 ⟩ +≡

```
  time_t timestamp;
```

340. ⟨ METAR initialization statements 70 ⟩ +≡

```
  dm→timestamp ← 0;
```

341. ⟨ Retrieve info from other sources 341 ⟩ ≡

```
  dm→timestamp ← adjust_timestamp(t, dm→day, dm→hour, dm→min);
```

See also sections 347, 357, 364, 369, 374, and 380.

This code is used in section 58.

342. The *timegm*( )function is available in the GNU C library. It converts a **tm** structure at "Greenwich Mean Time" into a UNIX timestamp (the inverse of the *gmtime*( ) function). If this function is not available, the *mktime*( ) function could be used to convert the structure to a timestamp, though the local time zone would need to be reset by manipulating the `TZ` environment variable.

⟨ Function Definitions 4 ⟩ +≡

```
  time_t adjust_timestamp(time_t *t, int day, int hour, int min)
  {
    struct tm *ztime ← gmtime(t);
    if (ztime→tm_mday ≠ day) fprintf(stderr, "Day␣of␣METAR␣report␣does␣not␣match␣day␣\
          of␣file:\n""␣␣File␣day␣=␣%02d,␣Report␣day␣=␣%02d\n", ztime→tm_mday, day);
    ztime→tm_mday ← day;
    ztime→tm_hour ← hour;
    ztime→tm_min ← min;
    ztime→tm_sec ← 0;
    return timegm(ztime);
  }
```

343. ⟨ Forward Defs 13 ⟩ +≡

```
  time_t adjust_timestamp(time_t *, int, int, int);
```

344. Location and timezone. Look up the 4-character station ID from the GDB file and retrieve the latitude, longitude, timezone, and city information.

```
⟨ METAR storage variables 69 ⟩ +≡
  float lat, lon, elev;
  char *city, *region, *country, *tz;
```

345.
```
⟨ METAR initialization statements 70 ⟩ +≡
  dm→lat ← dm→lon ← dm→elev ← FLAG_MISSING;
  dm→city ← dm→region ← dm→country ← dm→tz ← Λ;
```

346.
```
⟨ METAR deallocation statements 71 ⟩ +≡
  if (dm→city) free(dm→city);
  if (dm→region) free(dm→region);
  if (dm→country) free(dm→country);
  if (dm→tz) free(dm→tz);
```

347.
```
⟨ Retrieve info from other sources 341 ⟩ +≡
  get_station_data(dm→station_id, dm);
```

348.
```
⟨ Output statements 72 ⟩ +≡
  if (dm→lat ≠ FLAG_MISSING) printf("StLat:␣%g\n", dm→lat);
  if (dm→lon ≠ FLAG_MISSING) printf("StLon:␣%g\n", dm→lon);
  if (dm→elev ≠ FLAG_MISSING) printf("StElev:␣%g\n", dm→elev);
  if (dm→city) printf("StCity:␣%s\n", dm→city);
  if (dm→region) printf("StRegion:␣%s\n", dm→region);
  if (dm→country) printf("StCountry:␣%s\n", dm→country);
  if (dm→tz) printf("StTZ:␣%s\n", dm→tz);
```

349. Look up the station and unpack the values.

```
⟨ Function Definitions 4 ⟩ +≡
  void get_station_data(char *stid, DMptr dm)
  {
    datum key, value;
    key.dptr ← stid;
    key.dsize ← strlen(stid);
    value ← gdbm_fetch(gdbf, key);
    if (value.dptr) {
      unpack_values(value.dptr, &(dm→lat), &(dm→lon), &(dm→elev), &(dm→city), &(dm→region),
          &(dm→country), &(dm→tz));
      free(value.dptr);
    }
  }
```

350. Unpack the values from the string p and store them in the passed parameters. Fields within the string are arranged in a certain order, fixed-size fields first: latitude, longitude, elevation, ICAO (skipped), WMO (skipped), country. The variable-sized fields follow, with tab separations: region, city, station name (skipped), natid (skipped), and timezone.

See the program `parse-master` for details on how the fields have been stored. This program, `parse-master`, and the file generated by `parse-master` should all be compiled/created on the same architecture.

```
#define Assign(x, s, e)
        while (*e ∧ *e ≠ '\t') e++;
        *e ← '\0';
        x ← strdup(s); e ← (s ← e + 1)
#define SkipField(s, e)
        while (*e ∧ *e ≠ '\t') e++;
        e ← (s ← e + 1)
```

⟨ Function Definitions 4 ⟩ +≡

```
void unpack_values(char *p, float *lat, float *lon, float *elev, char **city, char **region, char
        **country, char **tz)
{
  char *s, *e;
  memcpy(lat, p, sizeof(float));
  memcpy(lon, &(p[sizeof(float)]), sizeof(float));
  memcpy(elev, &(p[sizeof(float) * 2]), sizeof(float));
  *country ← strndup(&(p[sizeof(float) * 3 + 9]), 2);
  s ← e ← &(p[sizeof(float) * 3 + 11]);
  Assign(*region, s, e);
  Assign(*city, s, e);
  SkipField(s, e);    /* station name */
  SkipField(s, e);    /* natid */
  Assign(*tz, s, e);
  ⟨ Shorten city element 351 ⟩
}
```

351. The *city* element may contain additional information following a pipe character. We don't need this info, so shorten truncate the string.

⟨ Shorten *city* element 351 ⟩ ≡

```
s ← *city;
while (*s ∧ *s ≠ '|') s++;
*s ← '\0';
```

This code is used in section 350.

352. ⟨ Forward Defs 13 ⟩ +≡

```
void get_station_data(char *, DMptr);
void unpack_values(char *, float *, float *, float *, char **, char **, char **, char **);
```

353. Humidity. If we have dewpoint and temperature, we can calculate the relative humidity. Relative humidity (RH) can be defined as follows:

$$RH = 100\frac{e}{e_s},$$

where e is the actual water vapor pressure and e_s is the equilibrium (or saturation) vapor pressure over a plane of water.

Lawrence provides a formula for the dewpoint temperature (t_d) in terms of relative humidity and ambient temperature (t):

$$t_d = \frac{B\left[\ln\left(\frac{RH}{100}\right) + \frac{At}{B+t}\right]}{A - \ln\left(\frac{RH}{100}\right) - \frac{At}{B+t}}$$

Rearranging to solve for RH gives us

$$RH = 100\,\exp\left[A\left(\frac{t_d}{B+t_d} - \frac{t}{B+t}\right)\right].$$

The constants $A = 17.625$ and $B = 243.04°$ C are due to Alduchov and Eskridge, applied to the Magnus (et al.) formulation for vapor pressure. The temperatures are expressed in °C. And now we can create a function.

```
⟨ Function Definitions 4 ⟩ +≡
  double relative_humidity(float dew, float temp)
  {
    static double A ← 17.625, B ← 243.04;
    return exp(A * ((dew/(B + dew)) − (temp/(B + temp))));
  }
```

354.
```
⟨ Forward Defs 13 ⟩ +≡
  double relative_humidity(float, float);
```

355.
```
⟨ METAR storage variables 69 ⟩ +≡
  float humidity;
```

356.
```
⟨ METAR initialization statements 70 ⟩ +≡
  dm→humidity ← FLAG_MISSING;
```

357.
```
⟨ Retrieve info from other sources 341 ⟩ +≡
  if (dm→precise_dew ≠ FLAG_MISSING)
    dm→humidity ← relative_humidity(dm→precise_dew, dm→precise_temp);
  else if (dm→dewpoint ≠ FLAG_MISSING)
    dm→humidity ← relative_humidity((float) dm→dewpoint, (float) dm→temperature);
```

358.
```
⟨ Output statements 72 ⟩ +≡
  if (dm→humidity ≠ FLAG_MISSING) printf("Humidity:␣%.0f%%\n", dm→humidity * 100.0);
```

359. Local date and time. If we have a timezone for a station, we can generate a local date and time string from the report time.

```
⟨ METAR storage variables 69 ⟩ +≡
  char *loctime;
```

360. ⟨ METAR initialization statements 70 ⟩ +≡
```
dm→loctime ← Λ;
```

361. ⟨ METAR deallocation statements 71 ⟩ +≡
```
if (dm→loctime) free(dm→loctime);
```

362. In order to convert the timestamp to the local timezone of the reporting station, we have to change the timezone environment variable "TZ", do the conversion, and then restore the previous value of the environment variable. If $tz \equiv \Lambda$, we instead set the environment variable to an empty string, which gives us a time translation to UTC, which is just fine here.

⟨ Function Definitions 4 ⟩ +≡
```
char *make_loctime(time_t t, char *tz, int dodate)
{
  struct tm breakout;
  char *oldtz, tstr[128];
  oldtz ← getenv("TZ");
  if (tz) setenv("TZ", tz, 1);
  else setenv("TZ", "", 1);
  tzset();
  localtime_r(&t, &breakout);
  if (dodate) strftime(tstr, sizeof (tstr), "%-l:%M%P␣%Z␣%a␣%-e␣%b␣%Y", &breakout);
  else strftime(tstr, sizeof (tstr), "%-l:%M%P␣%Z", &breakout);
  if (oldtz) setenv("TZ", oldtz, 1);
  else unsetenv("TZ");
  tzset();
  return strdup(tstr);
}
```

363. ⟨ Forward Defs 13 ⟩ +≡
```
char *make_loctime(time_t, char *, int);
```

364. If we don't have a local timezone for the station, it's OK to pass along a Λ pointer.

⟨ Retrieve info from other sources 341 ⟩ +≡
```
if (dm→timestamp) dm→loctime ← make_loctime(dm→timestamp, dm→tz, TRUE);
```

365. ⟨ Output statements 72 ⟩ +≡
```
if (dm→loctime) printf("LocTime:␣%s\n", dm→loctime);
```

366. Local Sunrise/Sunset. Here we'll calculate the local sunrise and sunset time for current station and calendar day.

⟨ METAR storage variables 69 ⟩ +≡
```
time_t sunrise, sunset;
```

367. ⟨ METAR initialization statements 70 ⟩ +≡
```
dm→sunrise ← dm→sunset ← 0;
```

368. We'll need the header file for the sunrise-sunset library:

⟨ Includes 3 ⟩ +≡
```
#include <srss.h>
```

369. ⟨ Retrieve info from other sources 341 ⟩ +≡

```
if (dm→lat ≠ FLAG_MISSING ∧ dm→lon ≠ FLAG_MISSING ∧ dm→timestamp ∧ dm→tz)
  calcSRSS(&dm→timestamp, dm→lat, dm→lon, dm→tz, &dm→sunrise, &dm→sunset);
```

370. ⟨ Output statements 72 ⟩ +≡

```
{
  char *tempstr;
  if (dm→sunrise) {
    tempstr ← make_loctime(dm→sunrise, dm→tz, FALSE);
    printf("Sunrise:␣%s\n", tempstr);
    free(tempstr);
  }
  if (dm→sunset) {
    tempstr ← make_loctime(dm→sunset, dm→tz, FALSE);
    printf("Sunset:␣%s\n", tempstr);
    free(tempstr);
  }
}
```

371. Moonphase.

⟨ METAR storage variables 69 ⟩ +≡

```
double mphase, mage, mfrac;
```

372. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→mphase ← dm→mage ← dm→mfrac ← FLAG_MISSING;
```

373. We need the header file to calculate moon phase.

⟨ Includes 3 ⟩ +≡

```
#include <moonphase.h>
```

374. ⟨ Retrieve info from other sources 341 ⟩ +≡

```
if (dm→timestamp) moonphase(&dm→timestamp, &dm→mphase, &dm→mage, &dm→mfrac);
```

375. ⟨ Output statements 72 ⟩ +≡

```
if (dm→mage ≠ FLAG_MISSING) {
  printf("MoonAgeDays:␣%.1f\n", dm→mage);
  printf("MoonAgeHours:␣%03d\n", (int)(dm→mage * 24.0 + 0.5));
  printf("MoonIllum:␣%.3f\n", dm→mphase);
  printf("MoonTermAng:␣%.3f\n", dm→mfrac);
  ⟨ Output moon phase 376 ⟩
}
```

376. The *dm→mfrac* value represents the termination angle scaled from 0.0 to 1.0, with full moon at 0.5. A phase tolerance value of 0.015 represents a span of time of about 21 hours at the full moon, a little less than a day, which should provide sufficient opportunity to capture the phase event milestones described here.

⟨ Output moon phase 376 ⟩ ≡

```
{
  double phasetol ← 0.015;
  if (fabs(dm→mfrac − 0.25) ≤ phasetol) printf("MoonPhase:␣First␣Quarter\n");
  else if (fabs(dm→mfrac − 0.5) ≤ phasetol) printf("␣␣␣␣␣MoonPhase:␣Full␣Moon\n");
  else if (fabs(dm→mfrac − 0.75) ≤ phasetol) printf("MoonPhase:␣Third␣Quarter\n");
  else if (fabs(dm→mfrac − 1.0) ≤ phasetol ∨ (dm→mfrac ≤ phasetol))
    printf("MoonPhase:␣New␣Moon\n");
  else if (dm→mfrac < 0.25) printf("MoonPhase:␣Waxing␣Crescent\n");
  else if (dm→mfrac < 0.5) printf("MoonPhase:␣Waxing␣Gibbous\n");
  else if (dm→mfrac < 0.75) printf("MoonPhase:␣Waning␣Gibbous\n");
  else printf("MoonPhase:␣Waning␣Crescent\n");
}
```

This code is used in section 375.

377. Weather Icon. Here we figure out what kind of image to display based on weather conditions or sky conditions. We'll output a simplified string with limited combinations. It will be up to the rendering entity to decide how to deal with this string. The output strings could be made into symbolic links pointing to the appropriate image, or the output could be run through a lookup table to be decoded into the proper image.

⟨ METAR storage variables 69 ⟩ +≡

```
Cptr wximg;
```

378. ⟨ METAR initialization statements 70 ⟩ +≡

```
dm→wximg ← Λ;
```

379. ⟨ METAR deallocation statements 71 ⟩ +≡

```
if (dm→wximg) free(dm→wximg);
```

380. The image should reflect the predominating phenomenon, so we start with the present weather encodings. If there are no present weather conditions indicated, we'll use the sky status to determine the image. We'll prefix the string with `'d'` to indicate day, and `'n'` to indicate night.

Rather than deal with hundreds of combinations, we'll limit the possibilities. We'll only use the first present weather element, since it's supposed to be the most important. If there is no present weather, we'll use the cloud cover conditions from the highest layer (presumably providing the most coverage).

⟨ Retrieve info from other sources 341 ⟩ +≡

```
{
  char imgstr[128];
  if (¬dm→sunrise ∨ (dm→timestamp ≥ dm→sunrise ∧ dm→timestamp ≤ dm→sunset)) strcpy(imgstr, "d");
  else strcpy(imgstr, "n");
  if (dm→wxcntr) add_wximg(imgstr, dm→wx[0]);
  else if (dm→skycntr) add_skyimg(imgstr, dm→sky[dm→skycntr − 1]);
  else strcat(imgstr, "CLR");    /* punt */
  if (strlen(imgstr) > 1) dm→wximg ← strdup(imgstr);
}
```

381. ⟨ Output statements 72 ⟩ +≡

```
if (dm→wximg) printf("WXimg:␣%s\n", dm→wximg);
```

382. Use cloud cover to determine the image string element.

```
⟨ Function Definitions 4 ⟩ +≡
  void add_skyimg(char *str, char *sky)
  {
    if (¬(strncmp(sky, "SKC", 3) ∧ strncmp(sky, "CLR", 3) ∧
          strncmp(sky, "CAVOK", 5) ∧ strncmp(sky, "NSC", 3) ∧
          strncmp(sky, "NCD", 3))) strcat(str, "CLR");
    else if (strncmp(sky, "FEW", 3) ≡ 0) strcat(str, "FEW");
    else if (strncmp(sky, "SCT", 3) ≡ 0) strcat(str, "SCT");
    else if (strncmp(sky, "BKN", 3) ≡ 0) strcat(str, "BKN");
    else if (strncmp(sky, "OVC", 3) ≡ 0 ∨ strncmp(sky, "VV", 2) ≡ 0) strcat(str, "OVC");
  }
```

383. We won't bother with "heavy" or "light" versions of anything. Neither will we bother with "shower" versions of precipitation, unless there is no precipitation indicated except showers. Thunderstorm should be recognized, however. If no precipitation is indicated, we'll go with something from the obscuration column or other column.

```
⟨ Function Definitions 4 ⟩ +≡
  void add_wximg(char *str, char *wx)
  {
    int rain ← 0, snow ← 0, hail ← 0, freezing ← 0;
    ⟨ Check for funnel clouds and thunderstorms 384 ⟩
    ⟨ Check for precipitation 385 ⟩
    ⟨ Determine the image for precipitation 386 ⟩
    ⟨ Look for obscurations or other phenomena 387 ⟩
  }
```

384. The presence of a tornado, waterspout, or thunderstorm overrides everything else. We can return immediately. From here we will output either "FC" or "TS".

```
⟨ Check for funnel clouds and thunderstorms 384 ⟩ ≡
  if (strstr(wx, "+FC")) {
    strcat(str, "FC");
    return;
  } else if (strstr(wx, "TS")) {
    strcat(str, "TS");
    return;
  }
```

This code is used in section 383.

385. We'll allow for various combinations of rain, snow, and ice/hail.

```
⟨ Check for precipitation 385 ⟩ ≡
  if (strstr(wx, "RA") ∨ strstr(wx, "DZ") ∨ strstr(wx, "UP")) rain++;
  if (strstr(wx, "SN") ∨ strstr(wx, "SG") ∨ strstr(wx, "IC")) snow++;
  if (strstr(wx, "PL") ∨ strstr(wx, "GR") ∨ strstr(wx, "GS")) hail++;
  if (rain ∧ strstr(wx, "FZ")) freezing++;
  if (¬(snow ∨ hail) ∧ strstr(wx, "SH")) rain++;
```

This code is used in section 383.

386. If there is precipitation, then the output will be one of these strings: "RA", "SN", "GR", "FZRA", "RASN", "RAGR", "SNGR", "RASNGR".

⟨ Determine the image for precipitation 386 ⟩ ≡

```
if (rain ∨ snow ∨ hail) {
  if ((freezing ∧ rain) ∧ ¬snow ∧ ¬hail) strcat(str, "FZ");
  if (rain) strcat(str, "RA");
  if (snow) strcat(str, "SN");
  if (hail) strcat(str, "GR");
  return;
}
```

This code is used in section 383.

387. We won't display combinations of these elements; we'll pick just one. The output from here will be (in order of severity): "VA", "DU", "FU", or "FG". We ignore spray.

⟨ Look for obscurations or other phenomena 387 ⟩ ≡

```
if (strstr(wx, "VA")) strcat(str, "VA");
else if (strstr(wx, "PO") ∨ strstr(wx, "SS") ∨ strstr(wx, "DS") ∨
        strstr(wx, "DU") ∨ strstr(wx, "SA")) strcat(str, "DU");
else if (strstr(wx, "FU")) strcat(str, "FU");
else if (strstr(wx, "FG") ∨ strstr(wx, "BR") ∨ strstr(wx, "HZ")) strcat(str, "FG");
```

This code is used in section 383.

388. ⟨ Forward Defs 13 ⟩ +≡

```
void add_skyimg(char *, char *);
void add_wximg(char *, char *);
```

389. Index.

⟨ Scan two heights 146 ⟩ Used in section 143.
⟨ Short remark token analysis 214, 227, 233, 244, 272, 290, 326, 333 ⟩ Cited in section 209. Used in section 209.
⟨ Shorten *city* element 351 ⟩ Used in section 350.
⟨ Structure Definitions 40 ⟩ Used in section 1.
⟨ Token types 77, 85, 92, 100, 107, 121, 129, 139, 155, 174, 181, 189, 202, 207, 213, 219, 226, 232, 243, 253, 271, 277, 283, 289, 297, 303, 310, 316, 325, 332 ⟩ Used in section 45.
⟨ Token-parsing states 76, 84, 91, 99, 108, 122, 130, 140, 154, 175, 182, 190, 203, 208 ⟩ Used in section 43.
⟨ Transfer accumulated miscellaneous tokens 60 ⟩ Used in section 58.
⟨ Update successful token search parsing state 63 ⟩ Used in section 61.
⟨ Verify time range and perform station lookup 33 ⟩ Used in section 31.
⟨ Write present weather output 164 ⟩ Used in section 158.
⟨ **enum**s and **typedef**s for certain METAR values 41, 43, 45, 73, 96, 104, 126, 186, 223, 307, 336 ⟩ Used in section 40.

METar REPort:
A Program to Retrieve and Decode METAR Reports

(Version 0.1)

Bret D. Whissel