The `sf` program is an ncurses-based file selector. A commandline of the form `sf /path/to/dir` shows the list of files in the directory `dir` and allows the user to select files from them. Upon exit, the `sf` utility writes the full path of all selected files onto the standard output, one file per line. We use an ncurses based UI as the primary motivation for this program is to enable the user to quickly select some files to be processed by other command line utilities.

The central data structure manipulated by the `sf` utility is the *file list*. Upon startup, the file list contains all files in the directory `dir`. All files are marked unselected upon startup. There are user commands to modify the selection and the file list. The program also maintains a *current file pointer* to help the user move around in the file list.

The following list describes the basic set of user commands.

`<num=1>j` Move the current file `num` positions down.

`<num=1>k` Move the current file `num` positions up.

`m` Mark the current position. Note that this commands marks a position and not a file. This means that if you store a mark and later change the ordering of files, then the next command that uses the stored mark does not have any information about the file that used to be in the marked position.

`<num=0>;` Set the current file to `num` or the marked file.

`<sortfield>a` Sort the file list in non-increasing order of sort field. Supported values of `sortfield` are `'n` (Name), `'m` (Last modified time), `'a` (Last accessed time), `'s` (size), `'g` (glob matched), `'u` (user selected), and `'t` (file type).

`r` Reverse the order of files.

`<glob>g` Go to the first file matching `<glob>`.

`n` Go to the next file matching the last entered glob command.

`N` Go to the previos file matching the last entered glob command.

`q` Quit and write paths of all selected files to standard output.

`s` Select the current file or if there is an active mark, select all files between the mark and the current file.

`S` Deselect as in `s` command.

`t` Toggle selection status as in `s` command.

The command processor maintains a data stack. The data items entered by the user and pushed onto the stack to be used by subsequent commands. Each data item is a string, or a number, or a character, or a mark. To push a data item to the stack, simply enter the data item. A mark is pushed to the stack by the `m` command. The commands always use the top data item. For example, if the user enters `'g "*sf*" 5 j g a`, then the following sequence of operations are performed.

- Move the current file 5 positions down.
- Glob search with the glob pattern `*sf*` and jump to the first matching file.
- Sort the files such that the glob matched files appear on the top.

The following are the additional user commands implemented that provide some advanced functions.

`p` Prints the top element of the data stack.

`P` Pops the top element of the data stack.

We now define data structures used in our program.

The `FileImpl` structure is the representation of a file presented to the user. The standard library provided `DirEntry` is a structure that is equivalent to this except that calling accessor methods such as `size` returns the current size. However, since our application is supposed to be a short running one, we will just take a snapshot of all relevant file attributes and store them throughout the lifetime of the program.

⟨*File implementation*⟩≡

```
struct FileImpl
{
  private:
```

```
  const string path;
  const string name;
  const string type;
  const ulong  size;
  const long   atime;
  const long   mtime;

  public:
  // XXX: May throw Exception.
  this(in string p)
  {
    this(DirEntry(p));
  }

  this(DirEntry d)
  {
    path = d.name;
    name = baseName(path);
    if (d.isDir)
      type = "Directory";
    else if (d.isSymlink)
      type = "Symlink";
    else if (!extension(name))
      type = "Unknown";
    else
      type = extension(name);
    size = d.size;
    atime = d.timeLastAccessed.toUnixTime;
    mtime = d.timeLastModified.toUnixTime;
  }
}
```

The `FileList` structure represents the list of files presented to the user. Each file read by the program gets a unique `FileIndex` that will remain fixed throughout the program. The `ViewIndex` of a file represents the position of the file in the list presented to the user. The `ViewIndex` of a file can be changed by the user, for example, by sorting the file list.

⟨*Indices definition*⟩≡
```
alias FileIndex = Index!("F");
alias ViewIndex = Index!("V");
alias GlobIndex = Index!("G");
```

The arrays `file_` and `view_` stores how the files are permuted. The element `file_[i]` is the file index of the i^{th} file in the user's view of the file list. Conversely, the element `view_[i]` is the position, on screen, of the file with file index i . Therefore, the file index is an iterator that is never invalidated. The array `glob_` stores the list of glob matched files. The array `selected_` is a bitmap of selected files. This inconsistency between `glob_` and `selected_` is to cater to different requirements. We need fast traversal through the glob matched files through `g`, `n`, and `N` commands. On the other hand, we need constant time updates of selection status of files.

⟨*FileList implementation*⟩≡
```
⟨Indices definition⟩
private {
  ⟨File implementation⟩
  SafeRange!(FileImpl[],  FileIndex) list_;
  SafeRange!(FileIndex[], ViewIndex) file_;
  SafeRange!(ViewIndex[], FileIndex) view_;
  alias list_ this;

  SafeRange!(FileIndex[], GlobIndex) glob_;
  SafeRange!(bool[], FileIndex)      selected_;
}
```

For safe iteration over the files in the file list, we allow the user to use `foreach` to iterate over the files using `ViewIndex`.

⟨*FileList accessor functions*⟩≡
```
int opApply(scope int delegate(ViewIndex) dg) {
  int result = 0;
  for (auto i = ViewIndex(0);
            i < list_.length;
            ++i) {
    result = dg(i);
```

```
    if (result) break;
  }
  return result;
}
```

The user may also iterate using a `foreach` over all the glob matched files.

⟨*FileList accessor functions*⟩+≡

```
int opApply(scope int delegate(GlobIndex) dg) {
  int result = 0;
  for (auto i = GlobIndex(0);
            i < glob_.length;
            ++i) {
    result = dg(i);
    if (result) break;
  }
  return result;
}
```

The `convert` function will allow the user to convert one type of index into another. Note that it is not possible to convert from a file index or a view index to a glob index.

⟨*FileList accessor functions*⟩+≡

```
To convert(To, From)(in From index) const {
  static if (is(To == From)) return index;

  static if (is(To == FileIndex)) {
    static if (is(From == ViewIndex)) {
      return file_[index];
    } else static if (is(From == GlobIndex)) {
      return glob_[index];
    }
  } else static if (is(To == ViewIndex)) {
    static if (is(From == FileIndex)) {
      return view_[index];
    } else static if (is(From == GlobIndex)) {
      return convert!To(glob_[index]);
```

```
    }
  } else {
    static assert(false, "convert: Unsupported conversion.");
  }
}
```

We now define various functions that access properties of files such as name, full path, size etc. These functions can take a `FileIndex` ,`ViewIndex`, or `GlobIndex` as parameter.

⟨*FileList accessor functions*⟩+≡

```
template isIndexType(T)
{
  enum isIndexType =
    is(T == FileIndex) ||
    is(T == ViewIndex) ||
    is(T == GlobIndex);
}

const {
  auto name(T)(in T f) if (isIndexType!T)
  {
    return list_[convert!FileIndex(f)].name;
  }

  auto path(T)(in T f) if (isIndexType!T)
  {
    return list_[convert!FileIndex(f)].path;
  }

  auto type(T)(in T f) if (isIndexType!T)
  {
    return list_[convert!FileIndex(f)].type;
  }

  auto size(T)(in T f) if (isIndexType!T)
  {
```

```
    return list_[convert!FileIndex(f)].size;
  }

  auto mtime(T)(in T f) if (isIndexType!T)
  {
    return list_[convert!FileIndex(f)].mtime;
  }

  auto atime(T)(in T f) if (isIndexType!T)
  {
    return list_[convert!FileIndex(f)].atime;
  }
}
```

We now define the sort method. The `sort` method modifies `view_` and `file_` arrays to reflect the new ordering. We also ensure that the list of glob matched files in `glob_` is ordered consistently with the order in the file list. If the sorting is based on selection or glob matches, then we simply have to shift those elements to the front. This can be done in $O(s)$ time where s is the size of the list to be moved to the beginning by the partitioning algorithm. Note that we cannot use `std.algorithm.partition` as checking whether a `FileIndex` is glob matched is an expensive operation.

⟨*FileList modifying functions*⟩≡

```
private void moveToFront(R)(in R s)
  if (isInputRange!R && hasLength!R &&
      is(ElementType!R == FileIndex))
{
  assert (s.length < file_.length);

  auto i = ViewIndex(0);
  foreach (e; s) {
    auto tf   = file_[i];
    auto tv   = view_[e];
    file_[i]  = e;
    view_[e]  = i;
    file_[tv] = tf;
    view_[tf] = tv;
```

```
    i++;
  }
}

enum SortField
{
  NAME,
  MTIME,
  ATIME,
  SIZE,
  GLOB,
  SELECT,
  FILETYPE
}

void sort(SortField sf)()
  if (sf == GLOB)
{
  moveToFront(glob_);
}

void sort(SortField sf)()
  if (sf == SELECT)
{
  auto s = &this.isSelected!FileIndex;
  algo.partition!(s)(vanillaIndexed(file_));
  algo.partition!(s)(vanillaIndexed(glob_));
  fixupView();
}
```

For the other sort fields, we simply sort based on the `<` operator of corresponding fields. The `fixupView` method ensures that the `view_` array encodes the inverse permutation of `file_`.

⟨*FileList modifying functions*⟩+≡

```
private bool fileLess(SortField sf)(in FileIndex i, in FileIndex j)
{
  switch (sf) {
```

```
    case NAME:     return name(i)  < name(j);
    case MTIME:    return mtime(i) < mtime(j);
    case ATIME:    return atime(i) < atime(j);
    case SIZE:     return size(i)  < size(j);
    case FILETYPE: return type(i)  < type(j);
    default: assert(false, "BUG: Invalid sort field\n");
  }
}

private void fixupView()
{
  auto i = ViewIndex(0);
  foreach (f; file_) {
    view_[f] = i;
    i++;
  }
}

void sort(SortField sf)()
  if (sf == NAME     ||
      sf == FILETYPE ||
      sf == MTIME    ||
      sf == ATIME    ||
      sf == SIZE)
{
  auto less   = &this.fileLess!sf;
  algo.sort!(less)(vanillaIndexed(file_));
  algo.sort!(less)(vanillaIndexed(glob_));
  fixupView();
}
```

The `reverse` method reverses the order of files as seen by the user.

⟨*FileList modifying functions*⟩+≡

```
void reverse()
{
  algo.reverse(vanillaIndexed(file_));
  algo.reverse(vanillaIndexed(glob_));
```

```
    fixupView();
  }
```

The `glob` method fills the array `glob_` with the list of files matching the input glob pattern.

⟨*FileList modifying functions*⟩+≡
```
  void glob(in string pattern)
  {
    glob_ = [];
    foreach (f; file_) {
      if (globMatch(name(f), pattern)) {
        glob_ ~= f;
      }
    }
  }
```

We also provide access to the number of glob matches through a property.

⟨*FileList accessor functions*⟩+≡
```
  @property
  auto globMatches() const
  {
    return glob_.length;
  }
```

We now define methods to manipulate the current selection. We maintain a bitmap to store the current selection. This makes the property `selected` require time that is linear in the number of all files, as opposed to time linear in the number of selected files. However, the operations of selecting, deselecting, and querying whether a file is selected is constant time.

⟨*FileList modifying functions*⟩+≡
```
  void select(T)(in T f) if (isIndexType!T)
  {
    selected_[convert!FileIndex(f)] = true;
  }
```

```
void deselect(T)(in T f) if (isIndexType!T)
{
  selected_[convert!FileIndex(f)] = false;
}

bool isSelected(T)(in T f) if (isIndexType!T)
{
  return selected_[convert!FileIndex(f)];
}
```

We have to ensure that the `selected` range is sorted according to the user's view. For this, we can simply filter the file indices from the `file_` array. Recall that `file_[0]` is the first file seen by the user.

⟨*FileList modifying functions*⟩+≡

```
@property
auto selected()
{
  auto pred = &this.isSelected!FileIndex;
  return algo.filter!(pred)(vanillaIndexed(file_));
}
```

We will allow creation of a `FileList` by specifying a directory or the list of files. After we finish fetching information about all files in the directory, skipping errors if any, we set up the `file_` and `view_` arrays to identity permutations.

⟨*FileList creation*⟩≡

```
static FileList loadDirectory(in string path)
{
  FileList result;

  auto files = dirEntries(path, SpanMode.shallow);
  while (!files.empty) { // nothrow?
    auto f = files.front; // nothrow?
    try {
      result.list_ ~= FileImpl(f);
    } catch (Exception e) {
```

```
        stderr.writeln("sf: Failed to load ", f.name);
      } finally {
        quitOnError(
          files.popFront,
          "sf: Failed to iterate directory " ~ path);
      }
    }

    if (result.list_.length > 0) {
      result.view_.reserve(result.list_.length);
      result.file_.reserve(result.list_.length);

      result.selected_ = new bool[result.list_.length];
      result.view_     = array(
                           iota(
                            ViewIndex(0),
                            ViewIndex(result.list_.length)));
      result.file_     = array(
                           iota(
                            FileIndex(0),
                            FileIndex(result.list_.length)));
    }

    return result;
  }
```

Finally, we combine all the above to define our `FileList` datatype.

⟨*FileList definition*⟩≡

```
  struct FileList
  {
    ⟨FileList implementation⟩
    ⟨FileList accessor functions⟩
    ⟨FileList modifying functions⟩
    ⟨FileList creation⟩
  }
```

The second major component of our program will be the user interface. We

will use ncurses for our UI. Each UI element in ncurses is defined by a rectangle. The `x, y` coordinates will describe the location of the top left corner.

⟨*MainUI variables*⟩≡
```
static struct Rectangle {
  int height;
  int width;
  int x;
  int y;
  WINDOW *win;
}

Rectangle screen;
Rectangle fileListWin;
Rectangle fileListPad;
Rectangle echoWin;
```

The full screen is desribed by the rectangle `screen`. The rectangles `fileListWin` and `echoWin` describes the file list area and echo area. Our UI consists of two parts: the file list and the echo area. The file list displays the list of files (or a part of it) in the user-specified order. The echo area is for interacting with the user. The file list window will occupy all but the last line of the user's screen. The last line will be occupied by the window `echoWin`.

The ncurses pad `fileListPad` contains a framebuffer representation of the file list. If there are n files in the file list, then the first n lines in the pad will be occupied by those files. We allocate a buffer space of `height` lines where `height` is the number of lines in the file list window. This simplifies the logic in `show` function by allowing us to simply project `height` lines from the first file to be displayed on the screen.

⟨*Initialize UI components*⟩≡
```
int x, y;

getmaxyx(stdscr, y, x);
screen.height = y + 1;
screen.width  = x + 1;
screen.x      = 0;
screen.y      = 0;
screen.win    = stdscr;
```

```
fileListWin.height = screen.height - 1;
fileListWin.width  = screen.width;
fileListWin.x      = screen.x;
fileListWin.y      = screen.y;
fileListWin.win    = subwin(screen.win,
  fileListWin.height,
  fileListWin.width,
  fileListWin.y,
  fileListWin.x);

echoWin.height = 1;
echoWin.width  = screen.width;
echoWin.x      = screen.x;
echoWin.y      = screen.y + fileListWin.height;
echoWin.win    = subwin(stdscr,
  echoWin.height,
  echoWin.width,
  echoWin.y,
  echoWin.x);
assert(OK == keypad(echoWin.win, true));

fileListPad.height = cast(int)fileList.length + fileListWin.height;
fileListPad.width  = screen.width;
fileListPad.x      = 0;
fileListPad.y      = 0;
fileListPad.win    = newpad(
  fileListPad.height,
  fileListPad.width);
```

The window `fileListWin` will display the appropriate portion of the pad. The variable `first` contains the view index of the the first file to be displayed.

⟨*show function*⟩≡

```
private void show() {
  int sminrow, smincol, smaxrow, smaxcol;
  int fst = cast(int)first;
```

```
    assert(fst >= 0);

    sminrow = fileListWin.y;
    smincol = fileListWin.x;

    int nfiles = cast(int)fileList.length - fst;

    smaxrow = sminrow + fileListWin.height - 1,
    smaxcol = smincol + fileListWin.width  - 1;

    prefresh(fileListPad.win,
             fst,
             0,
             sminrow,
             smincol,
             smaxrow,
             smaxcol);
  }
```

To transfer the file list onto the frame buffer, we simply print out the information of each file line by line in the view index order. The file at view index `v` is always written to the line `v` in the file list pad.

⟨*Write the file at ViewIndex v into the file list pad*⟩≡

```
  auto i = toInt(v);
  auto line = "%-5d %1s %1s %-20.20s %-d".
      format(
        i,
        current == v ? ">" : " ",
        fileList.isSelected(v) ? "*" : " ",
        fileList.name(v),
        fileList.size(v));
  wmove(fileListPad.win, i, 0);
  wclrtoeol(fileListPad.win);
  auto ncline = dstringz(line);
  waddwstr(fileListPad.win, ncline);
  free(ncline);
```

We will provide overloads for reloading the entire file list into the pad or only specific files.

⟨*reloadPad function*⟩≡
```
private void reloadPad()
{
  foreach (ViewIndex v; fileList) {
    ⟨Write the file at ViewIndex v into the file list pad⟩
  }
}

private void reloadPad(R)(R files)
  if (isInputRange!R && is(ElementType!R == ViewIndex))
{
  foreach (ViewIndex v; files) {
    ⟨Write the file at ViewIndex v into the file list pad⟩
  }
}
```

The main UI stores some state related to the file list such as the file list itself, the current file, and the current glob matched file in a glob search (which is null if there no glob search has been done yet).

⟨*MainUI variables*⟩+≡
```
FileList fileList;
ViewIndex current = ViewIndex(0);
Nullable!GlobIndex globCurrent;
bool writeFiles = false;
FILE* infile;
FILE* outfile;
```

We now define the main UI structure. The `alias sink` parameter is the sink function that consumes the range of selected files. It is called if the user pressed 'q'.

⟨*MainUI definition*⟩≡
```
struct MainUI(alias sink)
{
  ⟨MainUI variables⟩
```

```
this(in string dirpath) {
  fileList =
    quitOnError(
      FileList.loadDirectory(dirpath),
      "sf: Failed to load directory " ~ dirpath);

  setlocale(LC_ALL, "");

  infile  = fopen("/dev/tty", "rb");
  scope (failure) fclose(infile);

  outfile = fopen("/dev/tty", "wb");
  scope (failure) fclose(outfile);

  assert(newterm(cast(char*)null, outfile, infile));
  scope (failure) endwin();

  // endwin() restores everything if anything follows fails.

  assert(OK == noecho());
  assert(OK == nonl());
  assert(ERR != curs_set(0));

  ⟨Initialize UI components⟩
  reloadPad();
  show();
  loop();
}

~this() {
  endwin();
  fclose(infile);
  fclose(outfile);
  if (writeFiles) {
      sink(algo.map!(f => fileList.path(f))
                    (fileList.selected));
  }
```

```
  }

  ⟨show function⟩
  ⟨Echo area functions⟩
  ⟨reloadPad function⟩

  ⟨loop function⟩
}
```

The echo window can be accessed in the following ways: The function `readKey` reads a character entered by the user, the function `writeChar` writes the character to the current position in the echo area taking care of writing past the last position in the window, the function `clearEcho` clears the echo area.

⟨*Echo area functions*⟩≡

```
auto readKey()
{
  dchar c;
  assert(wget_wch(echoWin.win, &c) == OK);
  return c;
}

void writeChar(dchar ch)
{
  writeStr(to!string(ch));
}

void writeStr(string s)
{
  waddstr(echoWin.win, toStringz(s));
}

void clearEcho()
{
  wclear(echoWin.win);
}

void pushBackKey(dchar ch)
```

```
{
  unget_wch(ch);
}
```

Now we describe our command processing loop. The input to the command processor is a stack based programming language. We support the following data types: double-quoted strings, single-quote preceeded characters, natural numbers, and marks which are positions in the file list. The commands look at the top of the stack for arguments. For example, the `j` command looks for a number at the top of the stack and if it does not find one, uses the default value of 1. The variable `top` points to the next empty spot in the stack. Notice that all casts in the `set()` are safe as long as we do not modify values on the stack.

⟨*Data stack definition*⟩≡

```
static struct DataStack(size_t nelems) {
  static struct DataStackElem {
    enum _Type { NODATA, NUM, STR, CH, MARK }

    _Type type;

    union _Data {
      int num;
      string str;
      dchar ch;
    }

    _Data data;

    alias data this;

    void set(T, _Type t = _Type.NODATA)(in T data) {
      static if (is(T == int)) {
        static assert(t != _Type.NODATA,
          "DataStackElem: Specify type of int.");
        type = t;
        num  = cast(int)data;
      } else static if (is(T == string)) {
```

```
      type = STR;
      str = cast(string)data;
    } else static if (is(T == dchar)) {
      type = CH;
      ch = cast(dchar)data;
    } else {
      static assert(false,
        "DataStackElem: Unsupported data type.");
    }
  }

  string toString() const {
    final switch (type) {
    case NUM:    return to!string(num);
    case STR:    return "\"" ~ to!string(str) ~ "\"";
    case CH:     return "\'" ~ to!string(ch);
    case MARK:   return to!string(num) ~ "m";
    case NODATA: assert(0);
    }
  }
}

DataStackElem[nelems] stack;
int top = 0;

alias SType = DataStackElem._Type;

template STypeToType(SType t) {
  static if (t == SType.NUM || t == SType.MARK)
    alias STypeToType = int;
  else static if (t == SType.CH)
    alias STypeToType = dchar;
  static if (t == SType.STR)
    alias STypeToType = string;
}

void push(SType t, T = STypeToType!t)(in T d)
{
```

```
  assert (top < nelems);
  stack[top].set!(T, t)(d);
  ++top;
}

auto pop(SType t)()
{
  Nullable!(STypeToType!t) result;

  if (top == 0) return result;
  if (t == stack[top - 1].type) {
    static if (t == SType.NUM ||
               t == SType.MARK) {
      --top;
      result = stack[top].num;
    } else static if (t == SType.STR) {
      --top;
      result = stack[top].str;
    } else static if (t == SType.CH) {
      --top;
      result = stack[top].ch;
    } else {
      static assert(false, "pop: Invalid type.");
    }
  }
  return result;
}

auto peek()
{
  Nullable!DataStackElem result;
  if (top > 0) {
    result = stack[top - 1];
  }
  return result;
}

void popAny()
```

```
  {
    if (top > 0) --top;
  }
}

DataStack!100 dataStack;

alias SType  = dataStack.SType;
alias NUM    = SType.NUM;
alias MARK   = SType.MARK;
alias CH     = SType.CH;
alias STR    = SType.STR;
alias NODATA = SType.NODATA;
```

The words in the programming language are grouped into data or commands. The start state can determine the type of the word by only looking at the first non-blank character. If it is a " or ' or a number, then it's data, otherwise it must be a valid command.

⟨*States of the command processor*⟩≡

```
enum {
  START,
  READ_CH,
  READ_STR,
  READ_NUM,
  READ_COMMAND,
  QUIT
}

int state = START;
```

The character `c` is used to read the current character input by the user.

⟨*Local variables for command processor*⟩≡

```
dchar c;
```

The command processor works in iterations. In each iteration, it reads a key from the user and forwards the key to the appropriate state. The state

handles the key and sets the next state. The command processor should echo the data item being entered by the user to provide feedback.

⟨*Setup state for the command processor*⟩≡
```
c   = cast(dchar)readKey();
```

The start state simply looks at the key and delegates to the appropriate state to handle the key.

⟨*Start state*⟩≡
```
case START:
  clearEcho();
  if (c.isWhite) {
    state = START;
  } else if (c == '\"') {
    s = "";
    writeChar('\"');
    state = READ_STR;
  } else if (c == '\'') {
    writeChar('\'');
    state = READ_CH;
  } else if (c.isDigit) {
    n = 0;
    pushBackKey(c);
    state = READ_NUM;
  } else {
    pushBackKey(c);
    state = READ_COMMAND;
  }
  break;
```

Processing a data item is straight-forward. Simply read it and push it onto the stack. The variables `n` and `s` are used to keep track of the number and string input the user.

⟨*Local variables for command processor*⟩+≡
```
int n;
string s;
```

⟨*Read data items*⟩≡
```
case READ_CH:
  writeChar(c);
  dataStack.push!CH(c);
  state = START;
  break;

case READ_STR:
  if (c != '\"') {
    writeChar(c);
    s ~= c;
  } else {
    writeChar('\"');
    dataStack.push!STR(s);
    state = START;
  }
  break;

case READ_NUM:
  if (c.isDigit) {
    writeChar(c);
    n = n * 10 + charToInt(c);
  } else {
    dataStack.push!NUM(n);
    pushBackKey(c);
    state = START;
  }
  break;
```

Now we process the commands. All commands are single letter commands. So we simply have to look at the character, pop the appropriate data items from the stack, and update the file list. All commands being single letter also allows us to put the state transition outside the command handling `switch` statement.

⟨*Read commands*⟩≡
```
case READ_COMMAND:
  switch (c) {
```

```
    ⟨Handle commands⟩
  }
  state = START;
  break;

default: assert(0);
```

The reverse command.

⟨*Handle commands*⟩≡

```
case 'r':
  fileList.reverse(); break;
```

Handle motion commands. This part illustrates why mark and number has to be separate data types. If they were the same, any `j` or `k` after `m` will consume the pushed mark which is not what we would want. The right behaviour is to save the mark until the next `;` command (or, more importantly, one of the selection commands which uses the mark to specify the range of files).

⟨*Handle commands*⟩+≡

```
case 'j':
  setCurrent(cur + dataStack.pop!NUM().ifNull(1));
  break;
case 'k':
  setCurrent(cur - dataStack.pop!NUM().ifNull(1));
  break;
case ';':
  auto x = dataStack.pop!MARK();
  if (x.isNull) {
    x = dataStack.pop!NUM();
  }
  auto dest = x.ifNull(0);
  setCurrent(dest);
  break;
case 'm':
  dataStack.push!MARK(cur);
  break;
```

For the selection commands, we first check the stack to see whether some position is marked. If a mark is present, we select/deselect all files from the mark to the current position (both inclusive). If a mark is not present, then we select/deselect the current file. The range `[beg, end)` specifies the files to be selected.

⟨*Local variables for command processor*⟩+≡
```
int m, beg, end;
```

⟨*Obtain range of files*⟩≡
```
m = dataStack.pop!MARK().ifNull(cur);

if (m > cur) {
  beg = cur;
  end = m + 1;
} else {
  beg = m;
  end = cur + 1;
}
```

⟨*Handle commands*⟩+≡
```
case 's':
  ⟨Obtain range of files⟩
  foreach (v; beg .. end) {
    auto v1 = ViewIndex(v);
    fileList.select(v1);
  }
  break;
case 'S':
  ⟨Obtain range of files⟩
  foreach (v; beg .. end) {
    auto v1 = ViewIndex(v);
    fileList.deselect(v1);
  }
  break;
case 't':
  ⟨Obtain range of files⟩
  foreach (v; beg .. end) {
```

```
    auto v1 = ViewIndex(v);
    if (fileList.isSelected(v1))
      fileList.deselect(v1);
    else
      fileList.select(v1);
  }
  break;
```

The sort command accepts an argument on the stack that specifies the sort field. We use the file name as the default sort field.

⟨*Handle commands*⟩+≡

```
case 'a':
  auto sortarg = dataStack.pop!CH().ifNull(dchar('n'));
  switch (sortarg) {
    case 'n': fileList.sort!(NAME);     break;
    case 'm': fileList.sort!(MTIME);    break;
    case 'a': fileList.sort!(ATIME);    break;
    case 's': fileList.sort!(SIZE);     break;
    case 'g': fileList.sort!(GLOB);     break;
    case 'u': fileList.sort!(SELECT);   break;
    case 't': fileList.sort!(FILETYPE); break;
    default:  break;
  }
  break;
```

The glob command simply delegates the actual glob search to the underlying file list. The main work is to maintain `globCurrent` and `current`.

⟨*Handle commands*⟩+≡

```
case 'g':
  string globarg = dataStack.pop!STR().ifNull("");
  fileList.glob(globarg);
  if (fileList.globMatches) {
    setGlobCurrent(0);
    setCurrent(
      toInt(fileList
            .convert!ViewIndex(
```

```
                globCurrent.get)));
    }
    state = START;
    break;

  case 'n':
    if (!globCurrent.isNull) {
      auto g = toInt(globCurrent.get);
      setGlobCurrent(g + 1);
      setCurrent(
        toInt(fileList
              .convert!ViewIndex(
                globCurrent.get)));
    }
    state = START;
    break;

  case 'N':
    if (!globCurrent.isNull) {
      auto g = toInt(globCurrent.get);
      setGlobCurrent(g - 1);
      setCurrent(
        toInt(fileList
              .convert!ViewIndex(
                globCurrent.get)));
    }
    state = START;
    break;
```

Quit command.

⟨*Handle commands*⟩+≡

```
  case 'q':
    writeFiles = true;
    goto quit;
```

⟨*Handle commands*⟩+≡

```
  default: break; // Unknown command letter.
```

We define some commands to manipulate the data stack. The peek command prints the top stack element in the echo area. The pop command pops the top stack element.

⟨*Handle commands*⟩+≡

```
case 'p':
  clearEcho();
  auto top = dataStack.peek();
  if (!top.isNull) {
    writeStr(top.get.toString());
  }
  break;
case 'P':
  dataStack.popAny();
  break;
```

The following are convenience functions used by our command processor.

⟨*Convenience functions for command processor*⟩≡

```
void setCurrent(in int newcur)
{
  alias VI = ViewIndex;
  current = VI(bound(
                newcur,
                0,
                cast(int)fileList.length - 1));
}

@property
int cur()
{
  return toInt(current);
}

void setGlobCurrent(in int newcur)
{
  alias GI = GlobIndex;
```

```
  globCurrent = GI(bound(
                  newcur,
                  0,
                  cast(int)fileList.globMatches - 1));
}

static int charToInt(in dchar c)
{
  assert(c.isDigit);
  return c - '0';
}
```

Finally, we put everything together to make our command processor.

⟨*loop function*⟩≡

```
void loop()
{
  ⟨Convenience functions for command processor⟩
  ⟨Local variables for command processor⟩
  ⟨Data stack definition⟩
  ⟨States of the command processor⟩

  while (true) {
    ⟨Setup state for the command processor⟩
    switch(state) {
      ⟨Start state⟩
      ⟨Read data items⟩
      ⟨Read commands⟩
    }
    ⟨Display the appropriate portion of file list pad⟩
  }

quit: return;
}
```

We will now describe our main redrawing logic. The current file has to be displayed always. If the file list window has n lines, then we have to determine the appropriate n files to reload into the file list pad and project on to the

window. We keep track of the view index of the first file displayed. From this and the n, we can determine whether the current file is displayed or not. If the current file is not displayed, then we have to change the first file displayed so that the current file is displayed.

If the user has scrolled down too far, then we arrange things so that the current file is displayed approximately 1/5 from the beginning of the screen. If the user has scrolled up too far, then we arrange things so that the current file is displayed approximately 4/5 from the end of the screen.

⟨*MainUI variables*⟩+≡
```
ViewIndex first = ViewIndex(0);
```

⟨*Display the appropriate portion of file list pad*⟩≡
```
int nlines = fileListWin.height;

if (current < first) {
  first = ViewIndex(lbound(cur - 4 * nlines/5, 0));
} else if (cur >= first + nlines) {
  first = ViewIndex(lbound(cur - nlines/5, 0));
}

ViewIndex last = ViewIndex(
  ubound(toInt(first) + nlines,
         cast(int)fileList.length));
reloadPad(iota(first, last));
show();
```

Now we will write some output routines that print the files selected into standard output in various common formats.

⟨*Output routines*⟩≡
```
void printn(R)(R files)
{
  foreach (f; files) writeln(f);
}

void printq(R)(R files)
{
```

```
  foreach (f; files) {
    write("\""); write(f); write("\"");
    write(" ");
  }
}

void printz(R)(R files)
{
  foreach (f; files) {
    write(f);
    write("\0");
  }
}
```

Our application.

⟨*sf.d*⟩≡

⟨*Import statements*⟩
⟨*Definitions for index and array types*⟩
⟨*Convenience functions*⟩
⟨*FileList definition*⟩
⟨*Useful aliases*⟩
⟨*MainUI definition*⟩
⟨*Output routines*⟩

```
void main(string[] args)
{
  enum OutputFormat {
  n, // One file per-line. Assumes no newlines in filenames.
  q, // double-quoted separated by spaces.
  z, // separated by null bytes.
  }
  OutputFormat outFormat = OutputFormat.n;
  auto helpInformation =
    getopt(args,
           "output|o", r"Output format
    n - One file per-line. Assumes there are no newlines in filenames. (defau
    q - Double-quoted, separated by spaces.
```

```
    z - Separated by null bytes.", &outFormat
    );
  if (helpInformation.helpWanted) {
    defaultGetoptPrinter(
      "sf [-o{n|q|z}] [<dir>]: Select files from <dir> or current directory."
      helpInformation.options);
    return;
  }
  string dirpath = ".";
  if (args.length > 1) {
    dirpath = args[$ - 1];
  }
  final switch (outFormat) {
  case OutputFormat.n:
    MainUI!printn(dirpath);
    break;
  case OutputFormat.q:
    MainUI!printq(dirpath);
    break;
  case OutputFormat.z:
    MainUI!printz(dirpath);
    break;
  }
}
```

⟨*Import statements*⟩≡

```
import std.string    : toStringz;
import std.stdio     : writeln, write, stderr;
import std.format    : format;
import std.file      : DirEntry, dirEntries,
                       SpanMode, FileException;
import std.path      : baseName, extension, globMatch;
import std.traits    : isAssignable, TemplateOf;
import std.range     : isInputRange, isRandomAccessRange,
                       ElementType, hasSlicing, hasLength,
                       iota;
import std.array     : array;
```

```
import std.ascii     : isWhite, isDigit;
import std.typecons  : Nullable;
import std.getopt    : defaultGetoptPrinter, getopt;
import std.conv      : to, dtext;

import core.stdc.stdlib : exit, malloc, free;
import core.stdc.stdio  : FILE, fopen, fclose;
import core.stdc.locale : setlocale, LC_ALL;

import algo = std.algorithm;
import deimos.ncurses;
```

Now we define the various index and array types. The type `indexType!(cookie)`, where `cookie` is a literal string, is a distinct index type for distinct cookies. We disallow the possibility of mixing up conceptually distinct index types by overloading assignment in index types and using `SafeRange` in place of normal arrays. The function call `vanillaIndexed(array)` allows one to use plain `size_t` to index any array that can only be indexed using one of the special index types. This is useful to interface with standard library functions which expects all random access ranges to be indexable using `size_t`.

⟨*Definitions for index and array types*⟩≡

```
struct Index(string cookie)
{
  size_t i_;
  alias i_ this;
  Index opBinary(string op)(in int rhs)
  {
    Index other = this;
    mixin("other.i_ = i_ " ~ op ~ " rhs;");
    return other;
  }
  auto ref opAssign(in Index other)
  {
    i_ = other.i_;
  }
}
```

We will also allow to convert an index into an `int`.

⟨*Definitions for index and array types*⟩+≡

```
int toInt(T : Index!s, alias s)(in T x)
{
  return cast(int)x;
}

struct SafeRange(R, U)
  if (isRandomAccessRange!R &&
      hasSlicing!R          &&
      hasLength!R           &&
      is(U : size_t))
{
  alias IndexType = U;

  R list_;
  alias list_ this;

  auto ref opAssign(R other) {
    list_ = other;
    return this;
  }

  @property
  SafeRange save() { return this; }

  auto ref opIndex(in U i) inout {
    return list_[i];
  }

  SafeRange opSlice(in U l, in U h)
  {
    SafeRange other = this;
    other.list_ = list_[l .. h];
    return other;
  }
```

```
  @property
  U opDollar() const { return U(length); }

  private alias T = ElementType!R;
  static if (isAssignable!T) {
    auto ref opIndexAssign(in T val, in U i) {
      list_[i] = val;
      return list_[i];
    }
  }
}

struct VanillaIndexed(R)
  if (__traits(isSame, TemplateOf!R, SafeRange))
{
  R r_;
  alias r_ this;

  this(R r) {r_ = r;}

  @property
  VanillaIndexed save() { return this; }

  auto ref opIndex(in size_t i) inout {
    return r_[R.IndexType(i)];
  }

  VanillaIndexed opSlice(in size_t l, in size_t h)
  {
    VanillaIndexed other = this;
    other.r_ = r_[R.IndexType(l) .. R.IndexType(h)];
    return other;
  }

  private alias T = ElementType!R;
  static if (isAssignable!(T)) {
    auto ref opIndexAssign(in T val, in size_t i) {
      r_[R.IndexType(i)] = val;
```

```
      return r_[R.IndexType(i)];
    }
  }
}

// For automatic type deduction of R.
auto vanillaIndexed(R)(R r)
{
  return VanillaIndexed!(R)(r);
}
```

Some convenience functions.

⟨*Convenience functions*⟩≡

```
auto ifNull(T : Nullable!U, U)(in T x, in U d)
{
  return x.isNull ? d : x.get;
}

auto bound(int val, int min, int max)
{
  if (val < min) return min;
  else if (val > max) return max;
  return val;
}

auto lbound(int val, int min)
{
  if (val < min) return min;
  return val;
}

auto ubound(int val, int max)
{
  if (val > max) return max;
  return val;
}
```

```
auto quitOnError(E)(lazy E expr, in string msg)
{
  try {
    static if (is(E == void)) {
      expr();
      return;
    } else {
      auto result = expr();
      return result;
    }
  } catch (Exception e) {
    stderr.writeln (msg);
    exit(1);
  }
  assert(0);
}

auto dstringz(string s)
{
  auto r = cast(dchar*)malloc(dchar.sizeof * s.length + 1);
  auto x = dtext(s);
  for (auto i = 0; i < x.length; ++i) {
    r[i] = x[i];
  }
  r[x.length] = 0;
  return r;
}
```

Some useful aliases.

⟨*Useful aliases*⟩≡

```
alias FileList.ViewIndex ViewIndex;
alias FileList.FileIndex FileIndex;
alias FileList.GlobIndex GlobIndex;

alias NAME     = FileList.SortField.NAME;
alias MTIME    = FileList.SortField.MTIME;
```

```
alias ATIME    = FileList.SortField.ATIME;
alias SIZE     = FileList.SortField.SIZE;
alias GLOB     = FileList.SortField.GLOB;
alias SELECT   = FileList.SortField.SELECT;
alias FILETYPE = FileList.SortField.FILETYPE;
```

Some implementation notes

There are many casts from unsigned long to int in the program. These do not check for integer overflow.

Note that we are using `waddwstr` to print file information that should correctly handle unicode characters in the filename. The function `dstringz` converts a Dlang `string` into a null-terminated array of `wchar_t = dchar` (True in POSIX).

The program also makes the assumption `wchar_t = wint_t = dchar` which is true in POSIX.