lx: the library

22nd June 2002

Contents

1 Introduction

Lx is a library for building tools to do lexical analyses of text. And in particular, tools that can do real time, interactive exploration of texts.

Some of the tools easily available at the time lx was first conceived did things such as collocational analysis, but required that the text be preprocessed to prepare it. An initial goal for lx was to be able to do this upon text load, and then extend this to do collocational analysis of sets (classification schemes) of words.

A further, and perhaps the next major goal, is to have the interface to the data structures be a scripting language that can be used to construct ad-hoc queries and analyses on the fly.

1.1 A naming of parts

Lx is also called lx2, and is the second run at it after a hiatus of some time and the deletion of most parts of the original lx.

Lxt is a test program and semi-useful tool built on the lx library. It can do a number of things, including run simple batch scripts; but may be considered user unfriendly.

Moth is a hypothetical future analysis program built on the lx library.

This is the documentation for lx, the library.

1.2 Parts still needed

Lx (and lxt) needs several major things to be truly useful:

- Unicode and locale support for character sets other than ASCII.
- Better error and exception handling.
- A scripting language. The interface to the library should be more easily programmed than by writing C modules. Any number of useful things could be done as quick scripts.

Nevertheless, lx and lxt in their current form can do a few interesting and useful things.

This document needs:

- More and better examples.
- Better coverage of error conditions and results for each function.

2 Data structures overview

An understanding of the data structures used by lx is necessary in order to make use of it.

2.1 The lexical list

A text file is parsed into words and loaded into memory as a lexical list (called a lxLexlist). Each word is stored as node, with pointers to the next and previous words, as well as a pointer to the next occurence in the text of the same word. As well, punctuation and formatting is stored in the list, and in fact it is trivial to have lx print out the text exactly as it was input.

An application may have multiple texts loaded at a time, and there is some support for tracking them all as a linked list of loaded texts.

2.2 Markers and counters

It is possible to embed markers and counters in the text, and lx will track and use them.

- A marker has a type and defines a range of text that extends until the next marker of the same type. For example, a play may have markers of a type "speaker", which all the characters are members of, and which are used to separate roles.
- A counter has a name and a value. Each occurence of the counter in the text increments the value. Counters can be used to mark verses or lines (although there is an implicit, automatic counter for new-lines).

Ranges and counters are compiled as the text is loaded. A range is defined so:

```
<CORDELIA=speaker>
```

And then referenced in the text as:

```
<CORDELIA>
```

Or defined and referenced in one step as

```
<CORDELIA:speaker>
```

A counter is defined so:

```
<l=#>
```

And then referenced in the text as:

```
<l>
```

Or defined and referenced in one step as:

```
<l:#>
```

Counters and markers must be defined either before or at the first time they are referenced. An undefined marker or counter will generate a warning from the compiler.

2.3 Comments

By convention, items in square brackets "[]" are comments, are stored, but are not parsed into lexical items (or markers or counters).

2.4 The index

When the list is loaded, an index of the words it contains is constructed. The index only contains words (lexical items) and does not contain markers, counters, or objects in the list that contain punctuation, etc.

An entry in the index points to the first occurence of the word in the text, all other occurences can be found by following the chain of pointers to the next occurence of the word.

2.5 Roots and forms of words

The index into the lexical list supports mapping the forms of a word to a root word. The mapping is performed by running a table of roots and forms against the index after the text is loaded. The table may contain a much larger vocabulary than the loaded text does: anything not in the text will be ignored. More than one table can be run against a text. So it is possible to run a comprehensive table against any text, and then run a smaller more specialized one (for example, taking into account place and character names).

The scheme is meant to be used for one-to-many mappings, specifically roots and forms of words. But there is a great deal of flexibility in how tables are constructed, and the creator of root and form tables may be as precise as to differentiate between "refer" and the noun "referer", or loose enough to include "reference", or even to group words that have roots cognate across languages. Therefore some care and consistency is required when setting up tables.

2.6 Classes and categories

A classification scheme ("class" for short) can also be applied to a text. This is much more flexible than the root and forms above. Words are mapped to categories. Words can belong to multiple categories, and categories can contain sub-categories. A class or categories table is a table much like a roots and forms table, but is loaded into memory as an independent, named object. A class is then "bound" to a lexical list: its members are matched to existing words in the lexical list, and the lexical index entry for a word is also set to point back to the word's categories in the class.

Once loaded and bound to a text, operations can then be performed on categories of words; say, for example, the frequency of colour words in a text.

A classification scheme may be entirely arbitrary, mapping forms of words (singular/plural, present/past, noun/verb etc), themes, root language (Latin, French, Danish, etc.), or whatever else.

More than one class may be loaded at a time, but only one may be bound to a particular text at a time.

3 Libraries and files

All lx data structures are defined in lib/lx_lex.h, along with all the routines that manipulate them. The core library is liblex.a. Other libraries supply data structure support and utility functions. (These may (should) coelesce as they are already all quite non-optional.)

Library	Header files	Description
`liblex.a`	`lx_lex.h`	lx data structures and functions
`liblist.a`	`blist.h hash.h tree.h`	general data structures
`libutils.a`	`utils.h`	small utility functions
`liberror.a`	`error.h`	error message handling
`libio.a`	`io.h`	I/O abstraction

4 lx Library functions and macros

There are probably more functions here than are needed; and a number needed that are missing. There are three directions for lx development:

1. Adding new functions as the need is discovered in the writing of analysis tools;
2. Streamlining and abstracting functions as better or higher-level ways are discovered;
3. Deleting functions that are not used, or are superceded.

Number 3 happens much less than it should. But numbers 1 and 2 still have a long ways to go.

For the most part what is documented here is the interface presented to the application. A number of modules have private (static) functions that are not referenced here. A big example is many of the components used for data structure construction on text load, where even functions that are exposed aren't really for direct application use.

4.1 Object and flag referencing

These are functions and macros (macros, for the most) that return information about the items in the text lexical list, or in the list index. The purpose is to provide comfortable mnemonics, while allowing for transparent changes to the underlying data structures at some future date.

4.1.1 Object parts

```
char *lx_token(lxLexitem *item);
int   lx_inum(lxLexitem *item);
int   lx_num(lxLexitem *item);
char *lx_indextoken(lxIndexnode *li);
```

These macros return parts of items. lx_token() returns the character string for the list item. lx_inum() and lx_num() return the hard and soft item numbers respectively[1]. And lx_indextoken() returns the character string for the (first occurence of) the word in the text.

A note about the values of counters: the value of a counter is stored in its soft number (num). So having picked out a paragraph counter (*lx*) with lx_iscounter() below, its counter name is returned by lx_token(lx), and its value by lx_num(lx):

```
if (lx_iscounter(lx))
        printf("counter \"%s\" = %d\n", lx_token(lx), lx_num(lx));
```

4.1.2 Lexical item types and flags

```
int lx_islex(lxLexitem *item);
int lx_ispunct(lxLexitem *item);
int lx_ismarker(lxLexitem *item);
int lx_iscounter(lxLexitem *item);
int lx_isrange(lxLexitem *item);
int lx_iscomment(lxLexitem *item);
int lx_isomit(lxLexitem *item);
int lx_inrange(lxLexitem *item);
int lx_islinefeed(lxLexitem *item);
```

These macros return 0 or 1 if the corresponding flag bit is set in the item. They should be pretty self-evident. A line-feed forms a special type of counter whose name is the line-feed character, and whose num is the number of linefeeds seen since the beginning of the text.

4.1.3 Index item types

```
int lxf_isrootintext(lxIndexnode *li)
int lxf_isrootnotintext(lxIndexnode *li)
int lxf_isroot(lxIndexnode *li);
int lxf_isassignedform(lxIndexnode *li);
int lxf_isunassignedform(lxIndexnode *li)
int lxf_isform(lxIndexnode *li)
int lxf_isorphan(lxIndexnode *li);
```

These macros return 0 or 1 depending on the type of word form the index node is. Again, these should be pretty self-evident, except possibly for orphans. Orphans are nodes that were roots that weren't in the text, but the mapping has since been deleted. As a node was created in the tree, and since we don't (yet) support deletion of single nodes in the index tree, these items are orphaned. They will be ignored by routines that aren't specifically looking for them.

[1]The hard number is the absolute, unchanging position of the item in the text list. The soft number counts only lexical, non-stopped, items. The soft number may also be used for other things, such as counters, where because the item is not a lexical item, it will never be used as an item number by lx routines.

4.2 Loading and unloading text

```
int lx_loadnewlist(lxText *text, char *name, IO *io);
```

Create the new list *name* attached to the text list *text* and load it from the already opened IO *io*. See ioopen().

```
lxLexlist *lx_newlexlist(char *name);
```

Create a new lexical list. (If not managing more than one list, you can then use lx_loadlist() below to load.)

```
int lx_loadlistfromio(lxLexlist *list, IO *io);
```

Load the list *list* from the opened IO *io*.

```
int lx_unloadlistfortext(lxText *text, char *name);
```

Unload the list *name* (attached to text). This frees all data structures and memory associated with them.

```
int lx_unloadlist(lxLexlist *list);
```

The input parser can be selected by setting the function pointer lx_readtoken to an appropriate function. By default it is set to lx_default_readtoken() which is a plain parser and does not do markers or counters. The marker and counter parser can be selected by setting in the application[2]:

```
lx_readtoken = lx_marker_readtoken;
```

4.3 Basic list traversal functions

Basic list and index traversal functions. Most commonly used in for(;;) loops. The first functions return items of any type, functions further down the list are more discriminating and return objects that match various criteria.

They are all functions, not macros, so that pointers to them can be set and passed by other layers and callers (see the higher-level traversal functions below these in section 4.5 on page 11).

4.3.1 “Raw” items

```
lxLexitem *lx_firstitem(lxLexlist *list)
lxLexitem *lx_lastitem(lxLexlist *list);
lxLexitem *lx_nextitem(lxLexitem *item);
lxLexitem *lx_previtem(lxLexitem *item);
```

Returns the appropriate items. Items are raw items: that is the item is returned whether it is a word, a comment, a marker, etc.

```
lxLexitem *lx;
for (lx = lx_firstitem(list); lx; lx = lx_nextitem(lx)) {
        printf("%s", lx_token(lx));
}
```

[2] lx_marker_readtoken() should change to be the default!

4.3.2 “Raw” items in range

```
lxLexitem *lx_firstrngitem(lxLexlist *list);
lxLexitem *lx_lastrngitem(lxLexlist *list);
lxLexitem *lx_nextrngitem(lxLexitem *item);
lxLexitem *lx_prevrngitem(lxLexitem *item);
```

Returns the appropriate in-range items. Items are raw items: that is the item is returned whether it is a word, a comment, marker, etc. The only consideration is whether it falls with a mapped range.

4.3.3 Goto item

```
lxLexitem *lx_gotoitemnum(lxLexlist *list, int pos);
```

Goto item number *pos*. This is the "hard", or absolute, item number number (lx_inum()).

4.3.4 Lexical items (Words)

The following functions return only items that are lexical items (words). **NOTE:** all lex (lx_*lex()) operations only return lexical items that are in range.

```
lxLexitem *lx_firstlex(lxLexlist *list);
lxLexitem *lx_lastlex(lxLexlist *list);
lxLexitem *lx_nextlex(lxLexitem *item);
lxLexitem *lx_prevlex(lxLexitem *item);
```

Like the others above, but returning only words.

4.3.5 Goto lexical item

```
lxLexitem *lx_gotolexnum(lxLexlist *list, int pos);
```

Goto item number *pos.* lx_gotolexnum() seeks by "soft" number (lx_num()).

4.3.6 Flagged lexical items

```
lxLexitem *lx_firstflaggedlex(lxLexlist *list, int flags);
lxLexitem *lx_nextflaggedlex(lxLexitem *lx, int flags);
```

Return items that have the matching flags set.

```
int flags = 0;
lxLexitem *lx;
flags |= FL_ISUSR1;
for (lx = lx_firstflaggedlex(list); lx; lx = lx_nextflaggedlex(lx)) {
        printf("%s\n", lx_token(lx));
}
```

4.3.7 Non-stopped lexical items (no omit flag set)

```
lxLexitem *lx_firstnslex(lxLexlist *list);
lxLexitem *lx_lastnslex(lxLexlist *list);
lxLexitem *lx_nextnslex(lxLexitem *item);
lxLexitem *lx_prevnslex(lxLexitem *item);
```

Return items that do not have the omit bit set.

4.3.8 Non-stopped items with other flags set

```
lxLexitem *lx_firstnsflaggedlex(lxLexlist *list, int flags);
lxLexitem *lx_nextnsflaggedlex(lxLexitem *lx, int flags);
```

Return items that do not have the omit bit set, and also have the matching flags set.

4.4 Basic index traversal functions

```
lxIndexnode *lx_firstindex(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_lastindex(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_nextindex(TreeCursor *tp);
lxIndexnode *lx_previndex(TreeCursor *tp);
```

Walk the lexical index in order. Items are necessarily lexical items (or they wouldn't be indexed), and the routines automatically skip roots that do not appear in the text and orphan roots.

```
TreeCursor tp;
lxIndexnode *li;
for (li = lx_firstindex(list, &tp); li; li = lx_nextindex(&tp)) {
        printf("%s\n", lx_indextoken(li));
}
```

In the example above, walk the lexical index in order, printing each word.

4.4.1 Non-stopped words in the index

```
lxIndexnode *lx_firstnsindex(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_lastnsindex(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_nextnsindex(TreeCursor *tp);
lxIndexnode *lx_prevnsindex(TreeCursor *tp);
```

Walk the lexical index in order, and return index nodes whose dependent items do not have the omit bit set.

4.4.2 The first and following occurences of words in a text

```
lxLexitem *lx_firstoccur(lxIndexnode *lexinode);
lxLexitem *lx_nextoccur(lxLexitem *item);
lxLexitem *lx_firstnsoccur(lxIndexnode *lexinode);
lxLexitem *lx_nextnsoccur(lxLexitem *lex);
```

Example, return the first and next in-range occurences of items:

```
lxIndexnode *li;
lxLexitem *lx;
if ((li = lx_findcaselexinode(list, "cow")) == NULL);
        return -1;
for (lx = lx_firstoccur(li); lx; lx = lx_nextoccur(lx)) {
        printf("%s: %d\n", lx_token(lx), lx_inum(lx));
}
```

4.4.3 Index nodes that are root forms

```
lxIndexnode *lx_firstroot(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_lastroot(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_nextroot(TreeCursor *tp);
lxIndexnode *lx_prevroot(TreeCursor *tp);
```

Return root items, both in-text and dangling (not in text and orphaned). For example:

```
TreeCursor tp;
lxIndexnode *li;
for (li = lx_firstroot(list, &tp); li; li = lx_nextroot(&tp)) {
        printf("%s\n", lx_indextoken(li));
}
```

4.4.4 Any index node

```
lxIndexnode *lx_firstinode(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_lastinode(lxLexlist *list, TreeCursor *tp);
lxIndexnode *lx_nextinode(TreeCursor *tp);
lxIndexnode *lx_previnode(TreeCursor *tp);
```

Return the first and following index nodes of any type.

4.5 Higher level list and index traversal

The lxTrav functions provide a higher-level interface to list and index traversal. The primary advantage is that flags and lxtv structure can be passed down through calling functions, allowing a higher level function to setup behaviour for the lower level.

```
lxTrav *lx_travinit(lxLexlist *list, int flags);
```

Get and setup a new the traversal structure, and action. (See 4.5.1 for flag values and examples.) The traversal structure returned must be later freed by calling lx_travfree().

```
int lx_travsetup(lxTrav *lxtv, int flags);
```

Setup an existing traversal structure. (See 4.5.1 for flag values and examples.)

```
void lx_travfree(lxTrav *lxtv);
```

Free a traversal structure.

4.5.1 Flag values

Flag values passed to lx_travinit() set the behaviour for the traversal:

Flags	Description
LX_SELALL	Select any and all items (list items only)
LX_SELLEX	Select lexical items in range
LX_SELNSLEX	Select non-stopped lexical items in range

Example for selecting items from list:

```
lxTrav *lxtv;
lxLexitem *lx;
lxtv = lx_travinit(list, LX_SELLEX);
for (lx = lx_travfirst(lxtv); lx; lx = lx_travnext(lxtv)) {
        printf("%s\n", lx_token(lx));
}
lx_travfree(lxtv);
```

Example for selecting index entries:

```
lxTrav *lxtv;
lxIndexnode *li;
lxtv = lx_travinit(list, LX_SELNSLEX);
for (li = lx_travfirst(lxtv); li; li = lx_travnext(lxtv)) {
        printf("%s\n", lx_indextoken(li));
}
lx_travfree(lxtv);
```

Special index traversal and selection[3] NOTE: the following special actions apply only to index traversal for special purposes. To select the index entries for ordinary items in the text, use the flags above.

Flags	Description
LX_SELROOTS	Select only word roots
LX_SELFORMS	Select only word forms
LX_SELUNASS	Select only word forms unassigned to any root
LX_SELORPHANS	Select only orphan roots
LX_SELINODE	Select any index node

Example selecting root forms from the index:

```
lxTrav *lxtv;
lxIndexnode *li;
lxtv = lx_travinit(list, LX_SELROOTS);
for (li = lx_travfirst(lxtv); li; li = lx_travnext(lxtv)) {
        printf("%s\n", lx_indextoken(li));
}
lx_travfree(lxtv);
```

[3] LX_SELALL should encompass LX_SELINODE, and LX_SELINODE be dropped.

4.5.2 List items

```
lxLexitem *lx_travfirst(lxTrav *lxtv);
lxLexitem *lx_travlast(lxTrav *lxtv);
lxLexitem *lx_travnext(lxTrav *lxtv);
lxLexitem *lx_travprev(lxTrav *lxtv);
```

Traverse list items.

4.5.3 Index items

```
lxIndexnode *lx_travfirstindex(lxTrav *lxtv);
lxIndexnode *lx_travlastindex(lxTrav *lxtv);
lxIndexnode *lx_travnextindex(lxTrav *lxtv);
lxIndexnode *lx_travprevindex(lxTrav *lxtv);
```

Traverse the list index.

4.5.4 First and following occurences of words

Plain word

```
lxLexitem *lx_travfirstoccurtoken(lxTrav *lxtv, char *token);
lxLexitem *lx_travfirstoccur(lxTrav *lxtv, lxIndexnode *lexinode);
lxLexitem *lx_travnextoccur(lxTrav *lxtv);
```

Return the first and then following occurences of a word in the text.

Root word

```
lxLexitem *lx_trav_r_firstoccur(lxTrav *lxtv, lxIndexnode *rootinode);
lxLexitem *lx_trav_r_nextoccur(lxTrav *lxtv);
```

Return the first and then following occurences of a form of the root in the text.[4]

4.5.5 Flagged items

```
lxLexitem *lx_travfirstflagged(lxTrav *lxtv, int flags);
lxLexitem *lx_travnextflagged(lxTrav *lxtv, int flags);
```

4.5.6 Goto item

Return the first item in the list that has the matching flag bits set.

```
lxLexitem *lx_travgoto(lxLexlist *list, int pos);
```

Return the item at position *pos*. If selection method is LX_SELALL, then the position is the hard item number. If the selection method selects lexical items (LX_SELLEX, LX_SELNSLEX) the position number is the soft number.

[4] Should have lx_trav_r_firstoccurtoken().

4.6 Omitting words (stop lists)

Words can be flagged for omission by running a stop list against the loaded text. Lx provides some built-in functions to make this simple.

```
int lx_loadstops(lxLexlist *list, char *input);
int lx_stopfromlist(lxLexlist *list, blist *wordlist);
int lx_clearstops(lxLexlist *list);
int lx_reversestops(lxLexlist *list);
```

These functions are wrappers for the more general flag-setting functions below.

Item renumbering When a stop list is run against a text, the "soft" item numbers are renumbered so as to not include the stopped items.

4.7 Setting and clearing flags

4.7.1 Setting and clearing flags for words and lists of words

```
int lx_ioflagfromlist(lxLexlist *list, char *input, int flags);
void lx_flagfromlist(lxLexlist *list, blist *wordlist, int flags);
void lx_flagnotfromlist(lxLexlist *list, blist *wordlist, int flags);
void lx_flagoccurfromlist(lxLexlist *list, blist *wordlist, int flags);
void lx_unflagall(lxLexlist *list, int flags);
```

4.7.2 Providing an external function to select and change flags

```
int lx_changelistflags(lxLexlist *list, int (*select)(int, int));
int lx_changetokenflags(lxLexitem *item, int (*select)(int, int));
```

NOT IMPLEMENTED YET

4.7.3 Renumbering

```
void lx_renumber(lxLexlist *list, int fset, int fnset);
```

4.8 Roots and forms of words

4.8.1 Loading and printing roots and forms

```
int lxf_loadforms(lxLexlist *list, char *input);
int lxf_loadformsfromio(lxLexlist *list, IO *io);
void lxf_dumpformstoio(lxLexlist *list, IO *io);
```

4.8.2 Adding and deleting roots and forms

```
lxIndexnode *lxf_addroot(lxLexlist *list, char *root);
```

Add (create) a new root word in the index. If the word already exists in the text, then it is simply marked as being a root; if the word does not already exist, it is inserted in the index as a "dangling" entry (a word that exists in the entry but not the text).

```
int lxf_delroot(lxLexlist *list, char *root);
```

Delete a root word. If the word does not exist in the text, it is not deleted, but becomes an “orphan” root. If there are dependent forms mapped, lxf_delroot() will return -1 and set lxerrnum to LXASSFORM.

```
int lxf_delforms(lxLexlist *list, char *root);
```

Delete all forms for a root word. Returns -1 on failure if *root* is not a root word.

```
int lxf_delrootandforms(lxLexlist *list, char *root);
```

Delete all forms for a root word, and then delete the root itself. Returns -1 on failure if *root* is not a root word.

```
int lxf_addformbynames(lxLexlist *list, char *root, char *form);
```

Add a form *form* to the root word *root*.

```
int lxf_addform(lxLexlist *list, lxIndexnode *rootip, char *form);
```

If the root word index node is known, *form* can be added with lxf_addform().

```
int lxf_addformbyinodes(lxIndexnode *rootip, lxIndexnode *formip);
```

And most efficient of all, if both the root and form index nodes are known, lxf_addformbyinodes() will add *formip* with no lookups at all.

```
int lxf_delform(lxLexlist *list, char *form);
```

Delete the form *form*. (Really, just disassociate it from its root.) Returns -1 if *form* does not appear in the index, its not assignedf to a root, or if it is itself a root.

4.8.3 Low-level finding and dereferencing roots and forms

```
lxIndexnode *lxf_rootindexnode(lxIndexnode *li);
```

Return pointer to index node of root; NULL if not exist.

```
lxLexitem *lxf_rootaslex(lxIndexnode *li);
```

Return pointer to the first occurence of root for *li* in text. Returns NULL if either root does not exist, or does not occur in text.

```
char *lxf_rootastoken(lxIndexnode *li);
```

Return pointer to the token string of the first occurence of the root for li. Returns NULL if root does not exist.

```
lxIndexnode *lxf_firstformforroot(lxLexlist *list, char *root);
```

Looks up root in index, returns pointer to the first form of the root.

```
lxIndexnode *lxf_firstform(lxLexlist *list, char *word);
```

Looks up word in index, finds root, and returns pointer to the first form of the word.

```
lxIndexnode *lxf_nextform(lxIndexnode *li);
```

Returns the next form of the word in the index.

```
lxIndexnode *lxf_firstassigned(lxLexlist *list, TreeCursor *tp);
```

Returns the first assigned form in the index tree.

```
lxIndexnode *lxf_nextassigned(TreeCursor *tp);
```

Returns the next assigned form in the index tree.

```
lxIndexnode *lxf_firstunassigned(lxLexlist *list, TreeCursor *tp);
```

Returns the first unassigned form (word) in the index.

```
lxIndexnode *lxf_nextunassigned(TreeCursor *tp);
```

Returns the next unassigned form (word) in the index.

```
lxIndexnode *lxf_firstorphan(lxLexlist *list, TreeCursor *tp);
```

Returns the first orphan node in the index (was a not-in-text root, then was deleted).

```
lxIndexnode *lxf_nextorphan(TreeCursor *tp);
```

Returns the next orphan node in the index (was a not-in-text root, then was deleted).

```
char *lxf_itemtorootword(lxLexlist *list, lxLexitem *item);
```

Return token string for root of word as lexical item (in lexical list).

```
char *lxf_wordtorootword(lxLexlist *list, char *word);
```

Return token string for word as string.

```
void lxf_tellform(lxIndexnode *li, IO *io);
```

Dump form information for index node (for debugging).

4.8.4 Higher-level finding and dereferencing roots and forms

```
int lxt_setderef(lxTrav *lxtv, int flags);
```

Set the dereference type flags in a traversal structure. If this is not called, the default is to perform no root look-ups (LX_DEREFTOKEN).

The possible flags are:

Flag	Description
LX_DEREFTOKEN	Return word as word itself.
LX_DEREFROOT	Return root for word OR word itself if no root

```
char *lxt_dereftoken(lxTrav *lxtv);
```

Return the root word for the current item in the traversal, or the word itself if it is not assigned to a root.

```
char *lx_dereftoken(lxLexlist *list, lxLexitem *item, int flags);
```

Return the root word for *item*, or *item* itself if not assigned to a root.

```
char *lx_derefindextoken(lxIndexnode *inode, int flags);
```

Return the root word for the index node *inode* or the word for *inode* itself if it is not assigned to a root.

4.9 Classification schemes and categories

```
lxClass *lxc_addclass(lxClass *classes, char *classname);
```

Create a classification scheme and add to the list of classification schemes. Returns a pointer to the newly created class, which is also the head of the new class list. So be careful to preserve classes:

```
if ((cp = lxc_addclass(classes, "newclass")) == NULL);
        return -1;
classes = cp;
lxClass *lxc_delclass(lxClass *classes, char *classname);
```

Delete a classification scheme from the list of classification schemes. Returns a pointer to the (possibly new) head of the list of classes.

```
int lxc_addcateg(lxClass *class, char *categ);
```

Add a category to the classification scheme. If the category already exists, lxc_addcateg() will silently return 0 for success.

```
int lxc_addcategbyname(lxClass *classes, char *classname, char *categ);
```

Add a category to the classification scheme, selecting the scheme by name.

```
int lxc_delcateg(lxClass *class, lxCatinode *cinode);
```

Delete a category from the classification scheme.

```
int lxc_delcategbyname(lxClass *classes, char *classname, char *categ);
```

Delete a category from the classification scheme, selecting the scheme by name.

```
lxCatinode *lxc_addxcateg(lxClass *class, char *categ);
```

Add a category to the classification scheme. This does not check to see if the category exists. This function is used internally, and is exported for use in things like bulk scheme loaders.

```
int lxc_addsubcateg(lxClass *class, char *categ, char *subcateg);
```

Add a subcategory to a category. If either the category or the sub-category do not already exist they will be created.

```
int lxc_addmember(lxClass *class, char *categ, char *lex);
```

Add a lexical member to a category.

NOTE: If the category doesn't exist it will be created.

```
int  lxc_addmemberbyname(lxClass *classes, char *classname, char *categ, char *lex);
```

Add a lexical member to a category, selecting the class and category by name.

```
void lxc_freecategs(lxClass *class);
```

Free the category data in a classification scheme. Use lxc_delclass() to delete the classification itself.

4.9.1 Binding classes and lexical indexes

```
int lxc_bindclass(lxClass *class, lxLexlist *lexlist);
```

Attach the classification scheme to the lexical list. This sets up the pointers class->lexlist, and lexlist->class; and then for each item in the classification index, sets a pointer (item->lexinode); to the corresponding lexical item in the lexical index, and another pointer back from that item to the classification item (lex->catinode). So each classified term points to its occurences in the text, and each lexical item in the text points to its occurence in the classification scheme.

```
int lxc_bindclassbyname(lxClass *classes, char *classname, lxText *text, char *lexname);
```

lxc_bindclass() by name of class and lexical list.

```
int lxc_releaseclass(lxClass *class);
```

Dissassociate the class from the lexical list it is bound to. (Can be called through the lexical list without looking up the class by calling lxc_releaseclass(lexlist->class).

```
int lxc_releaseclassbyname(lxClass *classes, char *classname);
```

Dissassociate the named class from the lexical list it is bound to.

```
int lxc_releaseclasslexbyname(lxText *text, char *lexname);
```

Dissassociate the named lexical list from the class it is bound to.

4.9.2 Class and category searching and retrieval

```
lxClass *lxc_findclass(lxClass *classes, char *classname);
```

Find the class with the name classname in the list of classes.

```
lxCatinode *lxc_findcatinode(lxClass *class, char *token, int flags);
```

Find the category index node that indexes "token" where the flags is one of CAT_ISLEX or CAT_ISCAT.

```
lxIndexnode *lxc_findcatinode(lxClass *class, char *token);
lxIndexnode *lxc_findcatlex(lxClass *class, char *token);
```

Useful macros that call lxc_findcatinode() with the correct flags.

- lxc_findcateg() finds *token* in class where *token* is a category name.
- lxc_findcatlex() finds *token* in class where *token* is a word.

```
lxCatinode *lxc_catfromlnode(lxCatlnode *node);
lxCatinode *lxc_mbrfromlnode(lxCatlnode *node);
```

Macros: return the category index nodes pointed to from a category list node, member and category versions.

```
lxIndexnode *lxc_lexinodefromlnode(lxCatlnode)
lxLexitem *lxc_lexfromlnode(lxCatlnode);
```

Macros: return the lexical index node or the first instance of the lexical item itself, pointed to from a category list node. NOTE: Class must be bound or will return NULL; no error set.

```
lxCatStack *lxc_newcatstack(int maxsub);
```

Instantiate a new stack structure (for use below in lxc_firstwordforcat(); and lxc_nextwordforcat().

```
void lxc_setcatstack(lxCatStack *stack, int maxsub);
```

Reset the stack, and set it's maximum depth to be maxsub. (Absolute maximum depth is CAT_MAXSUBS defined in lx_lex.h) This is as far as the category walking routines will go in tracing subcategories. Once this depth has been reached, further subcategories will not be expanded (traced), but skipped over.

```
void lxc_freecatstack(lxCatStack *stack);
```

Free a category stack.

```
lxCatlnode *lxc_firstwordforcateg(lxCatStack *stack,
               lxClass *class, char *categ);
lxCatlnode *lxc_firstwordforcat(lxCatStack *stack, lxCatinode *cip);
lxCatlnode *lxc_nextwordforcat(lxCatStack *stack, lxCatlnode *cnp);
```

lxc_firstwordforcat() returns a pointer to the node for the first word for category "categ" in class "class".

stack is a pointer to a category stack previously created with lxc_newcatstack(). The node returned contains two pointers to items in the index. One, node->catlinode, points to the index node for the lexical item. The other, node->catcinode, points to the index node for the category name. The category name can be returned by node->catcinode->token. The lexical item (the string) can be returned by node->catlinode->token.

If the class is bound to a lexical list, the lexical items occurence in the index for that list is pointed to by node->catlinode->lexinode, and thence on to the lexical stream and the occurences of the lexical item in that stream.

Example (list the words in a category):

```
lxClass    *cp;
lxCatStack *sp;
lxCatlnode *clp;
cp = findclass(lx_Classes, "dylan.class");
sp = lxc_newcatstack(10);

for (clp = lxc_firstwordforcateg(sp, cp, "light"); clp;
                                    clp = nextwordforcat(sp, lp)) {
        fprintf(stdout, "%s\n", clp->catlinode->token);
}
lxc_freecatstack(sp);
lxCatlnode *
lxc_firstcatforword(lxClass *class, char *lex);
```

Returns the first category index node containing the word pointed to by "lex".

```
lxCatlnode *lxc_nextcatforword(lxCatlnode *last);
```

Returns the next category index node after "last".

4.9.3 Loading and dumping (printing) categories

```
void lxc_dumpclasstoio(lxClass *class, IO *io);
```

Writes the category index to io. The output is the same as the input accepted by lxc_loadclassfromio() below.

```
int lxc_loadclassfromio(lxClass *class, IO *io);
```

Read categories and members from io. Category names are tokens first on a line and end in ":", subcategories are marked with a leading "+".

For example:

```
nature:
        +animals
        +vegetation
        +water
        +geology
        sky
        clouds
animals:
        +wild_animals
        +domestic_animals
wild_animals:
        bear
        deer
        lynx
domestic_animals:
        cow
        horse
        dog
```

```
        cat
vegetation:
        tree
        leaf
        grass
        flower
water:
        river
        stream
        sea
        current
        banks
geology:
        mountain
        rock
        rocky
```

More than one item can appear on a line, but if redumped by lxc_dumpclasstoio(); the output will be "expanded" as above. Lines beginning with '#' are taken to be comments and are ignored (and will also be lost on a dump):

```
# nature is upper level and included several categories.
# we could perhaps create a seperate "sky" category.
nature: +animals +vegetation +water +geology sky clouds
animals: +wild_animals +domestic_animals
wild_animals: bear deer lynx
domestic_animals: cow horse dog cat
vegetation: tree leaf grass flower
water: river stream sea current banks
geology: mountain rock rocky
```

4.10 Gathering Context for Lexical Items

```
lxContext *lx_newcontext(lxLexlist *list, int flags, int left, int right);
```

Create and return new context structure for list. Flags indicate the lexical types that will be accepted, left and right indicate how much context to gather from the the left and right sides of items. The default is to collect only non-omitted, in-range, lexical items. Flags possible are LX_SELNSLEX (default, 0), LX_SELLEX (also collect stopped items), LX_SELALL (also collect non-lexical items). The pointer to the list can be NULL: but *catcontext operations below will fail and set lxerror LXCTXNOLIST.

```
void lx_freecontext(lxContext *lc);
```

lc_freecontext() must be called to free the memory allocated by lx_newcontext().

```
int lx_getlexcontext(lxContext *lc, char *lex);
```

Find lex in the index and then return lx_getcontext().

```
int lx_getcontext(lxContext *lc, lxLexitem *item);
```

Get context for an item. Pointers to selected items left and right of the passed item are returned in the lxContext structure and can be accessed with the functions below. lx_nextcontext() can be used to return the context for the next occurence of the item.

```
int lx_nextcontext(lxContext *lc);
int lx_getcategcontext(lxContext *lc, char *categ);
```

Find categ in the class index and then return lx_getcatcontext().

```
int lx_getcatcontext(lxContext *lc, lxCatinode *catinode);
```

Get context for an item bound to the category node catinode. lx_nextcatcontext() can be used to return the context for the next occurences of the item or the next item in the category.

```
int lx_nextcatcontext(lxContext *lc);
lxLexitem *lx_lfirstfromcontext(lxContext *lc);
```

Return the leftmost item from the context lc. Resets the internal index to the "current" context item. The sequences lx_lfirstfromcontext(), lx_lnextfromcontext(), lx_rfirstfromcontext() lx_rnextfromcontext() move from left to right in the text stream, with the key item in the center.

```
+--------------------------+--------------------------+
|   |   |      |           |           |   |   |      |
l0  l1  l2 ... ln        key item       r0  r1  r2 ... rn
^   ^... lx_lnextfromcontext()          ^   ^... lx_rnextfromcontext();
|                                       |
lx_lfirstfromcontext()                  lx_rfirstfromcontext();
```

The number of items varies depending on

1. the size of the left and right contexts specified in the call to lx_newcontext();, and
2. the number of items available in context; for example, the first word in the text will have nothing to the left of it, the last nothing to the right.

```
lxLexitem *lx_lnextfromcontext(lxContext *lc);
```

Return the next item (righthand) from the context lc.

```
lxLexitem *lx_rfirstfromcontext(lxContext *lc);
```

Return the first item to the right of the context key item. Resets the internal index to the "current" context item.

```
lxLexitem *lx_rnextfromcontext(lxContext *lc);
```

Return the next item to the right in the context.

```
int lx_lnumfromcontext(lxContext *lc);
```

Return the number of items in context to the left.

```
int lx_rnumfromcontext(lxContext *lc);
```

Return the number of items in context to the right.

```
int lx_llenfromcontext(lxContext *lc);
```

Return the sum of the lengths of the items in the left-hand context.

```
int lx_rlenfromcontext(lxContext *lc);
```

Return the sum of the lengths of the items in the right-hand context.

```
void lx_dumpcontext(lxContext *lc, IO *io);
```

Print (dump) context information. For debugging.

4.10.1 Context example

```
int
getandprintcontext(lxLexlist *list, lxLexitem *lx);
{
        lxContext *lc;
        lxLexitem *lxp;
        lc = lx_newcontext(list, LX_SELNSLEX, 4, 4);
        if (!lc);
               return -1;
        if (lx_getlexcontext(lc, lx) == -1);
               return -1;
        for (lxp = lx_lfirstfromcontext(lc); lxp;
            lxp = lx_lnextfromcontext));
                printf("%s ", lx_token(lxp));
        printf("%s ", lx_token(lx));
        for (lxp = lx_rfirstfromcontext(lc); lxp;
            lxp = lx_rnextfromcontext));
                printf("%s ", lx_token(lxp));
        return 0;
}
```

NOTE: we should provide need a simple context dumper, from left to right (lx_firstfromcontext()).

4.11 Formatting and printing items

These routines provide a facility for printing items and lines of text. State is gathered in a structure called a *lxLine*. Flags set at initialization (or changed later), control how and whether items are formatted.

These routines will:

- Return the entire line of text that an item is in;
- Return the line number of text that an item is in;
- Format or skip markers and counters for printing or display;
- Format individual items.

4.11.1 Include files

lxLine structure and function definitions are not included from lx_lex.h, but rather from lx_print.h. lx_lex.h must still be included, however.

```
#include <lx_lex.h>
#include <lx_print.h>
```

4.11.2 Flags

A number of flags are defined to control the formatting of the line items that are stored into lxLine:

Flags	Description	Example output
LXP_NOMKR	no markers	
LXP_NOOMIT	no omit	
LXP_NORNG	no range markers	
LXP_NOCTR	no counters	
LXP_NOCOMM	no comments	
LXP_RNG_NAME	print range name	cordelia
LXP_RNG_MARK	print as range marker	<cordelia>
LXP_RNG_TYPE	print range type	speaker
LXP_RNG_NAME|LXP_RNG_VALUE		cordelia=speaker
LXP_RNG_MARK|LXP_RNG_VALUE		<cordelia=speaker>
LXP_CTR_VALUE	print counter value	7
LXP_CTR_NAME	print counter name	p
LXP_CTR_MARK	print as counter marker	<p>
LXP_CTR_NAME|LXP_CTR_VALUE		p7
LXP_CTR_MARK|LXP_CTR_VALUE		<p=7>
LXP_DRF_PLAIN	plain lexical item	houses
LXP_DRF_ROOT	root of lexical item	house
LXP_DRF_MROOT	marked root	[house]
LXP_DRF_MFORM	marked form	[houses]

4.11.3 Flag testing macros

```
int lxp_nomarkers(int flags)
int lxp_noranges(int flags)
int lxp_nocomments(int flags)
int lxp_nocounters(int flags)
int lxp_noomit(int flags)
```

4.11.4 Line information

```
lxLexitem *lxp_linestartitem(lxLine *line);
```

Return the item at the start of the line.

```
lxLexitem *lxp_lineenditem(lxLine *line);
```

Return the item at the end of the line.

```
int lxp_lineno(lxLine *line);
```

Return the line number of the line.

4.11.5 Line initialization and reinitialization

```
lxLine *lxp_initline(lxLexlist *list, int flags);
```

Initialize a line structure for the lexical list "list". The flags control which list items are displayed, and how things like markers and counters are displayed

```
void lxp_setlineforitem(lxLine *line, lxLexitem *li, int flags);
```

Set the line structure to return the line that li is in on the next call to lxp_fillline(). If flags = 0, then no change to the flags. Otherwise the flags are set to the passed value.

```
void lxp_resetline(lxLine *line, int flags);
```

Reset the line structure back to the start of the list. Optionally change the flags.

```
void lxp_freeline(lxLine *line);
```

Free a line structure and its associated buffer.

4.11.6 Return formatted lines and items

```
char *lxp_fillline(lxLine *line);
```

Return line with the buffer filled with a formatted line. The buffer is automatically resized as needed.

```
char *lxp_fmtitem(lxLexlist *list, lxLexitem *li, int flags);
```

Return a pointer to a formatted string representing the item's token. This may be a pointer to a static buffer (rewritten on the next call) or a pointer to the item in the list. Do not try to alter or write to this string: copy first.

4.11.7 Line information for items

These functions are used internally by the lxLine routines, but are generally useful enough that they are exposed for other use.

```
lxLexitem *lxp_getlinestart(lxLexitem *li);
```

Return the pointer item at the start of the line that *li* is in.

```
lxLexitem *lxp_getlineend(lxLexitem *li);
```

Return a pointer to the item at the end of the line that *li* is in.

```
int lxp_getlineno(lxLexitem *li);
```

Return the line number of the line that *li* is in.

4.12 Ranges

Ranges are typically defined at text load. However, new ranges can be added, changed and deleted at any time.

A range set is a list of range elements. Each range element defines a start and end text list item, and flag as to whether the range element subtracts from or adds to the range. A range set also has a name ("CORDELIA") and a type ("speaker"). The text list head maintains a list of range sets defined for that list, and a pointer to the currently applied (mapped) range set. If the pointer to the current range set is NULL, the mapped range set is the *default* range set, or everything in the list. This is also called the "*" range set.

Range expressions provide a standardized method for defining a range set, or group of range sets, as a character string. See 4.12.5 below.

4.12.1 Name of current range

```
char * lxr_current_range(lxLexlist *list);
```

Returns a pointer to the name of the current range set, or "*" if there is no limiting set (the whole text is in range).

4.12.2 Mapping ranges

```
int lxr_apply_range(lxLexlist *list, char *name);
```

Clear current mapping and apply named mapping to entire list.

```
int lxr_default_mapping(lxLexlist *list);
```

Set the range bit(s) for the entire list.

```
int lxr_invert_mapping(lxLexlist *list);
```

Invert current mapping. Sets `list->ranges.curset->invert` to note that current mapping has been inverted.[5]

4.12.3 Adding ranges

```
int lxr_add_rangeset(lxLexlist *list, char *name);
```

Create a new empty range set of name "name".

```
int lxr_adjust_range(lxLexlist *list, char *name, lxLexitem *start, lxLexitem *end, int flags)
```

Insert a new range for a range set. Flags can be `LXRNGADD`, `LXRNGSUB`

```
int lxr_append_rangeset(lxLexlist *list, char *target, char *source, int invert);
```

Append one range set to another. The source range set is not modified. If invert is specified, invert the sense of the appended ranges, `LXRNGADD`s become `LXRNGSUB`s, and vice versa.

[5] but how clear when mapping changed? unset on change? but what if changed and not applied: OR is mapping ALWAYS applied on change?)
When adjusting range, invert flag must be checked and opposite sense of mapping applied?
Might be cleaner almost to undo inversion on any change? i.e. any adjustment results in global remapping? Ugh.

```
int lxr_append_by_type_member(lxLexlist *list, char *target, char *name, char *type, int inver
```

Append the named member range set of a type: for example, "Act1" of the type "Acts". The unnamed members are explicitly reverse-mapped: e.g. if they are `LXRNGADD`s then they become `LXRNGSUB`, and so will be mapped out of range. This itself can be inverted with the invert flag. If invert is specified, invert the sense of the appended ranges, `LXRNGADD`s become `LXRNGSUB`s, and vice versa.

XX need to do more than one member: must take array or list, or do overlay where only the specified mapping type changes: only adds are added, dels are ignored, or vice versa.

```
lxRangeset *lxr_scan_to_rangeset(lxLexlist *list, char *rsname);
```

Generate a range set by scanning the lexical list and mapping items that are in range.

4.12.4 Deleting ranges

```
int lxr_del_rangeset(lxLexlist *list, char *name);
```

Delete the rangeset *name*.

```
int lxr_clear_rangeset(lxLexlist *list, char *name);
```

Delete (clear) range elements for range set.

4.12.5 Range expressions

Ranges may be expressed as character string range expressions, which lxr_parserange() returns as a range set.

```
lxRangeset *lxr_parserange(lxLexlist *list, char *rangeexpr);
```

Find, or create and store range set in *list* range sets. Except in the case where the range already exists, **the range is created with the expression as its name.** If *rangeexpr* is `"*"` (the whole list), this function returns NULL with `lxerrnum` set to `LXRDEFRNG` if . The caller may then call lxr_default_mapping(). Not pretty really.

Valid range expressions are:

Expression	Description
`CORDELIA`	Returns map for existing range set
`l=211-251`	Map from counter l where values are 211 to 251
`CORDELIA+l=211-251`	The union of both of the above
`2021-3051`	Map from items 2021 to 3051 (hard item numbers)
`CORDELIA+FOOL`	Union of the range sets CORDELIA and FOOL
`*+!CORDELIA`	Everything except CORDELIA

Future syntax will be more “logical”: `ACT_1 & (CORDELIA | "KING LEAR")`

4.12.6 Range set setup and teardown

These functions are mostly used internally at list creation or teardown.

```
int lxr_init_ranges(lxLexlist *list);
```

Initialize ranges for list. Clear and free any existing ranges and range sets, set for default range (all items).

```
int lxr_free_ranges(lxLexlist *list);
```

Free all range structures. For list teardown.

4.12.7 Finding range sets

```
lxRangeset *lxr_get_rangeset(lxLexlist *list, char *name);
```

Return pointer to rangeset *name*.

4.12.8 Walking through range sets

```
lxRangeset *lxr_first_rangeset(lxLexlist *list);
```

Return pointer to first rangeset in list of rangesets.

```
lxRangeset *lxr_next_rangeset(lxRangeset *rset);
```

Return pointer to first next rangeset following *rset*.

```
lxRangeset *lxr_first_rangeset_of_type(lxLexlist *list, char *type);
```

Return pointer to first rangeset of type *type*.

```
lxRangeset *lxr_next_rangeset_of_type(lxRangeset *rset, char *type);
```

Return pointer to first rangeset of type *type* following prevrs.

4.12.9 Utility

```
int lxr_print_range(lxRangeset *rset, IO *io);
```

Print out values of ranges in rangeset. (not in lx_lex.h?)

```
int lxr_check_ranges(lxRangeset *rset, IO *io);
```

Check all structures, references and that all range values are valid (in range!) Check that applied range matches range values.[6]

```
lxRange *lxr_makeitemrange(lxLexlist *list, int start, int end);
```

Generate a range structure for the items numbered *start* and *end*. Returns NULL on malloc error, or if start or end do not occur in the list.

4.13 Counters

Unlike range markers, counters do not define a range of text, they merely count, or provide a positional indicator. Counters are stored in a hash table in the list structure, where the key is the name of the counter, and the value a pointer to the first instance. Further instances are chained as are regular lexical items.

4.13.1 Define a new counter

```
int lx_new_counter(lxLexlist *list, char *name, lxLexitem *li);
```

Define counter *name*, at item *li*. If the counter of that name does not exist, it is created. Otherwise *li* is added to the end of the list of items for the counter.

[6] use range bit RNG1 or 2, and then compare?

4.13.2 Find a counter by name

```
lxLexitem *lx_getcounter(lxLexlist *list, char *name, int num);
```

4.13.3 Goto counter by value

```
lxLexitem *lx_skiptocounter(lxLexitem *l, int num);
```

For example, if we have a counter called *para* that marks paragraphs, to return the items at the start and end of the range of paragraphs 101-123:

```
lxLexitem *lis;
lxLexitem *lie;
lis = lx_getcounter(list, "para", 101);
lie = lx_skiptocounter(lis, 123);
```

(lx_makecounterange() in 4.13.6 below is a special impelmentation of this.)

4.13.4 Walk through counters

```
lxLexitem *lx_firstcounter(lxLexlist *list, char *name);
lxLexitem *lx_nextcounter(lxLexitem *l);
```

4.13.5 Initialise and teardown the list counters table

These functions are used internally at text load and unload.

```
int lx_init_counters(lxLexlist *list);
void lx_free_counters(lxLexlist *list);
```

4.13.6 Make range element from counter pair

```
lxRange *lx_makecounterrange(lxLexlist *list, char *name, int start, int end, int mode);
```

Special support for incorporating counters into range expressions.

4.14 Opening and closing files, pipes, and linked lists

```
IO *lx_open(char *name, char *mode);
int lx_close(IO *io);
```

lx_open() and lx_close() provide interfaces to ioopen()/ioclose() with features specific to the lx library: they interpret the name of the object to be opened to determine whether it is a file, a pipe or a linked list. They use the global lx_Lists as a home (or anchor) for named linked lists (blists).

Form of string	Object opened	Example
name	file	`lx_open("lear.txt", "r");`
|name	pipe	`lx_open("|zcat lear.txt.gz", "r");`
@name	blist (linked list);	`lx_open("@lear_tmp", "r");`

If it is asked to open a list, lx_open() looks in the global list of lists lx_Lists for an list of that name; if it can't find one there it creates one. In turn, lx_close() calls io2blist() to drop the IO structure without deleting the blist, which now remains in lx_Lists; otherwise it calls ioclose().

4.15 Stream-like Behaviour for Lexical Lists

I am not sure of the utility of these routines.

- Needs reference counts for open lists
- Needs an lx_ungetc()
- This should probably go away: use lx_print.c

4.15.1 Open, close, seek, set

```
lxStream *lx_openstream(lxLexlist *list);
```

"Open" a stream: allocate and associate a lxStream structure with a lexical lexical list.

```
lxStream *lx_dupestream(lxStream *old);
```

Duplicate a lxStream.

```
int lx_closestream(lxStream *lxs);
```

Close down and free the lxStream.

```
int lx_seekstream(lxStream *lxs, size_t offset, int whence);
```

Seek to byte offset in stream. Like fseek().

```
size_t lx_tellstream(lxStream *lxs);
```

Return byte offset in stream. Like ftell().

```
int lx_setstream(lxStream *lxs, lxLexitem *lxi);
```

Set stream position to passed lexical item. Current pointer will be positioned at start of item string. If *lxi* is NULL, stream position is unaffected.

If the *lxi* is not a member of the stream lxs, lxerrnum will be set to LXINVAL and lx_setstream() will return -1.

4.15.2 Get character, get string

```
int lx_getc(lxStream *lxs);
```

Get the next character from lxs, return character or EOF if end of stream.

XXX should set lxerrnum to something, LXEOF.

```
char *lx_gets(char *s, size_t n, lxStream *lxs);
```

Get a line into s, return s, or NULL no input on EOF. XXX should set lxerrnum to something, LXEOF.

4.15.3 Get components

```
lxLexlist *lx_listfromstream(lxStream *lxs);
```

Return pointer to the list that lxStream lxs is associated with.

```
lxLexitem *lx_lexfromstream(lxStream *lxs);
```

Return pointer to the item that lxStream lxs is positioned at.

```
int lx_flagsfromstream(lxStream *lxs);
```

Return value of flags for item that lxStream is currently positioned in.
XXX CURRENTLY UNUSED, ALWAYS RETURNS 0.

4.16 Error codes and messages

Error String	Error Code	Description
"Error 0"	NOERR	No error code not set
"System error"	LXSYSERR	System call or library error (check errno)
"Lex I/O error"	LXIOERR	IO error(lx_open(), check errno)
"No such category"	LXNOCATEG	Cannot find named category
"No such class"	LXNOCLASS	Cannot find named class
"Lexical item not found"	LXNOITEM	Cannot find lexical item in index
"No such lexical list"	LXNOLEXL	Cannot find named list
"No such text list"	LXNOTEXT	Cannot find named text (list of lists)
"No such root form"	LXNOROOT	Cannot find root form in list index
"Category already exists"	LXECATEG	Attempt to add category that already exists
"Class already exists"	LXECLASS	Attempt to add class that already exists
"Item already exists"	LXEITEM	Attempt to add lexial item already existing
"Lexical list already exists"	LXELIST	Attempt to create list that already exists
"Text already exists"	LXETEXT	Attempt to create text that already exists
"Root form already exists"	LXEROOT	Attempt to define existing root form
"Form already assigned to root"	LXEFORM	Attempt to assign form already assigned
"Class already bound",	LXCLASBND	Attempt to bind class already bound
"Lexical list already bound",	LXLISTBND	Attempt to bind to list that is already bound
"Class not bound"	LXCLASNBND	Attempt to reference a classs not bound
"Lexical list not bound"	LXLISTNBND	Attempt to reference a list not bound
"Category cannot belong to self"	LXBADSUB	Category cannot have self as sub-category
"Can’t map back to list from context"	LXCTXNOLIST	Failure to get from context to list (deleted?)
"Invalid parameter"	LXINVAL	General invalid parameter or flag
"Range set already exists"	LXREXIST	Attempt to create existing range set
"No such range set"	LXRNOEXIST	Cannot find specified range set
"Lexical item does not exist in list"	LXRBADITEM	Bad pointer to item
"Cannot invert default range"	LXRINVDEF	Cannot have nothing
"Default range in expression"	LXRDEFRNG	Expresssion evaluates to everything
"Invalid range action flag"	LXRBADFLAG	Bad flag passed
"No closing comment marker"	LXMCLOSECOMM	Saw end of text before end of comment
"Item is not a marker"	LXMNOTMKR	Item referenced as a marker, isn’t
"Counter already exists"	LXCNTEXIST	Counter of that name already exists
"No such counter"	LXCNTNOEXIST	Cannot find specified counter
"Invalid counter number"	LXCNTBADNUM	No such counter with that value
"Cannot seek backward in counters"	LXCNTSEEKBCK	Can only go forward in counters
"Item is not a root form"	LXNOTROOT	Attempt to use non-root form as root
"Item is not an assigned form"	LXNOTFORM	Attempt to find root for non-assigned form
"Root still has assigned forms",	LXASSFORM	Cannot delete root before detaching forms
"Attempt to delete form that is really root"	LXDELROOT	Attempt to delete as form what is really root

5 General Data Structures

5.1 Buffer list functions

5.1.1 Finding an existing list

```
blist *findblist(blist *head, char *name);
```

Find and return a pointer to the blist named "name" in the list of blists headed by "head".

5.1.2 Creating a new list

```
blist *newblist(blist *head, char *name, int flags);
```

Create a new blist. If `head != NULL`, then new blist is becomes the new head of the list of blists. If `name != NULL`, the list is given "name" as a name. `blist->serial` is set to the last used serial # + 1; note that other operatations (particularily draining a list) will also cause the serial # to increment. It is a number to track content, more than the list itself.

The *flags* parameter is assigned to `blist->b_flags`, and controls the list's behaviour. Possible values are:

Flag	Description
`BL_ALLOC`	duplicate (alloc) data on insertion
`BL_FIXED`	not used here, but by blstr routines; allocate fixed buffer size and copy in
`BL_LOCK`	no new additions/deletions, read-only
`BL_LOCKHD`	lock blist head structure. delblist() then only deletes contents
`BL_USRLCK`	not used by blist library, but exists for caller use

5.1.3 Freeing lists and list contents

```
blist *freeblist(blist *head, blist *blist);
```

If "head" != NULL, remove "blist" from the list headed by "head". Call freebitems to delete all the items in "blist", and then delete "blist" itself. If BL_LOCK flag is set, list will not be deleted. If BL_LOCKHD flag is set, the list contents (the items) will be deleted, but the list will remain. That is, the list will be emptied.

```
void freebitems(blist *blist);
```

Delete all the items in "blist". If BL_ALLOC flag is set for list, data is also freed.

5.1.4 Adding items to a list

```
int addbitem(blist *blist, void *data, size_t size, int where);
```

Add an item to "blist". The location of its insertion depends on the parameter "where", which may be one of:

Flag	Description
`BL_INSERT_HEAD`	Insert at head of list
`BL_INSERT_TAIL`	Insert at end of list
`BL_INSERT_NEXT`	Insert after current item
`BL_INSERT_PREV`	Insert before current item

How data in inserted also depends on "blist"'s flag settings. If BL_ALLOC is set memory is allocated for the data and it is duped, else a pointer is merely set to point to the passed in data. The cursor (list->b_cur) is set to point to the just inserted item.

5.1.5 Freeing a single item

```
void freebitem(blist *blist, bitem *bitem);
```

Extract and delete a single bitem. If BL_ALLOC flag is set for list, data is also freed.

5.1.6 Returning current cursor position

```
int eoblist(blist *list);
```

Return 1 if cursor is at tail of list. 0 otherwise. NOTE: this will return 1 if list is empty.

```
void *curbitem(blist *blist);
```

5.1.7 Walking the list

Return a pointer to the data in the current item (blist->b_cur).

```
void *firstbitem(blist *blist);
```

Return a pointer to the data in the first item in "blist"; set blist->b_cur to point to the item.

```
void *nextbitem(blist *blist);
```

Return a pointer to the data in blist->b_cur->next; set blist->b_cur to point to the item.

```
void *prevbitem(blist *blist);
```

Return a pointer to the data in blist->b_cur->prev; set blist->b_cur to point to the item.

```
void *lastbitem(blist *blist);
```

Return a pointer to the data in the last item in "blist"; set blist->b_cur to point to the item.

5.1.8 Draining a list (stack and FIFO)

```
void *drainitem(blist *blist, int whence);
```

Return a pointer to the data in the item in "blist" specified by the whence flag. Set blist->b_cur to point to the item. XXX

Values of whence are: BL_DRAIN_HEAD, BL_DRAIN_TAIL, BL_DRAIN_CUR

If blist->b_flags does not hav BL_LOCK set, the item is deleted on read. A static pointer is returned to the data, and on the next call to drainitem(), if BL_ALLOC is set, the data to which it points is freed.

drainitem() can be used to implement a fifo or stack. For a fifo, insert at one end (head or tail) and read at the other. For a stack, insert and read at the same end.

Stack

```
blist = newblist(NULL, (BL_ALLOC|BL_DESTR));
/* push */
additem(blist, data, size, BL_INSERT_HEAD);
/* pop */
data = drainitem(blist, BL_DRAIN_HEAD);
```

FIFO

```
blist = newblist(NULL, (BL_ALLOC|BL_DESTR));
/* push */
additem(blist, data, size, INSERT_HEAD);
/* pop */
data = drainitem(blist, BL_DRAIN_TAIL);
```

5.1.9 Sorting lists

```
int sortblist(blist *blist, int (*compf)(const void *, const void *));
```

Sort the list using compf to compare data. Uses qsort(). If doing really big lists, might change to mergesort.

Note that the comparison function recieves two blist **'s as parameters. A sample comparison function where the data is strings might look like:

```
int
bl_strcmp(const void *s, const void *t);
{
        bitem **s1;
        bitem **t1;
        s = (const bitem **) s1;
        t = (const bitem **) t1;
        return strcmp((*s1)->i_data, (*t1)->i_data);
}
```

5.2 Hash tables

5.2.1 Quick example

```
ht = hashnew(31, HT_REPL|HT_KALLOC, strlwr);
while (fgets(line, sizeof line, fr))
        /* XXX [...] */
hashfree(ht);
```

5.2.2 Creating a hash table

```
Hashtable *hashnew(int tablesize, int flags, char *(*fold)(char *key));
```

tablesize: number of elements in hash table. Best if a prime number.

flags: or'd combinations of:

HT_REPL unique, new replaces old

HT_KALLOC malloc space for key

HT_DALLOC malloc space for datum

fold: optional case folding function

If a folding function is specified, the hash library will run it on the passed key before hashing. Folding turns on HT_KALLOC automatically, as the folded key must be stored for comparison on retrieval. "Hash" –> "hash", "hash" must then be stored.

This means there is noticable overhead in case folding, as space must be allocated, and the key is first copied to a buffer and then fold is run. The effect is exactly like:

```
ht = hashnew(31, HT_REPL, NULL);
n = strlen(key);
if (n > blen)
        strcpy(buffer, key);
fold(buffer);
hashinsert(ht, buffer, datum, sizeof datum);
```

5.2.3 Freeing a hash table

```
void hashfree(Hashtable *htable);
```

Free hash table. Behaviour with regard to data and keys depends on HT_*ALLOC flags set. If hashinsert() malloc'd space, hashfree() will free it.

5.2.4 Inserting into a hash table

```
int hashinsert(Hashtable *htable, char *key, void *datum, size_t datumsz);
```

Insert key/datum pair into hashtable. For space to be malloc'd for datum, HT_DALLOC must be specified (in hashnew()); and datumsz must be > 0. Otherwise just the pointer *datum is stored.

If HT_REPL is not set, hashinsert() will return the defined value HT_DUPLICATEKEY if an attempt is made to insert the same key twice, so you can check for this if it matters to you. htable->cur is set to point to the inserted item. This is particularily useful if one wants to insert an item if not already there, otherwise wants a pointer to the one that is there, saving the test hashget() followed by the hashinsert().

5.2.5 Deleting from a hash table

```
int hashdel(Hashtable *htable, char *key);
```

Delete the node specified by "key" from the hashtable. If space was allocated for the key or the datum, free that as well as appropriate.

5.2.6 Looking up a key in a hash table

```
void *hashget(Hashtable *htable, char *key);
```

Return the datum associated with the key. XXX what about NULL datums, since we decided to support that? XXX

5.2.7 Walk through the contents of a hash table

```
Hashnode *hashfirst(Hashtable *ht, Hashcursor *hc);
Hashnode *hashnext(Hashcursor *hc);
```

For example:

```
Hashcursor *hn;
Hashcursor hc;
for (hn = hashfirst(ht, &hc); hn; hn = hashnext(&hc)) {
        [ ... ]
}
```

5.2.8 Hash statistics and contents

```
void hashstats(FILE *fw, Hashtable *htable);
```

Print a little summary about the hash table and it's statistics.

```
void hashprint(FILE *fw, Hashtable *htable,
void (*print)(FILE *fw, int slot, Hashnode *hp));
```

For each item in htable run (*print)().

```
void hashprintgraph(FILE *fw, int i, Hashnode *hp);
```

For above, print "." for item.

```
void hashprintvalues(FILE *fw, int i, Hashnode *hp);
```

For above, print key:datum pair for item.

5.3 2-3 Tree functions

The 2-3 tree routines provide a self-balancing search tree. This implementation allows the tree to be traversed in either direction, from lowest to highest value, or vice versa. This implementation does not currently support the deletion of single items.

5.3.1 Creating and adding to a tree

Add an item to the tree. If the tree does not exist (*rootcur* is NULL) the tree is created. The TreeCursor structure returned points to the (possibly new) root of the tree. If the key already exists in the tree, the tree is unchanged, and the TreeCursor *oldcur* if passed is set to point to it.

```
TreeCursor *treeadd(TreeCursor *rootcur, void *data,
               int (*compf)(const void *, const void *), TreeCursor *oldcur);
```

Example:

```
char line[1024];
char *cp;
TreeCursor *tree = NULL;
TreeCursor tc;
while (fgets(line, stdin, sizeof(line))) {
        if ((cp = strchr(line, '\n')) != NULL)
                *cp = '\0';
        tree = treeadd(tree, (void *)line, strcmp, &tc);
        if (tc.data)
                fprintf(stderr, "duplicate line: \"%s\"", line);
}
```

5.3.2 Deleting a tree

```
void freetree(TreeCursor *rootcur, int freedata);
```

5.3.3 Searching a tree

```
void *treeget(TreeCursor *rootcur, TreeCursor *cursor,
          void *data, int (*comp)(const void *, const void *));
```

Example:

```
char *l;
l = treeget(tree, NULL, "this is a string", strcmp);
if (l != NULL)
        printf("match!: \"%s\"", l);
```

5.3.4 Traversing a tree

```
void *treefirst(TreeCursor *rootcur, TreeCursor *treecur);
void *treelast(TreeCursor *rootcur, TreeCursor *treecur);
void *treenext(TreeCursor *treecur);
void *treeprev(TreeCursor *treecur);
```

Example:

```
char *l;
TreeCursor tc;
for (l = treefirst(tree, &tc); l; l = treenext(&tc)) {
        printf("%s\n", l);
}
```

6 Utility Functions

6.1 Utility utility functions

```
char *ut_findapp(char *filename);
```

Look in the environment for a variable that matches the extension of *filename*. This is very simple-minded. (And completely unused by lx.)

```
char *ut_mkpath2file(char *path, char *file, char *ext);
```

Create a path from the components provided. The returned pointer points to allocated storage the caller should later free().

```
int ut_checkpath(char *path);
```

Check that the provided path can be stat'd.

```
time_t ut_getmtime(char *path);
```

Get the last modification time for a file.

```
int ut_isdirectory(char *path);
```

Return 1 if *path* is a directory, 0 if not, -1 if it can't be stat()'d.

```
int ut_aresamefile(char *file1, char *file2);
```

Return 1 if *file1* and *file2* are in fact the same inode on the same device. Return 0 if they are different. Return -1 if either cannot be stat()'d.

```
void ut_nlerase(char *s);
```

Remove any trailing newline ('\n') at the end of string *s*.

```
char *ut_eatw(char *s);
```

Eat (remove) any leading white space. *s* is not changed, a pointer is returned to the first non-white space character. NULL is returned if there are no-none white space characters.

```
char *ut_eatnw(char *s);
```

Return a pointer to the first white space character, or NULL if none encountered.

```
int ut_isempty(char *s);
```

Return 1 if *s* does not contain any non-white space characters, 0 if the line is empty or only consists of white space characters.

```
void ut_trimw(char *s);
```

Trim any trailing white space.

```
char *ut_getwtoken(char **bp);
```

Get a white-space separated token. *bp* is set to point just past it, so that a further call to ui_getwtoken() will return the next token.

```
char *ut_cfill(char *buf, size_t buflen, int npad, char cpad);
```

Return *buf* filled with *npad* number of characters *cpad*. If *npad* is greater then *buflen*, *buf* is filled only to *buflen*.

```
int ut_strsubstr(char *p, char *s);
```

Return 1 if s occurs in p, -1 if not. Weird return values.

```
int ut_setstr(char **var, char *s);
```

Change the value pointed to by *var*, to that pointed to by *s*. *var* must point to allocated (malloc'd) storage; it is free()d. *s* may be NULL, in which case *var* will be free()d and set to NULL

```
int ut_varonoff(int *var, char *s);
```

Set a variable *var* to 1 or 0 depending on the string *s*. If *s* is "yes" or "on", *var* is set to 1. If *s* is "no" or "off" then *s* is set to 0. Returns 0 on success, -1 if *s* is none of the above strings.

```
char *ut_strlwr(char *s)
```

Return a pointer to *s* all lowercase. This alters *s* in place.

6.2 IO functions

Stdio-like operations for blists, files, and pipes.

These routines provide a common interface for quite different structures. Accordingly, the behaviour may vary depending on the underlying source or target of operations. For example, when the object is a list, an fputs into the middle of the list causes new list items (lines) to be inserted, rather than previous data overwritten.

Lists are always read-write.

Lists (so far) do not respond to character by character writes: the minimum write granularity is the string. However, they do keep a character pointer and can be read character by character as a stream.

```
IO *ioopen(char *name, char *mode);
```

Open an IO structure and associate it with a file or a blist structure.

name is the name of the file to open, the name to give a created list if the type of IO is blist, or the command to run if it is a pipe.

mode is the creation mode, including the IO type, (modes are similar to fopen(); modes):

Flag	Description
f	**type is file**
l	**type is a linked list**
p	**type is a pipe**
w	**open for writing**
a	**open to append**
r	**open for reading**

NOTE: A list is always effectively opened for both reading and writing.

```
int ioclose(IO *io);
```

Shutdown an io. If the output is a file, then close the file. If it is a blist, then decrement the reference count, and if the reference count is now zero, unlock the list head and delete the blist. Free the IO structure.

```
void ioflush(IO *io);
```

If the output is flushable, flush it.

```
IO *iolist2io(blist *list);
```

Create an io structure, and attach an already existing list to it. The mode will be rw.

```
IO *iofile2io(FILE *f, int type);
```

Create an io structure and associate an already open file with it. Type can be one of IO_FILE or IO_PIPE, depending on the type of file stream represented by f.

```
FILE *io2file(IO *io);
```

The inverse of iofile2io(). io2file() tears down the io structure, and returns a FILE pointer. All io context is lost and the io structure is free'd. If the io is a list type, io2file() will return NULL.

```
blist *io2blist(IO *io);
```

The inverse of iolist2io(). io2list() tears down the IO structure, and returns a pointer to the underlying blist. All IO context is lost and the IO structure is free'd. The position of the blist cursor (b_cur) is unaltered, however. If the IO is a file type, io2list() will return NULL.

```
FILE *iogetfile(IO *io);
```

Simply return a pointer to the underlying FILE stream. Does not affect the IO structure or context. iogetfile_m is a macro version of this.

```
blist *iogetlist(IO *io);
```

Simply return a pointer to the underlying blist. Does not affect the IO structure or context. iogetlist_m is a macro version of this.

```
char *iogets(char *buf, size_t len, IO *io);
```

Like fgets().

```
int iogetc(IO *io);
```

Like getc().

```
int ioungetc(int c, IO *io);
```

Like ungetc(). A limited number of characters are guaranteed. The blist ungetc guarantees 3, the file ungetc() whatever the system ungetc(); guarantees. A call to a function that affects stream-positioning (ioseek, ioputs, ioprintf); will cause the pushed characters for that stream to be discarded.

```
int ioprintf(IO *io, char *fmt, ...);
```

Like fprintf().

```
int ioputs(char *s, IO *io);
```

Like fputs().

```
int ioseek(IO *io, long offset, int whence);
```

NOT IMPLEMENTED.

```
long iotell(IO *io);
```

NOT IMPLEMENTED.

```
void iorewind(IO *io);
```

Like rewind().

```
void iocopy(char *buf, size_t len, IO *iosrc, IO *iotgt);
```

Copies iosrc to iotgt, using buf (buf needed because line oriented and don't have a ioputc() yet).

```
int iogettoken(char *buf, size_t len, IO *io);
void ioeatw(IO *io);
```

For reading and parsing word lists, etc. We do do a lot of this.

6.3 Stdio-like functions for linked lists

Functions for using linked lists with for stdio-type calls. An interface layer for use by the IO functions above.

```
#include "blist.h"
#include "listio.h"
typedef struct io_lioliststruct LIST;
struct io_lioliststruct {
        blist *bl;
        char *cp;
        long offset;
        int ur;
        char ubuf[3];
};
LIST *lopen(char *name, char *mode);               /* fopen  */
LIST *blopen(blist *bl);                           /* fdopen  */
int  lclose(LIST *list);                           /* fclose  */
int  lflush(LIST *list);                           /* fflush  */
int  listseek(LIST *list, long offset, int whence); /* fseek   */
long ltell(LIST *list);                            /* ftell   */
void lrewind(LIST *list);                          /* rewind  */
char *lgets(char *buf, size_t len, LIST *list);    /* fgets   */
int  lgetc(LIST *list);                            /* getc    */
int  lungetc(int c, LIST *list);                   /* ungetc  */
```

A file positioning call, or a write to this list will lose the pushed back characters.

```
int vlprintf(LIST *list, char *fmt, va_list ap);       /* vfprintf */
int lputs(char *s, LIST *list);                        /* fputs   */
```

6.4 Error functions

NOT COMPLETE, the eprint* functions are broken

Arrangement make little sense. Not expected to eprintf whatever into error_string, and then have to call something else to get the string. Should be eformat(errnum, fmt, ...). errstr() should return error string, but string should be loaded by errset().

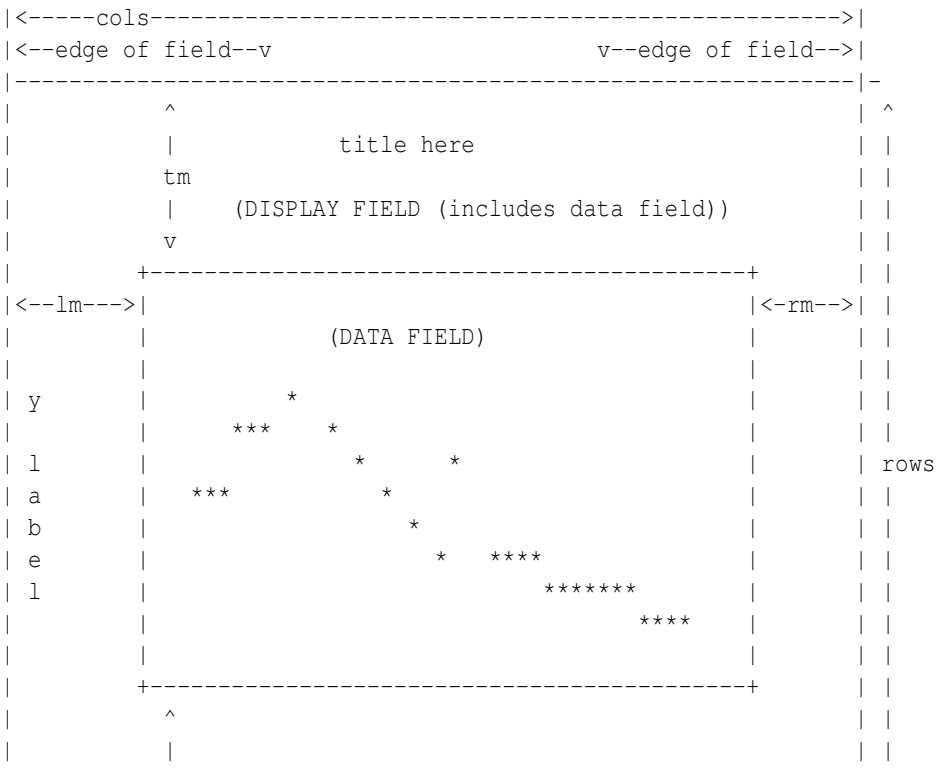
or by app wrapper?:

lxseterr(x) (lxerrnum=(x); _error_string=lx_strerr(x));

6.5 Ascii graphs

```
struct   graphdesc *ag_descgraph(
    int rows, int cols,
    int lm, int rm,
    int tm, int bm,
    char *framec,
    char *title,
    char *xlabel, char *ylabel);
```

Create a description for an ascii graph. The rows and cols parameters set the size of the display field. Margins between the display field and the data field (lm, rm, tm, bm) are measured in from the edges. framec points to the character set (string) used to draw the frame. If it is NULL, a default will be used. title and legend are optional title and label strings. If non-NULL, they are plotted into the graph on a call to frame_graph(); along with the rest of the frame. If margins are set to 0, default margin values will be used. These are 8 for the left margin, and 2 for the other three margins. This is to ensure there is room for the title, and for the scale and labels at bottom and left. NOTE: that the margins passed to init_graph() are converted to column numbers in the graphdesc structure returned. For example, a graph is created 75 columns wide, with rm passed as 2, the graph description structure will have g_rmargin set to 73.



```
|          bm                                                 | |
|          |               x label                            | |
|          v                                                  | v
|-------------------------------------------------------------|-
|<--edge of field--^                       ^--edge of field-->|
char **ag_allocgraph(struct graphdesc *gdesc);
```

Allocate and return a pointer to the character array for the display field.

```
void ag_setmaxmin(struct graphdesc *gdesc,
            float xmax, float xmin, float ymax, float ymin);
```

Set the data maximum and minimum values. These are used for scaling.

```
void ag_plotframe(struct graphdesc *gdesc, char **graph);
```

Draw the frame of the graph described by gdesc in the array pointed to by graph. If gdesc->g_title and/or gdesc->g_label are non-NULL, insert them into the graph as well.

```
void ag_plottitle(struct graphdesc *gdesc, char **graph, char *s);
```

Plot the string s into graph as the graph's title.

```
void ag_plotxlabel(struct graphdesc *gdesc, char **graph, char *s);
```

Plot the string s into graph as the graph's x axis label.

```
void ag_plotxlabel(struct graphdesc *gdesc, char **graph, char *s);
```

Plot the string s into graph as the graph's y-axis label.

```
void ag_plotsticks(struct graphdesc *gdesc, char **graph,
             float xint, float yint);
```

Plot scaled ticks into the X and Y axies of graph at intervals xint and yint.

```
int ag_plotsxy(struct graphdesc *gdesc, char **graph,
             float x, float y, char c);
```

Scale and plot the character c at x,y in the data field.

```
void ag_plotxy(char **graph, int x, int y, char c);
```

Plot the character c at x,y in the display field.

```
void ag_plothstr(char **graph, int x, int y, char *s, int len);
```

Plot horizontally at most len of string s at x,y in the display field.

```
void ag_plothcstr(char **graph, int xs, int xe, int y, char *s, int len);
```

Plot horizontally at most len of string s on row y in the display field, centred between columns xs and xe.

```
void ag_plotvstr(char **graph, int x, int y, char *s, int len);
```

Plot vertically at most len of string s at x,y in the display field. Plots from top down.

```
void ag_plotvcstr(char **graph, int xs, int xe, int y, char *s, int len);
```

Plot vertically at most len of string s in column x in the display field, centred between rows ys, and ye. Plots from top down.

```
int ag_xscale(struct graphdesc *gdesc, float x);
```

Scale x to fit within the display field. Return column to plot x into.

```
int ag_yscale(struct graphdesc *gdesc, float y);
```

Scale y to fit within the display field. Return row to plot y into.

```
extern char *ag_framec;
```

Set of default characters used to draw graph frame.

Index