

Lithp

Jerry Lawrence jsl.lithp@absynth.com

Contents

1	Audience	4
2	Introduction	4
3	The Lithp Interpreter's Main Program	5
3.1	Parser Callbacks	5
3.2	main()	5
3.3	The Source File <code>main.c</code>	7
4	The File Parser	8
4.1	List Builder	8
4.2	Tokenizer	10
5	List Manipulations	15
5.1	Structure	15
5.2	Creation and Destruction	15
5.2.1	leNew	15
5.2.2	leDelete	16
5.2.3	leWipe	16
5.3	Basic List Manips	17
5.3.1	leAddHead	17
5.3.2	leAddTail	18
5.4	Derived List Manips	18
5.4.1	leAddBranchElement	18
5.4.2	leAddDataElement	19
5.4.3	leDup	19
5.5	List Tagging	20
5.5.1	leClearTag	20
5.5.2	leTagData	20
5.5.3	leTagReplace	21
5.6	Debug Tools	22
5.6.1	leDump	22
5.6.2	leDumpEvalTree	23
5.6.3	leDumpEval	24
5.6.4	leDumpReformat	25
5.7	lists.c	27
5.8	lists.h	28

6	The Variable Mechanisms	29
6.1	List Manips	29
6.1.1	variableFind	29
6.1.2	variableFree	30
6.2	Get and Set Variables	30
6.2.1	variableSet	30
6.2.2	variableSetString	31
6.2.3	variableGet	32
6.2.4	variableGetString	32
6.3	Debug functions	32
6.4	vars.c	33
6.5	vars.h	34
7	The List Evaluator	35
7.1	Adding More Functionality	35
7.2	Callback Registry	35
7.3	Evaluator Callbacks	38
7.3.1	set	38
7.3.2	setq	40
7.3.3	cumehelper	41
7.3.4	addition	43
7.3.5	subtraction	43
7.3.6	multiplication	44
7.3.7	divide	45
7.3.8	oneplus	46
7.3.9	oneminus	47
7.3.10	?: modulus	47
7.3.11	lt: A less than B	48
7.3.12	lteq: A less than or equal to B	49
7.3.13	gt: A greater than B	49
7.3.14	gteq: A greater than or equal to B	50
7.3.15	eqsign: A equal to B	51
7.3.16	and	51
7.3.17	or	52
7.3.18	not	53
7.3.19	atom	54
7.3.20	car	55
7.3.21	cdr	56
7.3.22	cons	57
7.3.23	list	58
7.3.24	equal: similar objects?	60
7.3.25	equal: compare two lists	62
7.3.26	if	63
7.3.27	when and unless helper	63
7.3.28	unless	65
7.3.29	when	65

7.3.30	cond	66
7.3.31	princ	68
7.3.32	terpri	68
7.3.33	eval	69
7.3.34	proghelper	69
7.3.35	prog1	70
7.3.36	prog2	71
7.3.37	progn	71
7.3.38	defun	71
7.4	Utility methods	72
7.4.1	countNodes()	72
7.4.2	cast LE To Int	73
7.4.3	cast Int To LE	73
7.5	Evaluator Valves	74
7.5.1	evaluateBranch	74
7.5.2	evaluateNode	76
7.5.3	evaluateDefun	78
7.6	eval.c	81
7.7	eval.h	83
8	Sample files	85
8.1	Sample 01	85
8.2	Sample 02	85
8.3	Sample 03	86
9	Version information	92
9.1	The Source File <code>version.h</code>	94

1 Audience

The target audience for this document is those who wish to become more familiar with the inner workings of the Lithp LISP interpreter.

This is not a users guide for people new to LISP. It is implied that the reader has some familiarity with C programming as well as LISP programming, since in essence, this document is the source code to a LISP interpreter.

People wishing to integrate Lithp into their own projects will probably want to examine the included “`main.c`”, in §3 to see how it is done. If you wish to expand or reduce the functionality of Lithp, then §7 will be where you want to look.

2 Introduction

This document is broken up into a few sections. In §3, a sample main routine is implemented. In §4, the file parser is implemented. In §5, all of the list manipulation functions are implemented. In §6, the same list structures are used to store variables and user-defined functions. In §7, the list interpreter and evaluators are implemented. And finally, in §8, a few sample LISP files that work properly are given out as examples. §9 shows the current version information and changelog.

Each of these major sections begins with an introduction describing the internal layout of that section.

Most of this document is geared toward developers who wish to understand the internal workings of the interpreter.

3 The Lithp Interpreter's Main Program

This is a sample main program that uses the lithp functions to read in all files contained on the command line, and interpret each one separately.

Once the lists have been evaluated using the `leDumpEval` function, it dumps out the variable and defun lists as well.

3.1 Parser Callbacks

These are the callbacks so that we can read in from a file. You can probably write your own callbacks to read in from a buffer or the like. It expects a `stdio EOF` at the end of the file to be processed.

5a \langle *main parser globals 5a* $\rangle \equiv$ (7b)
`FILE * fp = NULL;`

First, the callback for the `getc` function...

5b \langle *main parser getc callback proto 5b* $\rangle \equiv$ (5c)
`int mygetc(void)`

5c \langle *main parser getc callback implementation 5c* $\rangle \equiv$ (7b)
 \langle *main parser getc callback proto 5b* \rangle
`{`
`return(getc(fp));`
`}`

And our callback for the `ungetc` function...

5d \langle *main parser ungetc callback proto 5d* $\rangle \equiv$ (5e)
`void myungetc(int c)`

5e \langle *main parser ungetc callback implementation 5e* $\rangle \equiv$ (7b)
 \langle *main parser ungetc callback proto 5d* \rangle
`{`
`ungetc(c, fp);`
`}`

3.2 main()

And now the main function itself.

5f \langle *main function variables 5f* $\rangle \equiv$ (7c)
`int fileno;`
`int lineno;`
`struct le * list = NULL;`

```

6a  <main check commandline 6a>≡ (7c)
      if (argc <= 1)
      {
          fprintf(stderr, "ERROR: You must specify a .lsp file!\n");
          return __LINE__;
      }

6b  <main function body 6b>≡ (7c)
      for (fileno = 0 ; fileno < argc-1 ; fileno ++ )
      {
          /* parse in the file */
          printf("==== File %02d: %s\n", fileno, argv[fileno+1]);
          fp = fopen(argv[fileno+1], "r");

          if (!fp)
          {
              fprintf(stderr, "ERROR: Couldn't open \"%s\".\n", argv[fileno+1]);
              continue;
          }
          lineno = 0;
          list = parseInFile(mygetc, myungetc, list, &lineno);

          /* evaluate the read-in lists and free */
          leDumpEval(list, 0);
          leWipe(list);

          /* display the variables and free */
          printf("Variables:\n");
          variableDump( mainVarList );
          variableFree( mainVarList );

          /* display the user-defined functions and free */
          printf("defun's:\n");
          variableDump( defunList );
          variableFree( defunList );

          fclose(fp);
          fp = NULL;
      }

```

3.3 The Source File main.c

The source file for the main program simply includes the headers for the standard C headers it uses.

```

7a  <main.c 7a>≡
      #include <stdio.h>
      #include "parser.h"
      #include "vars.h"

      All of the stuff for the parser callbacks...
7b  <main.c 7a>+≡
      <main parser globals 5a>

      <main parser ungetc callback implementation 5e>
      <main parser getc callback implementation 5c>

      Then, it includes the main routine.
7c  <main.c 7a>+≡
      int
      main( int argc, char* argv[] )
      {
          <main function variables 5f>
          <main check commandline 6a>
          <main function body 6b>
          return 0;
      }

```

4 The File Parser

The File Parser is basically a simple tokenizer of the file passed in. These tokens are pulled from the file stream in §4.2, and added to the list in §4.1.

4.1 List Builder

This is the main engine of the parser. It will use the below tokenizer to read in all elements of the file.

We will be passing in two pointers to functions. These two functions will be used to get and unget characters to the buffer or stream, or whatever input device you are using.

For the “get a character” function, we will be using this format:

```
8a  <Parse get character callback typedef 8a>≡ (14b)
      typedef int
      (*getcCallback)
      (
          void
      );
```

And to “unget a character”, we will use this format:

```
8b  <Parse unget character callback typedef 8b>≡ (14b)
      typedef void
      (*ungetcCallback)
      (
          int c
      );
```

```
8c  <Parse in file proto 8c>≡ (9 14b)
      struct le *
      parseInFile(
          getcCallback getachar,
          ungetcCallback ungetachar,
          struct le * list,
          int * line
      )
```

```

9  <Parse in file implementation 9>≡ (14a)
    <Parse in file proto 8c>
    {
        char * temp = NULL;
        enum tokename tokenid = T_NONE;
        int isquoted = 0;

        if (!getachar || !ungetachar) return( NULL );

        while (1){

            temp = snagAToken(getachar, ungetachar, &tokenid);

            switch (tokenid)
            {
            case (T_QUOTE):
                isquoted = 1;
                break;

            case (T_OPENPAREN):
                list = leAddBranchElement(
                    list,
                    parseInFile(getachar,
                                ungetachar,
                                NULL,
                                line),
                    isquoted
                );
                isquoted = 0;
                break;

            case (T_NEWLINE):
                isquoted = 0;
                *line = *line +1;
                break;

            case (T_WORD):
                list = leAddDataElement(
                    list,
                    temp,
                    isquoted
                );
                free(temp);
                isquoted = 0;
                break;
            }
        }
    }

```

```

        case (T_CLOSEPAREN):
        case (T_EOF):
            isquoted = 0;
            return (list);
    }
}
}

```

4.2 Tokenizer

Different handlings for different tokens:

whitespace: skip over **comment:** skip to end of line (: return NULL, tokenid gets OPENPAREN): return NULL, tokenid gets CLOSEPAREN **newline:** return NULL, tokenid gets NEWLINE **"foo":** return "foo", tokenid gets WORD **number:** return number in a string, tokenid gets WORD

buffers returned need to be freed later.

10a \langle Parse token enum 10a $\rangle \equiv$ (14b)

```

enum tokename {
    T_NONE,
    T_CLOSEPAREN,
    T_OPENPAREN,
    T_NEWLINE,
    T_QUOTE,
    T_WORD,
    T_EOF
};

```

10b \langle Parse snag a token proto 10b $\rangle \equiv$ (11 14b)

```

char *
snagAToken(
    getcCallback getachar,
    ungetcCallback ungetachar,
    enum tokename * tokenid
)

```

```

11  <Parse snag a token implementation 11>≡ (14a)
    <Parse snag a token proto 10b>
    {
        unsigned int pos = 0;
        int c;
        int doublequotes = 0;
        char temp[128];

        *tokenid = T_EOF;

        if (!getachar || !ungetachar)
        {
            *tokenid = T_EOF;
            return( NULL );
        }

        /* chew space to next token */
        while (1)
        {
            c = getachar();

            /* munch comments */
            if ( (c == '#')
                || (c == ';' )
                )
            {
                do {
                    c = getachar();
                } while (c != '\n');
            }

            if (( (c == '(')
                || (c == ')')
                || (c == '\t')
                || (c == '\n')
                || (c == '\"')
                || (c == '\\')
                || (c == EOF)
                || (c > '-')
                || (c <= 'z')
                ) && ( c != ' ' ) )
            {
                break;
            }
        }
    }

```

```
/* snag token */
if (c == '(')
{
    *tokenid = T_OPENPAREN;
    return( NULL );
} else

if (c == ')')
{
    *tokenid = T_CLOSEPAREN;
    return( NULL );
} else

if (c == '\')
{
    *tokenid = T_QUOTE;
    return( NULL );
} else

if (c == '\n')
{
    *tokenid = T_NEWLINE;
    return( NULL );
} else

if (c == EOF)
{
    *tokenid = T_EOF;
    return( NULL );
}

/* oh well. it looks like a string. snag to the next whitespace. */

if (c == '\"')
{
    doublequotes = 1;
    c = getachar();
}

while (1)
{
    temp[pos++] = (char) c;

    if (!doublequotes)
    {
```

```
    if ( (c == ')')
        || (c == '(')
        || (c == ';')
        || (c == '#')
        || (c == ' ')
        || (c == '\t')
        || (c == '\n')
        || (c == '\r')
        || (c == EOF)
    )
    {
        ungetachar(c);
        temp[pos-1] = '\0';

        if ( !strcmp(temp, "quote") )
        {
            *tokenid = T_QUOTE;
            return( NULL );
        }
        *tokenid = T_WORD;
        return( strdup(temp) );
    }
} else {
    switch (c)
    {
        case ( '\n' ):
        case ( '\r' ):
        case ( EOF ):
            ungetachar(c);

        case ( '\"' ):
            temp[pos-1] = '\0';
            *tokenid = T_WORD;
            return( strdup(temp) );

    }
}

c = getachar();
return( NULL );
}
```

14a *<parser.c 14a>*≡
#include "parser.h"
#include <string.h>

<Parse in file implementation 9>
<Parse snag a token implementation 11>

14b *<parser.h 14b>*≡
#include <stdio.h>
#include "lists.h"

<Parse token enum 10a>

<Parse get character callback typedef 8a>
<Parse unget character callback typedef 8b>

<Parse snag a token proto 10b>;
<Parse in file proto 8c>;

5 List Manipulations

The internal storage system that we're using for both parsed in LISP trees, as well as variables, and user-defined functions are all stored using the structures and mechanisms contained in this section.

5.1 Structure

This is the basic, multipurpose structure that we are using in this project.

If for example this node is a list of other items, then the `branch` item will be a pointer. If it is an atom of data, then the `data` item will be a pointer. In the case of the variable list, then both of these items are used.

If the data or item is quoted, then the `quoted` flag should be set to 1.

The `tag` field is used exclusively for the tagging functions, when a tree is to be marked up in some way for future processing.

This structure creates doubly-linked list, although there is nothing currently that requires this... that is to say that the `list_prev` references in this project can probably be removed with no harm done.

The `list_next` points to the next `le` structure on the same level of nesting as the current one.

```
15a <List Structure 15a>≡ (28)
      typedef struct le{
          /* either data or a branch */
          struct le * branch;
          char * data;
          int quoted;
          int tag;

          /* for the next in the list in the current parenlevel */
          struct le * list_prev;
          struct le * list_next;
      } le;
```

5.2 Creation and Destruction

We need ways to create and destroy these structures, and that is done here with the following functions:

5.2.1 leNew

This creates a new `le` structure with the passed in `text` as the data string. If no `text` is passed in, then a NULL pointer is set for it. The elements of this new item are initialized to something sane.

```
15b <List new proto 15b>≡ (16a 28)
      le * leNew(char * text)
```

16a \langle List new implementation 16a $\rangle \equiv$ (27)
 \langle List new proto 15b \rangle

```

{
    le * new = (le *) malloc (sizeof(le));
    if (new)
    {
        new->branch = NULL;
        new->data = (text)?strdup(text):NULL;
        new->quoted = 0;
        new->tag = -1;
        new->list_prev = NULL;
        new->list_next = NULL;
    }
    return( new );
}

```

5.2.2 leDelete

To delete an element, all that we need to do is to free the data pointed to in the element, and then free itself.

The `branch` and `list_prev` of the items around this will either be invalid or unreachable after this is called. This is really only useful for deleting known atoms.

16b \langle List delete proto 16b $\rangle \equiv$ (16c 28)

```

void leDelete(le * element)

```

16c \langle List delete implementation 16c $\rangle \equiv$ (27)
 \langle List delete proto 16b \rangle

```

{
    if (element)
    {
        if (element->data) free( element->data );
        free(element);
    }
}

```

5.2.3 leWipe

To delete an entire list, we will recursively call this function to delete all of the `branches`, and `next` elements in turn. This is a post-order iterator so that pointers don't get munged as we try to traverse the tree. We will free ourself after all of our descendants have been freed.

16d \langle List wipe proto 16d $\rangle \equiv$ (17a 28)

```

void leWipe(le * list)

```

```

17a  <List wipe implementation 17a>≡ (27)
      <List wipe proto 16d>
      {
          if (list)
          {
              /* free descendants */
              leWipe(list->branch);
              leWipe(list->list_next);

              /* free ourself */
              if (list->data) free( list->data );
              free( list );
          }
      }

```

5.3 Basic List Manips

The last thing we need are a few functions to add things onto the list for callers outside of this section.

5.3.1 leAddHead

This will add a new `element` onto the head of the list.

This simply takes the new `element`, appends the current `list` onto its `list_next` item, patches the `list`'s `list_prev` to point to the element, then return the element. It's just a simple insertion to the beginning of the list.

```

17b  <List add head proto 17b>≡ (17c 28)
      le * leAddHead(le * list, le * element)

17c  <List add head implementation 17c>≡ (27)
      <List add head proto 17b>
      {
          if (!element) return( list );

          element->list_next = list;
          if (list) list->list_prev = element;
          return( element );
      }

```

5.3.2 leAddTail

This will add a new `element` onto the end of the list.

We will simply go to the end of the `list` (if it exists) then tack ourselves on, modifying the `list_prev` of the `element`, as well as the `list_next` of the end of the list. Then we return the new list and we're all good.

```
18a  <List add tail proto 18a>≡ (18b 28)
      le * leAddTail(le * list, le * element)
```

```
18b  <List add tail implementation 18b>≡ (27)
      <List add tail proto 18a>
      {
          le * temp = list;

          /* if neither element or list don't
             exist return the 'new' list */
          if (!element) return( list );
          if (!list) return( element );

          /* find the end element of the list */
          while (temp->list_next)
          {
              temp = temp->list_next;
          }

          /* tack ourselves on */
          temp->list_next = element;
          element->list_prev = temp;

          /* return the list */
          return( list );
      }
```

5.4 Derived List Manips

And for ease of use, we have the following functions, which use the above.

5.4.1 leAddBranchElement

This will add on a new element onto the end of the `list` passed in, containing the `branch` passed in, with its `quoted` flag set appropriately.

```
18c  <List add branch proto 18c>≡ (19a 28)
      le * leAddBranchElement( le * list, le * branch, int quoted )
```

19a \langle List add branch implementation 19a $\rangle \equiv$ (27)
 \langle List add branch proto 18c \rangle
 {
 le * temp = leNew(NULL);
 temp->branch = branch;
 temp->quoted = quoted;
 return leAddTail(list, temp);
 }

5.4.2 leAddDataElement

This will add on a new element onto the end of the `list` passed in, containing the `data` passed in, with its `quoted` flag set appropriately.

19b \langle List add data proto 19b $\rangle \equiv$ (19c 28)
 le * leAddDataElement(le * list, char * data, int quoted)

19c \langle List add data implementation 19c $\rangle \equiv$ (27)
 \langle List add data proto 19b \rangle
 {
 le * newdata = leNew(data);
 if (newdata)
 {
 newdata->quoted = quoted;
 return leAddTail(list, newdata);
 }
 }

5.4.3 leDup

There are some cases where we want to duplicate an `le` tree. This function does exactly that.

It simply iterates over the current list, recursing down for branches, duplicating the `list` passed in. It returns the duplicated tree.

19d \langle List dup proto 19d $\rangle \equiv$ (20a 28)
 le * leDup(le * list)

20a \langle List dup implementation 20a $\rangle \equiv$ (27)
 \langle List dup proto 19d \rangle
 {
 le * temp;
 if (!list) return(NULL);

 temp = leNew(list->data);
 temp->branch = leDup(list->branch);
 temp->list_next = leDup(list->list_next);

 if (temp->list_next)
 {
 temp->list_next->list_prev = temp;
 }

 return(temp);
 }

5.5 List Tagging

For search and replace of items in a tree (for the implementation of the “defun” evaluator for example) we need a way to tag elements in a tree, and work based on these tags. The following functions accomplish this.

5.5.1 leClearTag

Set all of the tags in a list to -1.

20b \langle List tag clear proto 20b $\rangle \equiv$ (20c 28)
 void leClearTag(le * list)

20c \langle List tag clear implementation 20c $\rangle \equiv$ (27)
 \langle List tag clear proto 20b \rangle
 {
 if (!list) return;
 list->tag = -1;
 leClearTag(list->branch);
 leClearTag(list->list_next);
 }

5.5.2 leTagData

Sets all data that matches data with the tag numbered tagval.

20d \langle List tag data proto 20d $\rangle \equiv$ (21a 28)
 void leTagData(le * list, char * data, int tagval)

21a \langle List tag data implementation 21a $\rangle \equiv$ (27)
 \langle List tag data proto 20d \rangle

```

{
    if (!data || !list) return;

    while (list)
    {
        if( list->data )
        {
            if (!strcmp( list->data, data ))
            {
                list->tag = tagval;
            }
        }
        leTagData( list->branch, data, tagval );

        list = list->list_next;
    }
}

```

5.5.3 leTagReplace

Sets all nodes whose tag matches `tagval` and replaces the data/branch with the data/branch information from `newinfo`.

21b \langle List tag replace proto 21b $\rangle \equiv$ (22a 28)

```

void leTagReplace(le * list, int tagval, le * newinfo)

```

22a *<List tag replace implementation 22a>*≡ (27)
<List tag replace proto 21b>

```

{
    if (!list || !newinfo) return;

    while (list)
    {
        if( list->tag == tagval )
        {
            /* free any existing stuff */
            if ( list->data )
            {
                free( list->data );
                list->data = NULL;
            }

            /* NOTE: This next comparison might be flawed */
            if ( newinfo->list_next || newinfo->branch )
            {
                list->branch = leDup( newinfo );
                list->quoted = 1;
            }
            else if ( newinfo->data )
            {
                list->data = strdup( newinfo->data );
            }

        }
        leTagReplace( list->branch, tagval, newinfo );

        list = list->list_next;
    }
}

```

5.6 Debug Tools

These are for debug output, and can probably be removed if you're crunched for space.

5.6.1 leDump

Dump out the entire list, all pretty-like.

22b *<List dump proto 22b>*≡ (23a 28)
 void leDump(le * list, int indent)

23a \langle List dump implementation 23a $\rangle \equiv$ (27)
 \langle List dump proto 22b \rangle

```

{
    int c;
    le * temp = list;

    while (temp)
    {
        if (temp->data)
        {
            for( c=0 ; c<indent ; c++ ) printf( " " );
            printf( "%s %s\n",
                temp->data,
                (temp->quoted == 1) ? "quoted" : ""
            );
        } else {
            leDump(temp->branch, indent + 4);
        }

        temp=temp->list_next;
    }
}

```

5.6.2 leDumpEvalTree

Dump out the entire list, all pretty-like, while evaluating each node.

23b \langle List dump eval tree proto 23b $\rangle \equiv$ (24a 28)
 void leDumpEvalTree(le * list, int indent)

24a \langle List dump eval tree implementation 24a $\rangle \equiv$ (27)
 \langle List dump eval tree proto 23b \rangle

```

{
    int c;
    le * temp = list;

    while (temp)
    {
        for( c=0 ; c<indent ; c++ ) printf( " " );
        if (temp->data)
        {
            printf( "%s %s\n",
                    temp->data,
                    (temp->quoted == 1) ? "quoted" : ""
            );
        } else {
            le * le_value = evaluateBranch(temp->branch) ;
            printf( "B: %s", (temp->quoted) ? "quoted " : "" );
            leDumpReformat( stdout, le_value );
            printf( "\n" );
            leWipe(le_value);

            leDump(temp->branch, indent + 4);
        }

        temp=temp->list_next;
    }
}

```

5.6.3 leDumpEval

Dump out the entire list, all pretty-like, while evaluating each node.

24b \langle List dump eval proto 24b $\rangle \equiv$ (25a 28)
 void leDumpEval(le * list, int indent)

25a \langle List dump eval implementation 25a $\rangle \equiv$ (27)
 \langle List dump eval proto 24b \rangle

```

{
    int c;
    le * temp = list;
    le * le_value = NULL;

    while (temp)
    {
        if (temp->branch)
        {
            printf ("\n");
            leDumpReformat( stdout, temp->branch );

            printf ("\n==> ");
            le_value = evaluateBranch(temp->branch) ;
            leDumpReformat( stdout, le_value );
            leWipe(le_value);
            printf ("\n");
        }

        temp=temp->list_next;
    }
    printf("=====\n");
}

```

5.6.4 leDumpReformat

Print out the tree as a standard s-expression list (as originally read in from a file) to the FILE as defined by of.

25b \langle List dump reformat proto 25b $\rangle \equiv$ (26 28)
 void leDumpReformat(FILE * of, le * tree)

```

26  <List dump reformat implementation 26>≡ (27)
    <List dump reformat proto 25b>
    {
        le * treetemp = tree;
        int len;
        int notfirst = 0;
        char * retstring;

        if (!tree) return;

        fprintf( of, "(" );
        while (treetemp)
        {
            if (treetemp->data)
            {
                fprintf( of, "%s%s", notfirst?" ":"", treetemp->data);
                notfirst++;
            }

            if (treetemp->branch)
            {
                fprintf( of, " %s", (treetemp->quoted)? "\\':"": "");
                leDumpReformat( of, treetemp->branch );
            }

            treetemp = treetemp->list_next;
        }
        fprintf( of, ")" );
    }

```

5.7 lists.c

And finally, glue it all together in the .c file.

```
27 <lists.c 27>≡
    #include "lists.h"
    #include "eval.h"
    #include <string.h>

    <List new implementation 16a>
    <List delete implementation 16c>
    <List wipe implementation 17a>

    <List add head implementation 17c>
    <List add tail implementation 18b>

    <List add branch implementation 19a>
    <List add data implementation 19c>
    <List dup implementation 20a>

    <List tag clear implementation 20c>
    <List tag data implementation 21a>
    <List tag replace implementation 22a>

    <List dump implementation 23a>
    <List dump eval tree implementation 24a>
    <List dump eval implementation 25a>
    <List dump reformat implementation 26>
```

5.8 lists.h

And the header file as well.

```
28  <lists.h 28>≡
    #ifndef __LISTS_H__
    #define __LISTS_H__

    #include <stdio.h>

    <List Structure 15a>

    <List new proto 15b>;
    <List delete proto 16b>;
    <List wipe proto 16d>;

    <List add head proto 17b>;
    <List add tail proto 18a>;

    <List add branch proto 18c>;
    <List add data proto 19b>;
    <List dup proto 19d>;

    <List tag clear proto 20b>;
    <List tag data proto 20d>;
    <List tag replace proto 21b>;

    <List dump proto 22b>;
    <List dump eval proto 24b>;
    <List dump eval tree proto 23b>;
    <List dump reformat proto 25b>;
    #endif
```

6 The Variable Mechanisms

Rather than using another structure and linked-list system, we will just use the same list structure that we use for the files itself. This also gives us the flexibility of having a variable point to a structure or the like.

We will basically assume we have another list, called `varlist` in these methods, which will basically be a single backbone with all of the variable names. Their `branch` element will contain a pointer to the data that the variable defines.

All of these basic methods will interact with the list at the level of the list itself. That is to say, with the exception of macros, when you add a new variable or retrieve a variable, you will be handing around `le` structs.

Any data passed in will be duplicated internally where storage is involved. Any returned elements will be the new stored data bits. That is to say, that you should not free any pointers returned by these methods.

29a \langle Variable list definition 29a $\rangle \equiv$ (34)
`extern le * mainVarList;`

29b \langle Variable list initialization 29b $\rangle \equiv$ (33b)
`le * mainVarList = NULL;`

Since the mechanisms are identical for working with user-defined functions, we will store those lists in here as well, even though we don't have to.

29c \langle Defun list definition 29c $\rangle \equiv$ (34)
`extern le * defunList;`

29d \langle Defun list initialization 29d $\rangle \equiv$ (33b)
`le * defunList = NULL;`

6.1 List Manips

Some functions for working specifically with the variable lists. Since these use the same `le` structure as defined previously, but it is used a little differently, we need some functions for working with the lists. These are those functions.

6.1.1 variableFind

This will return the `le` element whose `data` matches the `key` passed in, in the variable list, `varlist`.

If it was not found, a `NULL` is returned.

29e \langle Variable find proto 29e $\rangle \equiv$ (30a 34)
`le * variableFind(le * varlist, char * key)`

30a \langle Variable find implementation 30a $\rangle \equiv$ (33b)
 \langle Variable find proto 29e \rangle

```

{
    le * temp = varlist;

    if (!varlist || !key) return( NULL );

    while (temp)
    {
        if (!strcmp(key, temp->data))
        {
            return( temp );
        }
        temp = temp->list_next;
    }

    return( NULL );
}

```

6.1.2 variableFree

Since we're using `le` lists for the variable system, the “free” function is just a macro that calls the appropriate `le` function, as seen here:

30b \langle Variable free macro 30b $\rangle \equiv$ (34)

```

#define variableFree( L ) \
    leWipe( L )

```

6.2 Get and Set Variables

And, of course, some simple methods for dealing with setting and getting of variables in the variable list.

6.2.1 variableSet

This will add a variable with the `key` and `value` passed in onto the end of the `varlist`, and then return the resulting list.

This is used when we set lists for variable values.

30c \langle Variable set proto 30c $\rangle \equiv$ (31a 34)

```

le * variableSet( le * varlist, char * key, le * value )

```

31a \langle Variable set Implementation 31a $\rangle \equiv$ (33b)
 \langle Variable set proto 30c \rangle
 {
 le * temp;

 if (!key || !value) return(varlist);

 temp = variableFind(varlist, key);
 if (temp)
 {
 leWipe(temp->branch);
 temp->branch = leDup(value);
 } else {
 temp = leNew(key);
 temp->branch = leDup(value);
 varlist = leAddHead(varlist, temp);
 }
 return(varlist);
 }

6.2.2 variableSetString

This will add a variable with the key and value passed in onto the end of the `varlist`, and then return the resulting list.

This is used when we set strings for variable values.

31b \langle Variable set string proto 31b $\rangle \equiv$ (31c 34)
 le * variableSetString(le * varlist, char * key, char * value)

31c \langle Variable set string Implementation 31c $\rangle \equiv$ (33b)
 \langle Variable set string proto 31b \rangle
 {
 le * temp;

 if (!key || !value) return(varlist);

 temp = leNew(value);

 varlist = variableSet(varlist, key, temp);

 leWipe(temp);

 return(varlist);
 }

6.2.3 variableGet

This will retrieve a variable with the `key` from the `varlist`. It will return the variable data, or a `NULL` if it was not found.

This is used when we want to retrieve list values.

32a \langle Variable get proto 32a $\rangle \equiv$ (32b 34)
`le * variableGet(le * varlist, char * key)`

32b \langle Variable get Implementation 32b $\rangle \equiv$ (33b)
 \langle Variable get proto 32a \rangle
`{`
`le * temp = variableFind(varlist, key);`
`if (temp && temp->branch)`
`return(temp->branch);`
`return(NULL);`
`}`

6.2.4 variableGetString

This will retrieve a variable with the `key` from the `varlist`. It will return the variable data, or a `NULL` if it was not found.

This is used when we want to retrieve string values.

32c \langle Variable get string proto 32c $\rangle \equiv$ (32d 34)
`char * variableGetString(le * varlist, char * key)`

32d \langle Variable get string Implementation 32d $\rangle \equiv$ (33b)
 \langle Variable get string proto 32c \rangle
`{`
`le * temp = variableFind(varlist, key);`
`if (temp`
`&& temp->branch`
`&& temp->branch->data`
`&& countNodes(temp->branch) == 1`
`)`
`return(strdup(temp->branch->data));`
`return(strdup("-1"));`
`}`

6.3 Debug functions

A simple iterator to print out all of the variables in the variable list `varlist` passed in.

32e \langle Variable dump proto 32e $\rangle \equiv$ (33a 34)
`void variableDump(le * varlist)`

33a *<Variable dump Implementation 33a>*≡ (33b)
<Variable dump proto 32e>

```

{
    le * temp = varlist;
    while (temp)
    {
        if (temp->branch && temp->data)
        {
            printf("%s \t", temp->data);
            leDumpReformat( stdout, temp->branch );
            printf("\n");
        }
        temp = temp->list_next;
    }
}

```

6.4 vars.c

Here we build up all of the above blocks into the .c file.

33b *<vars.c 33b>*≡

```

#include "vars.h"
#include <string.h>

<Variable list initialization 29b>
<Defun list initialization 29d>

<Variable find implementation 30a>

<Variable set Implementation 31a>
<Variable set string Implementation 31c>
<Variable get Implementation 32b>
<Variable get string Implementation 32d>

<Variable dump Implementation 33a>

```

6.5 vars.h

And we need to do the same for the header file as well.

```
34 <vars.h 34>≡
    #include <stdio.h>
    #include "lists.h"

    <Variable list definition 29a>
    <Defun list definition 29c>

    <Variable find proto 29e>;
    <Variable free macro 30b>;

    <Variable set proto 30c>;
    <Variable set string proto 31b>;
    <Variable get proto 32a>;
    <Variable get string proto 32c>;

    <Variable dump proto 32e>;
```

7 The List Evaluator

The list evaluator is basically a callback mechanism that traverses the list passed in, and returns a `char *` containing the result.

It will look up the first parameter of a list in the callback registry, then call that method with the list itself, without removing that head entry from it, so a list of `(foo a b c)` will trigger a callback for `foo`, which will receive the list `(foo a b c)` as the `branch` parameter.

Evaluate will also try to dereference variables if they exist.

The `evaluateBranch()` method will evaluate a complete list passed in, while `evaluateNode()` method will only evaluate the single branch passed in. `evaluateNode()` is useful for dereferencing variables or lists in a list when in the callback. It is perfectly safe to recurse in this manner.

There are also methods in this section relating to casting values to list nodes and back, as well as the evaluator callbacks for a basic LISP implementation.

7.1 Adding More Functionality

It is quite easy to add more functionality into this system. All that you need to do is to create a callback satisfying the prototype as described in the next section, then add it into the `evalTable`.

If you look at any of the following callbacks, you will see how it gets entered into the list, and some basic range checks that are done to make sure that the callback gets the right number of parameters and the like.

7.2 Callback Registry

The callbacks will get their branch as one parameter. The other parameter is the number of items on the list, including the first word. All callbacks must return a newly allocated `le` list containing the return value.

For example `(foo A B)` will get sent `3` as `argc` as well as the list `(foo A B)` as the `branch` parameter. This list is stored in `le` structures, which are just a simple tree/linked list.

```
35  <Eval callback typedef 35>≡ (83)
      typedef
      le *
      (*eval_cb)
      (
          const int argc,
          le * branch
      );
```

This is the lookup structure that we'll use to store all of our callbacks in. It is simply a list of command strings to match, as well as the function callbacks as defined above. Do note that the evaluator is currently case sensitive. That is to say that "foo" and "FOO" will get evaluated differently.

```
36  <Eval lookup struct 36>≡ (83)
      typedef struct evalLookupNode {
          char    * word;
          eval_cb  callback;
      } evalLookupNode;
```

And now, here is the list of builtin functions that we support.. The final element must be a pairing of NULLs so that the lookup function knows where to stop when looking through the table.

```

37  <Eval lookup table 37>≡ (81)
      evalLookupNode evalTable[] =
      {
        { "+"          , eval_cb_add          },
        { "-"          , eval_cb_subtract      },
        { "*"          , eval_cb_multiply      },
        { "/"          , eval_cb_divide        },

        { "1+"        , eval_cb_oneplus       },
        { "1-"        , eval_cb_oneminus     },

        { "%"          , eval_cb_modulus       },

        { "<"          , eval_cb_lt            },
        { "<="        , eval_cb_lt_eq        },
        { ">"          , eval_cb_gt            },
        { ">="        , eval_cb_gt_eq        },
        { "="          , eval_cb_eqsign       },

        { "and"        , eval_cb_and           },
        { "or"         , eval_cb_or            },
        { "not"        , eval_cb_not           },
        { "null"       , eval_cb_not           },

        { "atom"       , eval_cb_atom          },
        { "car"        , eval_cb_car           },
        { "cdr"        , eval_cb_cdr           },
        { "cons"       , eval_cb_cons          },
        { "list"       , eval_cb_list          },
        { "equal"      , eval_cb_equal         },

        { "if"         , eval_cb_if            },
        { "unless"     , eval_cb_unless        },
        { "when"       , eval_cb_when          },
        { "cond"       , eval_cb_cond          },

        { "princ"      , eval_cb_princ         },
        { "terpri"     , eval_cb_terpri        },

        { "eval"       , eval_cb_eval          },
        { "prog1"      , eval_cb_prog1         },
        { "prog2"      , eval_cb_prog2         },
      }

```

```

        { "progn"      , eval_cb_progn      },
        { "set"       , eval_cb_set       },
        { "setq"      , eval_cb_setq      },
        { "setf"      , eval_cb_setq      },

        { "defun"     , eval_cb_defun     },

        { NULL        , NULL              }
};

```

7.3 Evaluator Callbacks

These callbacks will get the raw branch for which they should process. The first element on the list is the keyword for which we were called. The remaining elements are the list parameters to be used. Each parameter used should get evaluated using the `evaluateNode()` function.

7.3.1 set

The basic param list for `set` is; (`set key value key value ...`). So we need to skip to the next element on the list, then start setting variables. both the key and variable will get evaluated.

```

38  <Eval cb set proto 38>≡ (39 83)
      le * eval_cb_set( const int argc, le * branch )

```

```

39  <Eval cb set implementation 39>≡ (81)
    <Eval cb set proto 38>
    {
        le * newkey;
        le * newvalue;
        le * current;

        if (!branch || argc < 3) return( leNew( "NIL" ) );

        current = branch->list_next;
        while ( current )
        {
            if (!current->list_next)
            {
                newvalue = leNew( "NIL" );
            } else {
                newvalue = evaluateNode(current->list_next);
            }

            newkey = evaluateNode(current);

            mainVarList = variableSet(
                mainVarList,
                /* this line is diff from setq */
                newkey->data,
                newvalue
            );

            leWipe(newkey);

            if (!current->list_next)
            {
                current = NULL;
            } else {
                current = current->list_next->list_next;
            }
        }
        return( leDup(newvalue) );
    }

```

7.3.2 setq

The basic param list for `setq` is; (`setq key value key value ...`). So we need to skip to the next element on the list, then start setting variables. The key portion of this pairing is not evaluated, while the value is.

40a $\langle \text{Eval cb setq proto 40a} \rangle \equiv$ (40b 83)
`le * eval_cb_setq(const int argc, le * branch)`

40b $\langle \text{Eval cb setq implementation 40b} \rangle \equiv$ (81)
 $\langle \text{Eval cb setq proto 40a} \rangle$
`{`
`le * newvalue;`
`le * current;`

`if (!branch || argc < 3) return(leNew("NIL"));`

`current = branch->list_next;`
`while (current)`
`{`
`if (!current->list_next)`
`{`
`newvalue = leNew("NIL");`
`} else {`
`newvalue = evaluateNode(current->list_next);`
`}`

`mainVarList = variableSet(`
`mainVarList,`
`current->data,`
`newvalue`
`);`

`if (!current->list_next)`
`{`
`current = NULL;`
`} else {`
`current = current->list_next->list_next;`
`}`
`}`
`return(leDup(newvalue));`
`}`

7.3.3 cumehelper

Since Add, Subtract, Multiply and Divide are all basically the same function, with only a small difference, we will abstract out their commonalities into this “cume helper” method. It basically accumulates in whichever style is defined by the `function` parameter. It starts off by setting the accumulator to the `value` passed in, and works along the `branch` passed in. The result value is returned.

To define which mathematical function needs to be done, we will pass in the `function` as one of the following enum values. It is pretty evident which one specifies what.

```
41a  <Eval cb cume helper enum 41a>≡ (83)
      enum cumefcn { C_NONE, C_ADD, C_SUBTRACT, C_MULTIPLY, C_DIVIDE };

41b  <Eval cb cume helper proto 41b>≡ (42 83)
      int
      eval_cume_helper(
          enum cumefcn function,
          int value,
          le * branch
      )
```

The basic methodology here is that while there is a parameter on the list, we evaluate it, then cast it to an integer, then accumulate it onto `value`. Once we've run out of parameters, return the `value`.

```

42  <Eval cb cume helper implementation 42>≡ (81)
    <Eval cb cume helper proto 41b>
    {
        int newvalue = 0;
        int first = 1;
        le * temp = branch;
        le * value_le;
        char * tval;
        if (!branch) return( 0 );

        while (temp)
        {
            value_le = evaluateNode(temp);
            newvalue = evalCastLeToInt(value_le);
            leWipe(value_le);

            switch(function)
            {
            case( C_ADD ):
                value += newvalue;
                break;

            case( C_SUBTRACT ):
                value -= newvalue;
                break;

            case( C_MULTIPLY ):
                value *= newvalue;
                break;

            case( C_DIVIDE ):
                value /= newvalue;
                break;
            }

            temp = temp->list_next;
        }

        return( value );
    }

```

7.3.4 addition

This handles lists such as (+ 2 3), (+ 9 foo), and so on. We simply check to see how many parameters are passed in, and if it is valid for our purposes, call the above `cume helper`.

```

43a  <Eval cb add proto 43a>≡ (43b 83)
      le * eval_cb_add( const int argc, le * branch )

43b  <Eval cb add implementation 43b>≡ (81)
      <Eval cb add proto 43a>
      {
          char * returnval = NULL;

          if (!branch || argc < 2) return( leNew( "NIL" ) );

          return( evalCastIntToLe(
                          eval_cume_helper(
                              C_ADD,
                              0,
                              branch->list_next
                          )
                      ) );
      }

```

7.3.5 subtraction

This handles lists such as (- 2 3), (- 9 foo), and so on. We simply check to see how many parameters are passed in, and if it is valid for our purposes, call the above `cume helper`.

We have to do an extra check in here to handle items such as (- 2) where we just need to multiply the parameter by negative one and return it.

```

43c  <Eval cb subtract proto 43c>≡ (44a 83)
      le * eval_cb_subtract( const int argc, le * branch )

```

44a \langle *Eval cb subtract implementation 44a* $\rangle \equiv$ (81)
 \langle *Eval cb subtract proto 43c* \rangle

```

{
    int firstitem = 0;
    le * lefirst;
    char * tval;

    if (!branch || argc < 2) return( leNew( "NIL" ) );

    lefirst = evaluateNode( branch->list_next );
    firstitem = evalCastLeToInt( lefirst );
    leWipe( lefirst );

    if (argc == 2)
    {
        return( evalCastIntToLe( -1 * firstitem ) );
    }

    return( evalCastIntToLe(
        eval_cume_helper(
            C_SUBTRACT,
            firstitem,
            branch->list_next->list_next
        )
    ) );
}

```

7.3.6 multiplication

This handles lists such as $(* 2 3)$, $(* 9 foo)$, and so on. We simply check to see how many parameters are passed in, and if it is valid for our purposes, call the above `cume helper`.

Since we're accumulating multiplications, we need to start this one off with a 1 rather than the 0 above, since we must start off the accumulator with the multiplicative identity, and not the additive identity.

44b \langle *Eval cb multiply proto 44b* $\rangle \equiv$ (45a 83)

```

le * eval_cb_multiply( const int argc, le * branch )

```

```

45a  <Eval cb multiply implementation 45a>≡ (81)
      <Eval cb multiply proto 44b>
      {
          if (!branch || argc < 2) return( leNew( "NIL" ) );

          return( evalCastIntToLe(
                      eval_cume_helper(
                          C_MULTIPLY,
                          1,
                          branch->list_next
                      )
                  ) );
      }

```

7.3.7 divide

This handles lists such as `(/ 2 3)`, `(/ 9 foo)`, and so on. We simply check to see how many parameters are passed in, and if it is valid for our purposes, call the above `cume helper`.

Since we're accumulating divisions, we need to start this one off with the first number passed in as the initial value.

One thing that we do not currently support is `(/ 2)` which should yield 0.5 or "one half"... inverses of numbers. Since all math is currently integer based, and not real based, it wouldn't make sense to implement this yet.

```

45b  <Eval cb divide proto 45b>≡ (46a 83)
      le * eval_cb_divide( const int argc, le * branch )

```

46a \langle *Eval cb divide implementation 46a* $\rangle \equiv$ (81)
 \langle *Eval cb divide proto 45b* \rangle

```

{
    int firstitem = 0;
    le * lefirst;
    if (!branch || argc < 2) return( leNew( "NIL" ) );

    lefirst = evaluateNode( branch->list_next );
    firstitem = evalCastLeToInt( lefirst );
    leWipe( lefirst );

    return( evalCastIntToLe(
        eval_cume_helper(
            C_DIVIDE,
            firstitem,
            branch->list_next->list_next
        )
    ) );
}

```

7.3.8 oneplus

This handles lists such as (1+ 3), (1+ foo), and so on.

This basically just converts the evaluated parameter to an integer, increments it, then returns that value back to the caller.

46b \langle *Eval cb oneplus proto 46b* $\rangle \equiv$ (46c 83)

```

le * eval_cb_oneplus( const int argc, le * branch )

```

46c \langle *Eval cb oneplus implementation 46c* $\rangle \equiv$ (81)
 \langle *Eval cb oneplus proto 46b* \rangle

```

{
    le * retle;
    int value;

    if (!branch || argc < 2) return( leNew( "NIL" ) );

    retle = evaluateNode( branch->list_next );
    value = evalCastLeToInt( retle );
    leWipe( retle );

    return( evalCastIntToLe( value + 1 ) );
}

```

7.3.9 oneminus

This handles lists such as (1- 3), (1- foo), and so on.

This basically just converts the evaluated parameter to an integer, decrements it, then returns an atom containing that value back to the caller.

47a \langle Eval cb oneminus proto 47a $\rangle \equiv$ (47b 83)
`le * eval_cb_oneminus(const int argc, le * branch)`

47b \langle Eval cb oneminus implementation 47b $\rangle \equiv$ (81)
 \langle Eval cb oneminus proto 47a \rangle

```
{
    le * retle;
    int value;

    if (!branch || argc < 2) return( leNew( "NIL" ) );

    retle = evaluateNode( branch->list_next );
    value = evalCastLeToInt( retle );
    leWipe( retle );

    return( evalCastIntToLe( value - 1 ) );
}
```

7.3.10 %: modulus

This handles lists such as (\% 2 3), (\% 9 foo), and so on.

We simply convert the two evaluated parameters to integers, then return an atom with the modulus of those numbers back to the caller.

47c \langle Eval cb modulus proto 47c $\rangle \equiv$ (48a 83)
`le * eval_cb_modulus(const int argc, le * branch)`

48a \langle Eval cb modulus implementation 48a $\rangle \equiv$ (81)
 \langle Eval cb modulus proto 47c \rangle
 {
 le * letemp;
 int value1, value2;

 if (!branch || argc != 3) return(leNew("NIL"));

 letemp = evaluateNode(branch->list_next);
 value1 = evalCastLeToInt(letemp);
 leWipe(letemp);

 letemp = evaluateNode(branch->list_next->list_next);
 value2 = evalCastLeToInt(letemp);
 leWipe(letemp);

 return(evalCastIntToLe (value1 % value2));
 }

7.3.11 lt: A less than B

This handles lists such as (< 2 3), (< 9 foo), and so on.

The values of the two parameters are evaluated, then compared with each other. If the first is less than the second, a "T" atom is returned, otherwise a "NIL" atom is returned.

48b \langle Eval cb lt proto 48b $\rangle \equiv$ (48c 83)
 le * eval_cb_lt(const int argc, le * branch)

48c \langle Eval cb lt implementation 48c $\rangle \equiv$ (81)
 \langle Eval cb lt proto 48b \rangle
 {
 le * letemp;
 int value1, value2;

 if (!branch || argc != 3) return(leNew("NIL"));

 letemp = evaluateNode(branch->list_next);
 value1 = evalCastLeToInt(letemp);
 leWipe(letemp);

 letemp = evaluateNode(branch->list_next->list_next);
 value2 = evalCastLeToInt(letemp);
 leWipe(letemp);

 return(leNew ((value1 < value2) ? "T" : "NIL"));
 }

7.3.12 lteq: A less than or equal to B

This handles lists such as (`<= 2 3`), (`<= 9 foo`), and so on.

The values of the two parameters are evaluated, then compared with each other. If the first is less than or equal to the second, a “T” atom is returned, otherwise a “NIL” atom is returned.

49a $\langle \textit{Eval cb lt eq proto 49a} \rangle \equiv$ (49b 83)
`le * eval_cb_lt_eq(const int argc, le * branch)`

49b $\langle \textit{Eval cb lt eq implementation 49b} \rangle \equiv$ (81)
 $\langle \textit{Eval cb lt eq proto 49a} \rangle$

```

{
    le * letemp;
    int value1, value2;

    if (!branch || argc != 3 ) return( leNew( "NIL" ) );

    letemp = evaluateNode( branch->list_next );
    value1 = evalCastLeToInt( letemp );
    leWipe( letemp );

    letemp = evaluateNode( branch->list_next->list_next );
    value2 = evalCastLeToInt( letemp );
    leWipe( letemp );

    return( leNew ( (value1 <= value2 )?"T":"NIL" ) );
}

```

7.3.13 gt: A greater than B

This handles lists such as (`> 2 3`), (`> 9 foo`), and so on.

The values of the two parameters are evaluated, then compared with each other. If the first is greater than the second, a “T” atom is returned, otherwise a “NIL” atom is returned.

49c $\langle \textit{Eval cb gt proto 49c} \rangle \equiv$ (50a 83)
`le * eval_cb_gt(const int argc, le * branch)`

50a \langle Eval cb gt implementation 50a $\rangle \equiv$ (81)
 \langle Eval cb gt proto 49c \rangle
 {
 le * letemp;
 int value1, value2;

 if (!branch || argc != 3) return(leNew("NIL"));

 letemp = evaluateNode(branch->list_next);
 value1 = evalCastLeToInt(letemp);
 leWipe(letemp);

 letemp = evaluateNode(branch->list_next->list_next);
 value2 = evalCastLeToInt(letemp);
 leWipe(letemp);

 return(leNew ((value1 > value2)?"T":"NIL"));
 }

7.3.14 gteq: A greater than or equal to B

This handles lists such as (\geq 2 3), (\geq 9 foo), and so on.

The values of the two parameters are evaluated, then compared with each other. If the first is greater than or equal to the second, a “T” atom is returned, otherwise a “NIL” atom is returned.

50b \langle Eval cb gt eq proto 50b $\rangle \equiv$ (50c 83)
 le * eval_cb_gt_eq(const int argc, le * branch)

50c \langle Eval cb gt eq implementation 50c $\rangle \equiv$ (81)
 \langle Eval cb gt eq proto 50b \rangle
 {
 le * letemp;
 int value1, value2;

 if (!branch || argc != 3) return(leNew("NIL"));

 letemp = evaluateNode(branch->list_next);
 value1 = evalCastLeToInt(letemp);
 leWipe(letemp);

 letemp = evaluateNode(branch->list_next->list_next);
 value2 = evalCastLeToInt(letemp);
 leWipe(letemp);

 return(leNew ((value1 \geq value2)?"T":"NIL"));
 }

7.3.15 eqsign: A equal to B

This handles lists such as (= 2 3), (= 9 foo), and so on.

The values of the two parameters are evaluated, then compared with each other. If the first is equal to the second, a “T” atom is returned, otherwise a “NIL” atom is returned.

51a \langle Eval cb eqsign proto 51a $\rangle \equiv$ (51b 83)
`le * eval_cb_eqsign(const int argc, le * branch)`

51b \langle Eval cb eqsign implementation 51b $\rangle \equiv$ (81)
 \langle Eval cb eqsign proto 51a \rangle
`{`
`le * letemp;`
`int value1, value2;`

`if (!branch || argc != 3) return(leNew("NIL"));`

`letemp = evaluateNode(branch->list_next);`
`value1 = evalCastLeToInt(letemp);`
`leWipe(letemp);`

`letemp = evaluateNode(branch->list_next->list_next);`
`value2 = evalCastLeToInt(letemp);`
`leWipe(letemp);`

`return(leNew ((value1 == value2)?"T":"NIL"));`
`}`

7.3.16 and

This handles lists such as (and A B), (and A (...)), and so on.

Evaluate all of the arguments until one of them yields a NIL, then return NIL. The remaining parameters are not evaluated. If none evaluates to NIL, then the last one’s evaluation is returned.

51c \langle Eval cb and proto 51c $\rangle \equiv$ (52a 83)
`le * eval_cb_and(const int argc, le * branch)`

52a \langle Eval cb and implementation 52a $\rangle \equiv$ (81)
 \langle Eval cb and proto 51c \rangle

```

{
    le * temp;
    le * result = NULL;
    if (!branch || argc < 2 ) return( leNew( "NIL" ) );

    temp = branch->list_next;
    while( temp )
    {
        if( result ) leWipe( result );

        result = evaluateNode(temp);
        if (result->data)
        {
            if (!strcmp ( result->data, "NIL" ) )
            {
                return( result );
            }
        }
        temp = temp->list_next;
    }
    return( result );
}

```

7.3.17 or

Evaluate all of the arguments until one of them yields a non-NIL, then return their value. The remaining arguments are not evaluated. If all parameters evaluate to NIL, then a NIL atom is returned.

52b \langle Eval cb or proto 52b $\rangle \equiv$ (53a 83)
`le * eval_cb_or(const int argc, le * branch)`

53a \langle *Eval cb or implementation 53a* $\rangle \equiv$ (81)
 \langle *Eval cb or proto 52b* \rangle

```

{
    le * temp;
    le * result = NULL;
    if (!branch || argc < 2 ) return( leNew( "NIL" ) );

    temp = branch->list_next;
    while( temp )
    {
        if( result ) leWipe( result );

        result = evaluateNode(temp);
        if (result->data)
        {
            if (strcmp ( result->data, "NIL" ))
            {
                return( result );
            }
        }
        temp = temp->list_next;
    }
    return( result );
}

```

7.3.18 not

If the evaluated parameter yields a true value (perhaps T) then return a NIL atom. If the evaluated parameter yields a NIL, then we return a T atom.

53b \langle *Eval cb not proto 53b* $\rangle \equiv$ (54a 83)

```

le * eval_cb_not( const int argc, le * branch )

```

54a \langle Eval cb not implementation 54a $\rangle \equiv$ (81)
 \langle Eval cb not proto 53b \rangle

```

{
    le * result = NULL;
    if (!branch || argc != 2 ) return( leNew( "NIL" ) );

    result = evaluateNode(branch->list_next);

    if (result->data)
    {
        if (!strcmp (result->data, "NIL" ))
        {
            leWipe( result );
            return( leNew( "T" ) );
        } else {
            leWipe( result );
            return( leNew( "NIL" ) );
        }
    } else if (result->branch) {
        leWipe( result );
        return( leNew( "NIL" ) );
    }

    leWipe( result );
    return( leNew( "T" ) );
}

```

7.3.19 atom

If the evaluated parameter is a list, return a NIL atom, otherwise return a T atom.

54b \langle Eval cb atom proto 54b $\rangle \equiv$ (55a 83)
`le * eval_cb_atom(const int argc, le * branch)`

55a $\langle \text{Eval cb atom implementation 55a} \rangle \equiv$ (81)
 $\langle \text{Eval cb atom proto 54b} \rangle$

```

{
    le * result = NULL;
    if (!branch || argc != 2 ) return( leNew( "NIL" ) );

    result = evaluateNode(branch->list_next);

    if (countNodes(result) == 1)
    {
        leWipe( result );
        return( leNew( "T" ) );
    }
    return( leNew( "NIL" ) );
}

```

7.3.20 car

Return the topmost atom of the list passed in. If an atom was passed in, then we simply return it.

There is some extra magic in here to dereference nesting by one layer since we're returning the atoms from the passed in list, rather than just sublists, like what CDR does.

55b $\langle \text{Eval cb car proto 55b} \rangle \equiv$ (56a 83)

```

le * eval_cb_car( const int argc, le * branch )

```

```

56a  <Eval cb car implementation 56a>≡ (81)
      <Eval cb car proto 55b>
      {
        le * result = NULL;
        le * temp = NULL;
        if (!branch || argc != 2 ) return( leNew( "NIL" ) );

        result = evaluateNode(branch->list_next);

        if( result == NULL ) return( leNew( "NIL" ) );

        if (countNodes(result) <= 1)
        {
          if (result->branch)
          {
            temp = result;
            result = result->branch;
            temp->branch = NULL;
            leWipe( temp );
          }
          return( result );
        }

        result->list_next->list_prev = NULL;
        leWipe( result->list_next );
        result->list_next = NULL;

        if (result->branch)
        {
          temp = result;
          result = result->branch;
          temp->branch = NULL;
          leWipe( temp );
        }

        return( result );
      }

```

7.3.21 cdr

Return all but the topmost atom of the passed in list after it has been evaluated. If the list contains just one entry, we instead return a NIL atom.

```

56b  <Eval cb cdr proto 56b>≡ (57a 83)
      le * eval_cb_cdr( const int argc, le * branch )

```

57a \langle Eval cb cdr implementation 57a $\rangle \equiv$ (81)
 \langle Eval cb cdr proto 56b \rangle

```

{
    le * result = NULL;
    le * temp = NULL;
    if (!branch || argc != 2 ) return( leNew( "NIL" ) );

    result = evaluateNode(branch->list_next);

    if( result == NULL ) return( leNew( "NIL" ) );

    if (result == NULL || countNodes(result) == 1)
    {
        return( leNew( "NIL" ) );
    }

    temp = result;
    temp->list_next->list_prev = NULL;
    result = result->list_next;

    temp->list_next = NULL;
    leWipe( temp );

    return( result );
}

```

7.3.22 cons

Evaluate the two parameters, then add the first parameter, an atom, onto the second parameter, a list. Cons does the opposite of CAR and CDR.

57b \langle Eval cb cons proto 57b $\rangle \equiv$ (58a 83)
 $le * eval_cb_cons(const int argc, le * branch)$

58a \langle Eval cb cons implementation 58a $\rangle \equiv$ (81)
 \langle Eval cb cons proto 57b \rangle

```

{
    le * result1 = NULL;
    le * result2 = NULL;

    if (!branch || argc != 3 ) return( leNew( "NIL" ));

    result1 = evaluateNode(branch->list_next);
    if ( result1 == NULL ) return( leNew( "NIL" ));

    result2 = evaluateNode(branch->list_next->list_next);
    if ( result2 == NULL )
    {
        leWipe( result1 );
        return( leNew( "NIL" ));
    }

    if ( countNodes(result1) > 1 )
    {
        le * temp = leNew( NULL );
        temp->branch = result1;
        result1 = temp;
    }
    result1->list_next = result2;
    result2->list_prev = result1;

    return( result1 );
}

```

7.3.23 list

Return a list generated by joining all of the passed in parameters, after evaluating them. Each of the parameters are treated like atoms being joined into the final list.

58b \langle Eval cb list proto 58b $\rangle \equiv$ (59 83)
`le * eval_cb_list(const int argc, le * branch)`

```

59  <Eval cb list implementation 59>≡ (81)
    <Eval cb list proto 58b>
    {
        le * currelement = NULL;
        le * finaltree = NULL;
        le * lastadded = NULL;
        le * result = NULL;

        if (!branch) return( leNew( "NIL" ));

        currelement = branch->list_next;
        while (currelement)
        {
            result = evaluateNode(currelement);
            if ( result == NULL )
            {
                leWipe( finaltree );
                return( leNew( "NIL" ));
            }

            if( countNodes(result) > 1)
            {
                le * temp = leNew( NULL );
                temp->branch = result;
                result = temp;
            }

            if (!finaltree)
            {
                finaltree = result;
                lastadded = result;
            } else {
                lastadded->list_next = result;
                result->list_prev = lastadded;
                lastadded = result;
            }

            currelement = currelement->list_next;
        }

        if (!finaltree)
        {
            return( leNew( "NIL" ));
        }
        return( finaltree );
    }

```

```
}
```

7.3.24 equal: similar objects?

This is a helper function for the following `equal` callback. This gets recursively called by itself. It basically traverses the `list1` tree, comparing it to `list2`, making sure it has the same structure and elements in it.

If the lists are the same, it returns a 1. If they differ, it will return a 0.

```
60 <Eval cb equal helper proto 60>≡ (61 83)
    int eval_cb_lists_same( le * list1, le * list2 )
```

```

61  <Eval cb equal helper implementation 61>≡ (81)
    <Eval cb equal helper proto 60>
    {
        if (!list1 && !list2)    return( 1 );

        while( list1 )
        {
            /* if list2 ended prematurely, fail */
            if (list2 == NULL)
            {
                return( 0 );
            }

            /* if one has data and the other doesn't, fail */
            if ( (list1->data && ! list2->data)
                || (list2->data && ! list1->data))
            {
                return( 0 );
            }

            /* if the data is different, fail */
            if (list1->data && list2->data)
            {
                if (strcmp( list1->data, list2->data ))
                {
                    return( 0 );
                }
            }

            /* if one is quoted and the other isn't, fail */
            if (list1->quoted != list2->quoted)
            {
                return( 0 );
            }

            /* if their branches aren't the same, fail */
            if (!eval_cb_lists_same( list1->branch, list2->branch ))
            {
                return( 0 );
            }

            /* try the next in the list */
            list1 = list1->list_next;
            list2 = list2->list_next;
        }
    }

```

```

        /* if list2 goes on, fail */
        if (list2)
        {
            return( 0 );
        }

        return( 1 );
    }

```

7.3.25 equal: compare two lists

Evaluate the two parameters, and compare them to see if they have the same structure and elements. We will just call the above ...`same()` method which will return a 1 if they are the same.

We then will return a T atom if they were the same, and a NIL if they were different.

62a \langle Eval cb equal proto 62a $\rangle \equiv$ (62b 83)
 le * eval_cb_equal(const int argc, le* branch)

62b \langle Eval cb equal implementation 62b $\rangle \equiv$ (81)

```

    {
        le * letemp;
        le * list1 = NULL;
        le * list2 = NULL;
        int retval = 0;

        if (!branch || argc != 3 ) return( leNew( "NIL" ) );

        list1 = evaluateNode( branch->list_next );
        list2 = evaluateNode( branch->list_next->list_next );

        retval = eval_cb_lists_same( list1, list2 );

        leWipe( list1 );
        leWipe( list2 );

        return( leNew ( (retval == 1) ? "T" : "NIL" ) );
    }

```

7.3.26 if

The standard conditional. (if (conditional) (then-block) (else-block))

if will evaluate the first parameter. If it evaluates to the T atom, then if will then evaluate the following then block. If it was non-T, then it will evaluate the else block.

63a \langle Eval cb if proto 63a $\rangle \equiv$ (63b 83)
`le * eval_cb_if(const int argc, le * branch)`

63b \langle Eval cb if implementation 63b $\rangle \equiv$ (81)
 \langle Eval cb if proto 63a \rangle

```

{
    le * retcond = NULL;
    le * retblock = NULL;

    if (!branch || argc < 3 || argc > 4) return( leNew( "NIL" ) );

    /* if */
    retcond = evaluateNode(branch->list_next);

    if (!strcmp ( retcond->data, "NIL" ))
    {
        /* else */
        if (argc == 3) /* no else */
            return( retcond );

        leWipe( retcond );
        return( evaluateNode( branch->list_next->list_next->list_next ) );
    }

    /* then */
    leWipe( retcond );
    return( evaluateNode(branch->list_next->list_next) );
}

```

7.3.27 when and unless helper

This will evaluate the first parameter.

If it evaluates to NIL and the which parameter is set to WU_UNLESS, OR it evaluates to non-NIL the which parameter is set to WU_WHEN then the body blocks get evaluated, otherwise a NIL atom is returned. The final body block to get evaluated has its value returned.

63c \langle Eval cb when unless helper enum 63c $\rangle \equiv$ (83)
`enum whenunless { WU_WHEN, WU_UNLESS };`

```

64a  <Eval cb when unless proto 64a>≡ (64b 83)
      le *
        eval_cb_whenunless_helper(
          enum whenunless which,
          const int argc,
          le * branch
        )

```

```

64b  <Eval cb when unless implementation 64b>≡ (81)
      <Eval cb when unless proto 64a>
      {
        le * retval = NULL;
        le * retblock = NULL;
        le * trythis = NULL;

        if (!branch || argc < 3 ) return( leNew( "NIL" ) );

        /* conditional */
        retval = evaluateNode(branch->list_next);

        if ( which == WU_UNLESS )
        {
          /* unless - it wasn't true... bail */
          if ( strcmp( retval->data, "NIL" ) )
          {
            leWipe( retval );
            return( leNew( "NIL" ) );
          }
        } else {
          /* when: it wasn't false... bail */
          if ( !strcmp( retval->data, "NIL" ) )
          {
            return( retval );
          }
        }

        trythis = branch->list_next->list_next;
        while( trythis )
        {
          if ( retval ) leWipe( retval );

          retval = evaluateNode(trythis);
          trythis = trythis->list_next;
        }
        return( retval );
      }

```

7.3.28 unless

(unless (conditional) (block) (block) ...)

unless will evaluate the first parameter. If it evaluates to NIL then the body blocks get evaluated, otherwise a NIL atom is returned. The final body block to get evaluated has its value returned.

65a \langle Eval cb unless proto 65a $\rangle \equiv$ (65b 83)
`le * eval_cb_unless(const int argc, le * branch)`

65b \langle Eval cb unless implementation 65b $\rangle \equiv$ (81)
 \langle Eval cb unless proto 65a \rangle
`{`
`return(eval_cb_whenunless_helper(`
`WU_UNLESS,`
`argc,`
`branch));`
`}`

7.3.29 when

(when (conditional) (block) (block) ...)

when evaluates the first parameter. If it returns a non-NIL value, then the remaining body blocks get evaluated, otherwise it will return a NIL atom. The last conditional's value gets returned otherwise.

This is basically the same as **unless** but with a reversed conditional. This might be integrated with it into a helper function eventually.

65c \langle Eval cb when proto 65c $\rangle \equiv$ (65d 83)
`le * eval_cb_when(const int argc, le * branch)`

65d \langle Eval cb when implementation 65d $\rangle \equiv$ (81)
 \langle Eval cb when proto 65c \rangle
`{`
`return(eval_cb_whenunless_helper(`
`WU_WHEN,`
`argc,`
`branch));`
`}`

7.3.30 cond

```
(cond ((conditional) (block) ... ) ((conditional) (block) ... ) ((conditional)
(block) ... ) )
```

`cond` checks each conditional in turn. The first one returning non-NIL gets its body blocks evaluated. The last block related to that conditional gets its value returned.

If the succeeding conditional has no block, the return value of the conditional is returned.

If no conditional satisfies it, a NIL atom will be returned.

```
66 <Eval cb cond proto 66>≡ (67 83)
    le * eval_cb_cond( const int argc, le * branch )
```

```

67  <Eval cb cond implementation 67>≡ (81)
    <Eval cb cond proto 66>
    {
        le * retval = NULL;
        le * retblock = NULL;
        le * trythis = NULL;
        le * tryblock = NULL;
        int newargc;

        if (!branch || argc < 2 ) return( leNew( "NIL" ));

        trythis = branch->list_next;
        while (trythis)
        {
            newargc = countNodes( trythis->branch );
            if (newargc == 0) continue;

            /* conditional */
            if (retval) leWipe(retval);
            retval = evaluateNode(trythis->branch);

            if ( strcmp(retval->data, "NIL" ))
            {
                if (newargc == 1)
                {
                    return( retval );
                }

                tryblock = trythis->branch->list_next;
                while (tryblock)
                {
                    if (retblock) leWipe(retblock);
                    retblock = NULL;

                    retblock = evaluateNode(tryblock);
                    tryblock = tryblock->list_next;
                }
                return( retblock );
            }

            trythis = trythis->list_next;
        }
        return( retval );
    }

```

7.3.31 princ

Simply print to standard output the parameters passed in.

68a \langle *Eval cb princ proto 68a* $\rangle \equiv$ (68b 83)
`le * eval_cb_princ(const int argc, le * branch)`

68b \langle *Eval cb princ implementation 68b* $\rangle \equiv$ (81)
 \langle *Eval cb princ proto 68a* \rangle
`{`
`le * thisnode;`
`le * retblock = NULL;`
`if (!branch || argc < 1) return(leNew("NIL"));`

`thisnode = branch->list_next;`
`while (thisnode)`
`{`
`if (retblock) leWipe(retblock);`
`retblock = evaluateNode(thisnode);`
`leDumpReformat(stdout, retblock);`

`thisnode = thisnode->list_next;`
`}`
`return(retblock);`
`}`

7.3.32 terpri

Simply print to standard output a newline character.

68c \langle *Eval cb terpri proto 68c* $\rangle \equiv$ (68d 83)
`le * eval_cb_terpri(const int argc, le * branch)`

68d \langle *Eval cb terpri implementation 68d* $\rangle \equiv$ (81)
 \langle *Eval cb terpri proto 68c* \rangle
`{`
`le * thisnode;`
`le * retblock = NULL;`
`if (!branch || argc != 1) return(leNew("NIL"));`

`printf("\n");`
`return(leNew("NIL"));`
`}`

7.3.33 eval

`eval` evaluates its parameter, and returns the result from the evaluation. It must basically do a double evaluation due to the way things are stored internally.

69a $\langle \textit{Eval cb eval proto 69a} \rangle \equiv$ (69b 83)
`le * eval_cb_eval(const int argc, le * branch)`

69b $\langle \textit{Eval cb eval implementation 69b} \rangle \equiv$ (81)
 $\langle \textit{Eval cb eval proto 69a} \rangle$

```

{
    le * temp;
    le * retval;
    if (!branch || argc != 2 ) return( leNew( "NIL" ) );

    temp = evaluateNode(branch->list_next);
    retval = evaluateBranch(temp);
    leWipe( temp );
    return( retval );
}

```

7.3.34 proghelper

Since the functions `prog1`, `prog2` and `progn` are nearly identical in nature, we will use the following function to do most of their work. The only difference is that there is an extra parameter, `returnit`. If `returnit` is 1, then the first code block's result gets returned. If `returnit` is 2, then the second code block's result gets returned. If `returnit` is negative, then the last code block's result gets returned.

Basically, we will evaluate all body blocks passed in, and return the appropriate return list.

69c $\langle \textit{Eval cb prog proto 69c} \rangle \equiv$ (70a 83)
`le * eval_cb_prog(const int argc, le * branch, int returnit)`

70a $\langle \text{Eval cb prog implementation 70a} \rangle \equiv$ (81)
 $\langle \text{Eval cb prog proto 69c} \rangle$

```

{
    le * curr;
    le * retval = NULL;
    le * tempval = NULL;
    int current = 0;
    if (!branch || argc < (returnit +1) ) return( leNew( "NIL" ) );

    curr = branch->list_next;
    while (curr)
    {
        ++current;

        if ( tempval ) leWipe (tempval);
        tempval = evaluateNode( curr );

        if (current == returnit)
            retval = leDup( tempval );

        curr = curr->list_next;
    }

    if (!retval) retval = tempval;

    return( retval );
}

```

7.3.35 prog1

Using the above helper, we want to evaluate all code blocks, returning the return value from the first block.

70b $\langle \text{Eval cb prog1 proto 70b} \rangle \equiv$ (70c 83)
 $\text{le * eval_cb_prog1(const int argc, le * branch)}$

70c $\langle \text{Eval cb prog1 implementation 70c} \rangle \equiv$ (81)
 $\langle \text{Eval cb prog1 proto 70b} \rangle$

```

{
    return( eval_cb_prog( argc, branch, 1 ) );
}

```

7.3.36 prog2

Using the above helper, we want to evaluate all code blocks, returning the return value from the second block.

71a $\langle \text{Eval cb prog2 proto 71a} \rangle \equiv$ (71b 83)
`le * eval_cb_prog2(const int argc, le * branch)`

71b $\langle \text{Eval cb prog2 implementation 71b} \rangle \equiv$ (81)
 $\langle \text{Eval cb prog2 proto 71a} \rangle$
`{`
`return(eval_cb_prog(argc, branch, 2));`
`}`

7.3.37 progn

Using the above helper, we want to evaluate all code blocks, returning the return value from the final block.

71c $\langle \text{Eval cb progn proto 71c} \rangle \equiv$ (71d 83)
`le * eval_cb_progn(const int argc, le * branch)`

71d $\langle \text{Eval cb progn implementation 71d} \rangle \equiv$ (81)
 $\langle \text{Eval cb progn proto 71c} \rangle$
`{`
`return(eval_cb_prog(argc, branch, -1));`
`}`

7.3.38 defun

This stores away a function to be used later. The format for this is: (defun funcname (parameters) (bodyblock))

None of this gets evaluated when it is called here. It will get evaluated later on if they get called. We will just store it aside for now.

The funcname is returned as an atom.

The parameters for this function are not the same as variables in the system. They are local variables of local scope, and will override any global variables when the function is called later.

71e $\langle \text{Eval cb defun proto 71e} \rangle \equiv$ (72a 83)
`le * eval_cb_defun(const int argc, le * branch)`

72a \langle *Eval cb defun implementation 72a* $\rangle \equiv$ (81)
 \langle *Eval cb defun proto 71e* \rangle
 {
 le * thisnode;
 le * retblock = NULL;
 if (!branch || argc < 4) return(leNew("NIL"));

 if (!branch->list_next->data) return(leNew("NIL"));

 defunList = variableSet(
 defunList,
 branch->list_next->data,
 branch->list_next->list_next
);

 return(leNew(branch->list_next->data));
 }

7.4 Utility methods

We need a few utility functions to help us do all of the above work. Those utilities follow. They are `countNodes`, `evalCastLeToInt` and `evalCastIntToLe`.

7.4.1 `countNodes()`

This simply takes in a branch and returns the number of `le` nodes along its primary list. (Through the `list next` pointer). Empty lists will return 0. This is just a simple iterator, traversing the list.

72b \langle *Eval utility counter proto 72b* $\rangle \equiv$ (72c 83)
`int countNodes(le * branch)`

72c \langle *Eval utility counter implementation 72c* $\rangle \equiv$ (81)
 \langle *Eval utility counter proto 72b* \rangle
 {
 int count = 0;

 while (branch)
 {
 count++;
 branch = branch->list_next;
 }
 return(count);
 }

7.4.2 cast LE To Int

This is a simple 'cast' function which takes in a `le` atom, converts its data to an integer using the standard `atoi()` call. That value is then returned.

73a \langle *Eval cast le to int proto 73a* $\rangle \equiv$ (73b 83)
`int evalCastLeToInt(const le * levalue)`

73b \langle *Eval cast le to int implementation 73b* $\rangle \equiv$ (81)
 \langle *Eval cast le to int proto 73a* \rangle
`{`
`if (!levalue) return(0);`
`if (!levalue->data) return(0);`
`return(atoi(levalue->data));`
`}`

7.4.3 cast Int To LE

This is a simple 'cast' function which takes in an integer, and then builds a new `le` atom using that value as its data. That new atom is then returned.

73c \langle *Eval cast int to le proto 73c* $\rangle \equiv$ (73d 83)
`le * evalCastIntToLe(int intvalue)`

73d \langle *Eval cast int to le implementation 73d* $\rangle \equiv$ (81)
 \langle *Eval cast int to le proto 73c* \rangle
`{`
`char buffer[80];`
`sprintf (buffer, "%d", intvalue);`
`return(leNew(buffer));`
`}`

7.5 Evaluator Valves

These two functions are the brains behind the interpreter. They will traverse lisp trees, and recursively evaluate each part of them. They look up values in the variable list where necessary.

7.5.1 evaluateBranch

Evaluate branch will for example evaluate all of (+ 3 A).

It first looks to see if the current entry is a list. If it is, it will evaluate it to determine the keyword to use. It will then use this keyword to look up a callback in the `evalTable`. If it was found, it will simply return what the callback returns. If no function had been called, it will try to evaluate the node using the `evaluateNode` function below.

```
74 <Eval evaluateBranch proto 74>≡ (75 83)
    le * evaluateBranch(le * trybranch)
```

```

75  <Eval evaluateBranch implementation 75>≡ (81)
    <Eval evaluateBranch proto 74>
    {
        le * keyword;
        int tryit = 0;
        if (!trybranch) return( NULL );

        if (trybranch->branch)
        {
            keyword = evaluateBranch(trybranch->branch);
        }
        else
            keyword = leNew( trybranch->data );

        if (!keyword->data)
        {
            leWipe( keyword );
            return( leNew( "NIL" ) );
        }

        for ( tryit=0 ; evalTable[tryit].word ; tryit++)
        {
            if (!strcmp(evalTable[tryit].word, keyword->data))
            {
                leWipe( keyword );
                return( evalTable[tryit].callback(
                    countNodes( trybranch ),
                    trybranch)
                );
            }
        }

        leWipe( keyword );
        return( evaluateNode( trybranch ) );
    }

```

7.5.2 evaluateNode

EvaluateNode will for example evaluate just the A of (+ 3 A).

It will look to see if the node has a branch. If it does, it will evaluate it using the above function. If it had been quoted, then that branch will just get returned.

If it has no branch, then it will attempt to retrieve the variable with the name specified in the data. If that was unsuccessful, it will just return the data itself as an atom.

76 *<Eval evaluateNode proto 76>*≡ (77 83)
 le * evaluateNode(le * node)

```

77  <Eval evaluateNode implementation 77>≡ (81)
    <Eval evaluateNode proto 76>
    {
        le * temp;
        le * value;

        if (node->branch)
        {
            if( node->quoted )
            {
                value = leDup( node->branch );
            } else {
                value = evaluateBranch( node->branch );
            }
        } else {
            temp = variableGet( defunList, node->data );

            if (temp)
            {
                value = evaluateDefun( temp, node->list_next );
            } else {
                temp = variableGet( mainVarList, node->data );

                if (temp)
                {
                    value = leDup( temp );
                } else {
                    value = leNew( node->data );
                }
            }
        }

        return( value );
    }

```

7.5.3 evaluateDefun

This is a tricky one. It gets called when a predefined function, set up with the “defun” command earlier, gets called during interpretation time.

This could be done any number of ways, using a stack based system or the like for variable lists, but instead I decided to go with a macro-like pre-processing design.

First both lists passed in, the function definition `fcn` as well as the list of new parameters `params` are both non-NULL, and contain an equal number of parameters. If this is not the case, we instantly bail out and return a NIL atom.

Next, we `leDup` the function, `fcn` into a new structure, `function`.

Now, we make two passes over this new structure. First we go through and tag each variable in `function` as defined with its variable list. The first parameter’s occurrence in the structure gets tagged with “1”, the second with “2” and so on.

The second pass replaces the items tagged as “1” with the first parameter in the `params` list, “2” with the second in the `params` list, and so on.

Then all we need to do is to evaluate the resulting `function` list, which now has all of its local variables replaced with the passed in parameters, and return that value.

We do two passes to make sure that any variables in the parameter list get replaced appropriately, in case their names clash with the global variable list entries.

78 $\langle \text{Eval evaluateDefun proto 78} \rangle \equiv$ (79 83)
 `le * evaluateDefun(le * fcn, le * params)`

```

79  <Eval evaluateDefun implementation 79>≡ (81)
    <Eval evaluateDefun proto 78>
    {
        le * function;
        le * thisparam;
        le * result;
        int count;

        /* make sure both lists exist */
        if (!fcfn || !params ) return( leNew( "NIL" ));

        /* check for the correct number of parameters */
        if (countNodes(fcn->branch) != countNodes(params))
            return( leNew( "NIL" ));

        /* allocate another function definition, since we're gonna hack it */
        function = leDup(fcn);

        /* pass 1: tag each node properly.
           for each parameter: (fcfn)
           - look for it in the tree, tag those with the value
        */
        count = 0;
        thisparam = fcn->branch;
        while (thisparam)
        {
            leTagData(function, thisparam->data, count);
            thisparam = thisparam->list_next;
            count++;
        }

        /* pass 2: replace
           for each parameter: (param)
           - evaluate the passed in value
           - replace it in the tree
        */
        count = 0;
        thisparam = params;
        while (thisparam)
        {
            result = evaluateNode( thisparam );
            leTagReplace(function, count, result);
            thisparam = thisparam->list_next;
            leWipe(result);
            count++;
        }
    }

```

```
/* then evaluate the resulting tree */
result = evaluateBranch( function->list_next );

/* free any space allocated */
leWipe( function );

/* return the evaluation */
return( result );
}
```

7.6 eval.c

Here we build up all of the above blocks into the .c file.

```

81  <eval.c 81>≡
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "eval.h"
    #include "vars.h"

    <Eval lookup table 37>

    <Eval evaluateBranch implementation 75>
    <Eval evaluateNode implementation 77>
    <Eval evaluateDefun implementation 79>

    <Eval utility counter implementation 72c>

    <Eval cast le to int implementation 73b>
    <Eval cast int to le implementation 73d>

    <Eval cb cume helper implementation 42>
    <Eval cb add implementation 43b>
    <Eval cb subtract implementation 44a>
    <Eval cb multiply implementation 45a>
    <Eval cb divide implementation 46a>
    <Eval cb oneplus implementation 46c>
    <Eval cb oneminus implementation 47b>
    <Eval cb modulus implementation 48a>

    <Eval cb lt implementation 48c>
    <Eval cb lt eq implementation 49b>
    <Eval cb gt implementation 50a>
    <Eval cb gt eq implementation 50c>
    <Eval cb eqsign implementation 51b>

    <Eval cb and implementation 52a>
    <Eval cb or implementation 53a>
    <Eval cb not implementation 54a>

    <Eval cb atom implementation 55a>
    <Eval cb car implementation 56a>
    <Eval cb cdr implementation 57a>
    <Eval cb cons implementation 58a>
    <Eval cb list implementation 59>
    <Eval cb equal helper implementation 61>

```

<Eval cb equal implementation 62b>

<Eval cb if implementation 63b>

<Eval cb when unless implementation 64b>

<Eval cb unless implementation 65b>

<Eval cb when implementation 65d>

<Eval cb cond implementation 67>

<Eval cb princ implementation 68b>

<Eval cb terpri implementation 68d>

<Eval cb eval implementation 69b>

<Eval cb prog implementation 70a>

<Eval cb prog1 implementation 70c>

<Eval cb prog2 implementation 71b>

<Eval cb progn implementation 71d>

<Eval cb set implementation 39>

<Eval cb setq implementation 40b>

<Eval cb defun implementation 72a>

7.7 eval.h

And we need to do the same for the header file as well.

```
83 <eval.h 83>≡
    #include "lists.h"

    <Eval callback typedef 35>
    <Eval lookup struct 36>

    <Eval evaluateBranch proto 74>;
    <Eval evaluateNode proto 76>;
    <Eval evaluateDefun proto 78>;

    <Eval utility counter proto 72b>;

    <Eval cast le to int proto 73a>;
    <Eval cast int to le proto 73c>;

    <Eval cb cume helper enum 41a>
    <Eval cb cume helper proto 41b>;
    <Eval cb add proto 43a>;
    <Eval cb subtract proto 43c>;
    <Eval cb multiply proto 44b>;
    <Eval cb divide proto 45b>;
    <Eval cb oneplus proto 46b>;
    <Eval cb oneminus proto 47a>;
    <Eval cb modulus proto 47c>;

    <Eval cb lt proto 48b>;
    <Eval cb lt eq proto 49a>;
    <Eval cb gt proto 49c>;
    <Eval cb gt eq proto 50b>;
    <Eval cb eqsign proto 51a>;

    <Eval cb and proto 51c>;
    <Eval cb or proto 52b>;
    <Eval cb not proto 53b>;

    <Eval cb atom proto 54b>;
    <Eval cb car proto 55b>;
    <Eval cb cdr proto 56b>;
    <Eval cb cons proto 57b>;
    <Eval cb list proto 58b>;
    <Eval cb equal helper proto 60>;
    <Eval cb equal proto 62a>;
```

<Eval cb if proto 63a>;
<Eval cb when unless helper enum 63c>
<Eval cb when unless proto 64a>;
<Eval cb unless proto 65a>;
<Eval cb when proto 65c>;
<Eval cb cond proto 66>;

<Eval cb princ proto 68a>;
<Eval cb terpri proto 68c>;

<Eval cb eval proto 69a>;
<Eval cb prog proto 69c>;
<Eval cb prog1 proto 70b>;
<Eval cb prog2 proto 71a>;
<Eval cb progn proto 71c>;

<Eval cb set proto 38>;
<Eval cb setq proto 40a>;

<Eval cb defun proto 71e>;

8 Sample files

This is a bunch of sample files to test the system with.

8.1 Sample 01

85a `<sample01.lsp 85a>≡
; sample file.`

8.2 Sample 02

85b `<sample02.lsp 85b>≡
;; this is a test
; this should be a comment
; (+ 4 5)

; first some simple math functions
(+ 3 2)
(- 4 5)
(* 4 9)
(/ 100 20)

; some nests and paren testing.
(+ (- 4 5) (+ 3 4) 10)
(* (+ 3 4 (- 10 7) 9) 17)
(* (+ 3 4 (- 10 7) 9) 17)
(* (+ 3 4(- 10 7)9)17)
(- 4)`

8.3 Sample 03

```

86  <sample03.lsp 86>≡
    ; this is a bunch of supported stuff.  it's good regression testing material...
    ; uncomment the part you want to try...
    ; (+ 4 5)

    ; some simple math functions
    (+ 4 3 2)
    (- 10 4)
    (- 3)
    (- (- 1 10) )
    (* 4 9)
    (/ 100 20)

    ;(1+ 30)
    ;(1- 30)

    ;(1+ (+ 20 30))
    ;(1- (+ 20 30))
    ;(% 2001 4)
    ;(% 4 2001)

    ;(4)
    ;(-4)
    ;4

    ;(and (< 100 (setq a 1)) (> 300 (setq a 2)) (setq a 3) )
    ;(a)
    ;(and (> 100 (setq a 1)) (> 300 (setq a 2)) (setq a 3) )
    ;(a)
    ;(and (> 100 (setq a 1)) (> 300 (setq a 2)) (setq a 400) )
    ;(a)
    ;(and (> 100 (setq a 1)) (> 300 (setq a 2)) )
    ;(a)

    (or (< 100 (setq a 1)) (< 300 (setq a 2)) NIL)
    (a)

    ;(< 100 5)
    ;(< 5 100)
    ;(< 5 5)

    ;(<= 100 5)
    ;(<= 5 100)
    ;(<= 5 5)

```

```
;> 100 5)
;> 5 100)
;> 5 5)

;>= 100 5)
;>= 5 100)
;>= 5 5)

;= 100 5)
;= 5 100)
;= 5 5)

;(not (= 100 5))
;(not (= 5 100))
;(not (= 5 5))

;(setq hello '(0 1 1 3 4))
;(hello)
;(atom hello)

;(atom (setq foo (+ 2 3)))
;(atom (setq bar '(+ 2 3)))
;(atom 'a)
;(atom 8)
;(atom '(a b c))

;(setq hi quote (H E L L O))
;(setq hi2 '(H E L L O))

;(setq floop goo)
(setq cheese "cheese is quite yummy")
;(setq f 34)
;(setq g '(+ 4 3))
;(setq q 3)
;(setq w '3)
;(setq e '(3)) ; these don't currently get handled properly in eval
;(setq r '((3))) ; these don't currently get handled properly in eval
;(setq t '(())) ; these don't currently get handled properly in eval
;(setq g (+ 9 2))
;(setq foo '(+ (- 3 4 5) (* 2 3 4)))
;(setq bar (+ (- 3 4 5) (* 2 3 4)))
;(+ 2 (setq p (+ 5 3)))

;(+ f g p)
```

```
;( + 2 (setq x (* 3 4)))
;( + 0 x)

(setq x 5)
(setq y (1+(+ 0 x)))
(x) (y)

;( + 4 '(+ 3 4 '5 6 7))
;( + 4 quote (+ 3 4 '5 6 7))

;(setq mud "dirt" smog "smoke")

;(car '(a b c))
;(cdr '(a b c))
;(setq x '(a b c))
;(car x)
;(cdr x)

;(cdr '())
;(car '())
;(cdr '(a))
;(car '(a))

;(car '((a b)))
;(car (car '((a b))))
;(cdr '((a b) (c d)))

;(cdr '(a b))
;(car (cdr '(a b c d e)))
;(car '( (a b c) (d e f) (g h i) ))
;(cdr '( (a b c) (d e f) (g h i) ))

;(car (cdr '((a b) (c d) )))
;(cdr (car '((a b) (c d) )))

;(atom (cdr '((a b) (c d) )))
;
;(cdr (car '((a b) (c d))))
;
;(cdr (car '((a b c) (d e f))))
;(car (car '((a b c) (d e f))))
;(car (cdr '((a b c) (d e f))))
;(car (car (cdr '((a b c) (d e f)))))
(cdr (car (cdr '((a b c) (d e f)))))
```

```

(car (cdr (car (cdr '((a b c) (d e f))))))

;(cons 'a '(b c))
;(setq x (cons 'a '(b c)))
;(car x)
;(cdr x)
;(cons 'a '(b))
;(cons '(a b) '(c d))
;(cons 'a (cons 'b '(c d)))

;(setq x '(a b))
;(cons (car x) (cons (car (cdr x)) '(c d)) )

;(setq x 'a)
;(setq y '(b c))
;(cons x y)
;(x)
;(y)
;(car (setq x '(a b c)))
;(car '(setq x '(a b c)))

(list 'a 'b 'c)
(list 'a '(b c) 'd)
(list 'a 'b 'c 'd)
;(list 'a)
;(list)

(setq result (if (< 3 4) (setq a '(t r o o)) (setq b '(f a l e s))) )
(a)(b)(result)
(setq result (if (> 3 4) (setq c '(t r o o)) (setq d '(f a l e s))) )
(c)(d)(result)

;(setq result (if (< 3 4) (setq a '(t r o o)) ) )
;(a)(b)(result)
;(setq result (if (> 3 4) (setq c '(t r o o)) ) )
;(c)(d)(result)

;(unless (< 3 4) (setq x (+ 3 4)) (setq y '(+ 9 8)) )
;(unless (> 3 4) (setq w (+ 3 4)) (setq z '(+ 9 8)) )
;
;(when (< 3 4) (setq a (+ 3 4)) (setq d '(+ 9 8)) )
;(when (> 3 4) (setq s (+ 3 4)) (setq f '(+ 9 8)) )

;(cond ( (> 3 4) (setq a 'one1) (setq b 'one2) )
;       ( (= 3 4) (setq c 'two1) (setq d 'two2) )
;       ( (< 3 4) (setq e 'three1) (setq f 'three2) )

```

```
;)
;
;(cond ( (> 3 4) (setq g 'one1) (setq h 'one2) )
;      ( (= 3 4) (setq i 'two1) (setq j 'two2) )
;      ( (< 3 4) )
;)

;(princ "hello world")
;(terpri)
;(princ "this is also a test");

;(setq a 'b)
;(setq b 'c)
;(a)
;(b)
;(eval a)
;
;(eval (cons '+ '(2 3)))

;(prog1 (car '(a b c)) (cdr '(a b c)) (cdr '(d e f)) (cdr '(g h i)))
;(prog2 (car '(a b c)) (cdr '(a b c)) (cdr '(d e f)) (cdr '(g h i)))
;(progn (car '(a b c)) (cdr '(a b c)) (cdr '(d e f)) (cdr '(g h i)))

;(setq hello '(0 1 1 3 4))
;(hello)
;(atom hello)

;(setq a '(f 0 o))
;(setq b (+ 3 4))
;(setq c (+ 3 4) d (1+ b) e '(4 5 6))
;(setq f (+ 3 4) g (1+ b) h)

;(setq g '(a b c d))
;(set 'b '(a b c d))
;
;(set (car g) (cdr g))

;(setq x '(a b c))
;(setq y (cdr x))
;(setq z '(b c))
;
;(equal (cdr x) y)
;(equal y z)
;(equal z z)
;(equal x z)
;(equal x '(a b c))
```

```
(defun addthree (x) (+ x 3))
;
;(defun addtwoto3 (x y) (+ x y 3))
;
;(addtwoto3 2 3)
;(addtwoto3 2 3 4)
;
(addthree 3)
;(addthree 3 5 7)
;
;(addthree (* 4 (1- 7)))
;
;(defun average (x y) (/ (+ x y) 2))
;(average 7 (car '(21 3 4 5)))
;(average 9 31)
;
;(setq foo '(7 21 34 22 99))
;(car foo) (car (cdr foo))
;(/ (+ (car foo) (car (cdr foo))) 2)
;
;(defun averagel (x) (/ (+ (car x) (car (cdr x))) 2))
;(averagel '(7 21 34 22 99))
;(averagel '(9 31 34 22 99))
;
(defun three (x) (+ 2 1))
(three 4)
```

9 Version information

- 92a \langle *version build version 92a* $\rangle \equiv$ (94)
"0.5"
- 92b \langle *version build date 92b* $\rangle \equiv$ (94)
"2001-10-10"
- 92c \langle *version build tpotc 92c* $\rangle \equiv$ (94)
"Storm"

2001-10-10 "Storm" 0.5

- Changed the parser to use a pair of callbacks. (more flexible)

2001-10-10 "Trebuchet" 0.4

- new: defun
- List tagging functions (list.nw) added for the above

2001-10-09 "Collapsing" 0.3

- new: set, setf, setq (new version)
- new: equal
- old: quote

2001-10-08 "Uakti" 0.2

- Philip Glass album, and Brazillian band
- added 'quote' parsing
- converted over to full list passing (setq hello '(0 1 1 3 4)) works
- fixed the evaluation of a variable name as a return value
- new: and, or, not, null, atom, car, cdr, cons, list
- new: if, unless, when, cond
- new: eval, prog1, prog2, progn
- new: printc, terpri

2001-10-07 "In The Window"

- basics of setq done. integrated and debugged the variable mechanism
- it does not yet do things like: (setq hello '(0 1 1 3 4))

2001-10-03 "Building A Wall" 0.1

- Orbital - "Illuminate"
- initial file parser
- initial tree builder
- variable list tools
- beginnings of the evaluator
- add, subtract, multiply, divide added

9.1 The Source File `version.h`

94 `<version.h 94>`≡

```
#define VER_BUILD_DATE <version build date 92b>
#define VER_BUILD_VER  <version build version 92a>
#define VER_BUILD_TPOTC <version build tpotc 92c>
```