LL v2.0

A Label Language Compiler

DISCLAIMER

THIS SOFTWARE IS PROVIDED "AS IS" AND NO WARRANTIES ARE OFFERED NOR WILL ANY LIABILITY BE ASSUMED DUE TO ITS USE. THIS SOFTWARE HAS BEEN THOROUGHLY TESTED OVER MONTHS AND HAS BEEN FOUND TO BE DEFECT-FREE.

THIS SOFTWARE IS FREE OF CHARGE "FREEWARE." NO MONEY SHALL BE CHARGED FOR THIS PROGRAM EXCEPT DISTRIBUTION/COPYING COSTS WHICH HAVE NOT TO EXCEED $5 US. THE LL PACKAGE MAY BE DISTRIBUTED IN ANY FORM, ELECTRONICALLY OR OTHERWISE AS LONG AS THE FULL CONTENTS REMAIN IN THE ARCHIVE.

THE SOURCE CODE IS NOT INCLUDED BUT MAY BE OBTAINED FROM THE AUTHOR.

Kevin A. McHugh
4671 Mebane Rogers Road.
Mebane, NC 27302
U.S.A.

Internet:	mchugh@hamlet.uncg.edu
FidoNet:	sysop@1:3644/13.0
BBS:	(919) 563-0183

Introduction

What is LL? LL is a compiler that takes your source code and produces formatted output based on instructions in the code. I wrote this program because every month I distribute postcards for the local Amiga users group and had not found a flexible utility for making both the addresses on the front and the text on the back of the card.

I had wanted to use my compiler skills for a while so they wouldn't get rusty and this is why I chose a Shell based interface over the standard GUI of the Amiga. Due to the fact this is a Shell interface it can easily be ported to any platform that will compile ANSI-C.

LL reads source files that are written in a language I call "L." It reads instructions and blocks of data and then produces a final result based on your code. L is a very simple language and even if you don't program it will not be hard for you to use.

Before you use the program please read and acknowledge the disclaimer on the previous page. For program support and information I can be reached at the resources shown above. Internet mail is preferred.

The Language "L"

L is a very simple and easy language. It's format is modelled after the C language. Unlike a true language it doesn't offer input/output or user interaction. It is more closely related to other text compilers such as the Unix "troff" etc..

The language is defined as follows:

1. *Variables*
2. *Printer Commands*
3. *The Data*
4. *Data Functions*
5. *Printer Commands*
6. *Functions*

Comments can be used anywhere throughout the source file. L accepts the standard C comments and the newer C++ comments. A C comment is a block of text that starts with /* ends with */ and has anything in-between. The newer C++ comments are started with // and terminate at the end of the same line. **Note: You cannot nest comments.**

```
/*
** This Is A C Comment
*/

//  This Is A C++ Comment  ;-)
```

L is a case-insensitive language, basically this means that if L sees the words PRINT, print, Print, PrInT or any combination it interprets them all as the print command. Normally we type everything in lower or UPPER case for comprehension. Any data enclosed in a data block **is case sensitive** so you control exactly what is being output.

Generally speaking, all white space is ignored unless relevant. This means that you can format your L code however makes you feel comfortable, the program is smart enough to know when to skip white space and when to keep it. By definition, white space is any character which you cannot regularly see. (carriage returns, linefeeds, spaces, tabs etc...)

1. Variables

L has pre-defined variables for formatting your text. You can use them however you like and in whatever order you like as long as they precede what you want to print. All variables are entered in the following format:

#define *variable_name* **=** *value* **;**

Where *variable_name* is one of the pre-defined variables and *value* is a valid value. The equal sign "=" is optional but the compiler will produce a warning if it is omitted.

The variables are as follows:

device
height
width
space
indent
header
footer
postage
eol

DEVICE

Example: **#define device = prt:;**
Default: **l.out**

The device variable tells the compiler where to send the output it generates. Output can go to a text file or a device such as a printer. If you choose the device to be **prt:** (the standard AmigaDOS printer device) then all printer information from Preferences will be used. If you will be using the Postnet bar code feature of LL then in order for your output to come out correctly you will need to select the **PAR:** device as your final output destination. Postnet will be discussed later in the manual.

Generally it is best to send output to a temporary file until you have everything correct for final printing. This can be done by specifying a filename instead of a device like follows:

```
#define device = test.output;
#define device = "address list";
```

Note that you need to quote the filename if it contains spaces. If you enter multiple device entries then only the last one will be used.

HEIGHT

Example: **#define height = 24;**
Default: **10**

The height variable sets the height of the output form so that correct paper handling will be produced. The height is in printer lines. If you know the height in inches you can easily derive the height in lines:

lines = height_in_inches * lines_per_inch

Your printer manual and Workbench preferences will show you how many lines per inch your printer is set for. Note that height must be an integer and cannot contain a floating point value.

WIDTH

Example: **#define width = 60;**
Default: **40**

The width variable specifies the width of the output form. The width is measured in characters. A standard letter sized 8.5" x 11" page prints 80 characters across so you can estimate approximately 10 characters per inch (CPI). Your CPI rate will be found in your printers manual and information on how to change it will also be available. Note that width must be an integer.

SPACE

Example: **#define space = 10;**
Default: **2**

The space variable indicates the number of lines to feed the paper before printing should commence. Like the height, space is measured in printer lines. For obvious reasons, space should be less than the page height. Note space must also be an integer.

INDENT

Example: **#define indent = 7;**
Default: **5**

The indent variable is the number of characters to move the printhead in before printing the data. The size of a character can be calculated by refering to your printers current CPI setting.

HEADER

Example: **#define header = "This Is A Header";**
Default: **None**

The header variable is a string of characters that will be printed before each data block. This is handy for printing dates and times etc... The header may be enclosed in quotes but it is not absolutely required but highly recommended.

Headers can also use imbedded codes to produce run-time output for example the date or the time. The imbedded codes are similar to the codes used in the C language and are listed below:

Escape Codes:

**\ **	**Backslash**	****
\"	**Quote Character**	**"**
\ d	**Date**	**01/03/94**

\ e	**Escape**	**ASCII 27**
\ i	**Indent** [1]	**Indent's Spaces**
\ n	**Newline**	**ASCII 14**
\ r	**Carrage Return**	**ASCII 13**
\ t	**Time**	**18:06:51**
\ x	**Hex Time** [2]	**2d2c0347**

[1] The indent code will indent the following text by the number of spaces defined by the *indent* variable.

[2] The Hex Time code represents the number of seconds passed since January 1, 1970 to now. It is expressed in hexadecimal notation.

Here are some examples of text using embedded codes followed by the actual output after compilation.

#define header = "Mailed: \d\r\nAt: \t";

Mailed: 02/11/94
At: 19:25:01

#define header = "Code ***\x***";

Code ***2d2c3561***

The possibilities are endless and can add extra style to your mailings without having to do extra work. Note if no header is declared then none will be printed.

FOOTER

Example: **#define footer = "This Is A Footer";**
Default: **None**

The footer serves the exact same purpose of the header except that it is printed proceding every data block instead of before. The footer variable also excepts a string of characters and can contain embedded escape codes.

#define footer = "-----------------------------------";

The above will simply print a dotted line after each data block. Note if no footer is declared then none will be printed.

POSTAGE

Example: **#define postage = 0.19;**
Default: **0.00**

The postage variable is only used if you choose to have LL make a summary of your mailings at the end of each session. If you use the report() function in your program then the total cost of the mailing session will be computed using the above number as the cost for mailing one data block. The value entered is in dollars and cents and can be integer or floating point.

```
#define postage = .01        // 1 cent for each mailing.
#define postage = .1         // 10 cents for each mailing.
#define postage = 1;         // $1 for each mailing.
#define postage = 1.00;      // $1 for each mailing.
```

If the postage variable is not declared and a cost summary is requested then the compiler will assume there is no mailing cost and substitute a value of 0.00 for the postage variable.

EOL

Example: **#define eol = crlf;**
Default: **lf**

The EOL variable stands for "End Of Line." It tells the compiler how to handle a situation when it reaches the end of a line and needs to feed the paper. There are two options you can use for the eol variable:

lf
crlf

LF means that when you reach the end of a line simply send the linefeed character to the printer and continue with printing. This will work for most printers.

CRLF means that when the end of line is reached make the printer issue a carriage return and then do a linefeed.

2. Printer Commands

L has a built in command called PRINTER. It's sole purpose is to send commands to the printer. The command is entered in the following way:

printer " *character_string* ";

Where *character_string* is a piece of text separated by quotation marks. The string can contain embedded escape codes (see Section 1 "HEADER") and is generally used to setup your printer.

With the embedded escape codes you have the ability to send actual setup codes to your specific printer using the \E escape code.

printer "\eq1";
printer "\eq0";

The above example will turn on the Shadow font on my Epson 5000 printer while the second line turns it back off. So you can easily see that custom printing is easily accessible with LL.

The printer command can be issued as many times as needed but can only appear before and after the data block. This was designed for setting up and then resetting your printer's defaults.

3. The Data (Data Block)

The main data block follows both the #defines and the printer commands. This is where your data to be printed goes. In order for the compiler to know about the data we use the key word **address**, after all LL was written for making address labels. Following the word address is the data to be printed enclosed in curly braces. Here is the template:

address (*zip_code*) { *data* } *function* ;

The *function* is completely optional and the available functions will be covered in section 4.

The zip_code which is entered between the parenthesis is optional. If a zip code is entered then the LL compiler will print the United States Postnet Bar Code for the corresponding zip code to the output device. Postnet will be covered in the next section (3.1).

The data is unlimited in size. The data block is allocated dynamically as the program in run so your printing area is only limited by your available memory. If you run out of memory during a compile then the compiler will issue a fatal warning and exit gracefully. Here is an example:

```
address (27302)
        {
        Kevin McHugh
        4671 Mebane Rogers Road
        Mebane, NC 27302
        };
```

Note that the address block doesn't have to be on one line. Because the compiler is intelligent it knows what we mean so we can make the source code more readable for ourselves. When formatting the data block we position the text just as we want it to appear in print. White space before and after any text is omitted and spacing is decided upon your earlier #defines so neatness is not required as LL will take care of that for you.

You may specify as many data entries as you wish and in as many different ways as you wish, for example:

```
address ( )
{
This is my first data block :-)
};

address ( ) { This is my second };

address
( )
{ This is my third and
final one for now :) };
```

So you can see that your style is truly your own. I tend to format mine like I do my C code so you can line up the braces vertically but that is entirely preference.

3.1. The PostNet Bar Code Standard

The United States Post Office uses a bar code on their mailing systems for faster processing and therefore quicker delivery. This bar code is a coded representation of the addressees zip code. The zip code may be either a standard Zip or the newer more accurate Zip+4 code.

Whichever you choose you have the option of letting LL print the bar code for you.

At the release date, version 2.0 only supports bar code printing to Epson and IBM compatible graphics printers. Future versions will fully support the Amiga Preferences graphics standard so any Preference printer may be used.
LL accesses the graphics modes on the printers directly therefore if you choose your output device as PRT: then the graphics characters will be interpreted by Preferences and you will have a row of strange looking characters instead of the bar code. To correct this problem we simply bypass the PRT: device and go straight to the parallel device **PAR:.** This eliminates any interaction on the behalf of Preferences and the bar codes will be printed correctly.

NOTE: PostNet Bar Codes will only be printed correctly if using the **PAR:** device.

Here is a bar code using the PRT: device.

¤¤¤¤¤¤¥¥¥¥¥¥yyyyy¤¤¤¤¤¤¤¤

And here is one using PAR:

All LL needs to know in order to print a Postnet Bar Code is the addressees zip code. The way to do this is to pass the zip code as an argument to the address function. The zip code can be entered as an integer (27302 or 273021234 for the Zip+4), or as a string if you wish to use the dash (-) in the Zip+4 format.

Examples:

```
address (27302)                    // Regular Zip Code As Integer
        {
        Kevin McHugh
        4671 Mebane Rogers Road.
        } print;

address ("27302")                  // Regular Zip Code As A String
        {
        ...
        };

address (273021234)                // Zip+4 Format As An Integer
        {
        ...
        };
```

```
address ("27302-1234")                    // Zip+4 Format As A String
        {
        ...
        };
```

Using the Postnet feature is purely optional. If you do not want the bar code printed then do not supply the Zip code between the parenthesis like so:

```
address ( )
        {
        Kevin McHugh ...
        };
```

4. The Data Block Functions

The data blocks which we discussed above are useless without some functions which tell the compiler what to do with them. Each data block may have one function or none at all. If a data block doesn't have its own function then it will inherit the function from the block before it. Here are a list of functions:

print
hold
copies
exit

The function to call is placed after the right brace and before the semi-colon of its data block. The function will then perform its operations on the data block it is linked to.

PRINT

Example: **print**

The print command simply transfers the data block over to the print spool where it is formatted and then output according to all set variables.

HOLD

Example: **hold**

The hold function clears the data block buffer and returns control over to the compiler without formatting the block or printing it. This is handy if you have a mailing list and you are cutting down on expenses. You can put someone's address on hold and keep it in the list without having it printed. If a data block is on hold it is not charged a postage fee (see Section 1 "POSTAGE").

COPIES

Example: **copies(** *number_of_copies* **)**

The copies function unlike the others takes an argument. The argument is an integer and is the number of copies the compiler should make of the current data block. Copies will simply continue to call the print function until the compiler has outputted the *number_of_copies* requested. This function is handy for printing out multiple copies of messages on the backs of postcards.

EXIT

Example: **exit**

The exit function calls the print routine on the current data block and then terminates compilation. This is used if you only need to print out a certain portion of the mailing list. Using the exit function frees all memory and resources just like a conventional exit.

Here is an example of each function at work:

```
address ( )
        {
        Mike Smith
        123 My Street
        This Town, USA
        } print;
```

This data block will be nicely formatted and then printed to the output file or device. Note that if the data block doesn't contain any text at all then the compiler will issue an error message.

```
address ( )
        {
        Bill Gates
        Microsoft Corp.
        DOSVille, USA
        } hold;
```

This will not print out Bill Gates' address but it will skip it and continue on down the list.

```
address ( )
        {
        PC User
        999 Dos Street
        IBM, USA
        };
```

Notice that this block doesn't have a function. What will happen? Well this block will not be printed but rather skipped. Why? Simple, because the last function that was processed was the hold function so it is still in effect. Check for this when trouble-shooting.

```
address ( )
        {
        Welcome to the monthly postcard!!!
        ------------------------------------------------

        Don't miss this months exciting meeting featuring the all new
        toaster 4000 and GVP's new EGS 24 bit graphics card.

        Also this month will feature officer elections so come prepared
        to vote.

        See you at the meeting!!
        } copies(50);
```

This data block will print 50 copies of the text, neatly formatted and spaced correctly.

```
address ( )
        {
        Bill Johnson
        862 Main Street
        Amigaville, USA 12345-6789
        } exit;
```

This data block will print Bill's address info and then exit the program.

5. Printer Commands

The Printer function is also allowed here after all the data has been processed. A good example here would be to possible reset your printer to its default settings so when LL exits everything will be the same as when LL was started.

For more information on the printer command see Section 2.

6. Functions

The functions are the very last entry in the source file. They are not to be confused with the data functions which execute on the data in each data block but these manipulate the overall process of the compilation.

REPORT

Example: **report( );**

This function is responsible for generating a summary both financial and statistical. Here is a sample report generated by the report function:

```
Report For Compilation And Processing
---------------------------------------------------------------

Filename:             acug.l
Submitted At:         Mon Feb  7 20:36:39 1994
Completed At:         Mon Feb  7 20:36:39 1994
Compilation:          0 Seconds.
-------
Number Of Entries:    20
Entries Spooled:      22
Entries Held:         1
Postage Per Entry:    $0.19
-----------------------------
TOTAL POSTAGE         $4.18

End Of Report.
```

The report generated by the report function is output to a file in the current directory called *report.dat*. If for some reason that file could not be created then the report is sent to stdout usually the terminal screen.

At the time of release no other functions have yet been implemented.

Example Code

Here is a fully functional piece of L code. The names have been changed to protect the innocent:

```
/* demo.l
**
** Sample L Code
** List Of Addresses
**
** Last Updated:           February 01, 1994
**                         Kevin McHugh
**
**
** Standard Postcard Size:  24x60 @ (6 lpi)
**
*/


#define indent   = 20;
#define space    = 12;
#define height   = 24;
#define width    = 60;
#define device   = out;
#define eol      = lf;
#define postage= 0.19;
#define header  = ***-\d-***;


printer "\e2";                          // Selects 6 lines per inch spacing.


address ( )
        {
        Jerry Smith
        123 Main Street
        New York, NY
        } print;



address (27302)
        {
        Kevin McHugh
```

```
        4671 Mebane Rogers Road
        Mebane, NC 27302
        } hold;                         // No Point In Mailing Myself A Postcard.


address ( )
        {
        Steven Jones
        987 Teacup Ave.
        Los Angeles, CA
        } print;

address ( )
        {
        Bart Simpson
        346 Nice Lane
        Foxville, IL
        } print;

address ( )
        {
        Jimmy Albert
        701 Malone Rd.
        Smithville, TN
        } hold;                         // Hasn't Paid His Membership Fees This Year.

address (90210)
        {
        Kenneth Thomas
        0101 Binary Ave.
        Beverly Hills, CA 90210
        } print;


/* End Of Addresses */

report();                               // Generate Cost Report.
```

The above code segment gives a simple but effective look at how easy and versatile the LL compiler is.

Using LL

LL can only be used from the shell or from the Execute command under workbench. The format for LL is as follows:

LL v1.5 (C) Feb 7 1994, McHugh Data Systems
A Label Language Compiler.

Usage: ll <options> <source file>

At the present time LL only supports one option. The -c option which means to compile the code only but not produce any output. This is a handy feature for debugging code. It will run as normal but no output will be produced. However, a report will be produced if requested as it is not part of the formatted output of LL. Example:

Shell> ll -c demo.l

Compile demo.l only. Don't produce output.

Shell> ll demo.l

Compile and produce output based on the file demo.l

Just issuing the LL command by itself will bring up the template.

There are multiple versions of LL in the archive:

LL	*LL for 68000 Amigas*
LL020	*LL for 68020, 68030, 68040 Amigas*
LL020881	*LL for 68020 or better and 68881, 68882 Math Coprocessors.*

Just copy the appropriate executable to C:ll and you're all installed.

Registering LL

LL is freeware but I would appreciate it if you would drop me a line and let me know you are using it; that way I will know whether to publicly release future updates and type all this documentation.

Incase you are reading all these docs in text then you may not know that there is a handsomely bound professional 20 page printed manual which is included in the archive as manual.ps but can be obtained from me by sending cash or money order for $10 US.

As mentioned in the disclosure you can obtain the source code for LL. If you wish to have the source code then send cash or money order in US dollars for $12 and I will mail you the complete source and latest executables. Add $8 for the manual.

Technical Info About LL

LL consists of a top-down one pass parser which is responsible for all syntactical and grammatical findings. The parser is fed a token from the lexicographical scanner which dynamically scans the file and can maneuver bidirectionally.

The scanner was hand written in C rather than using a parser generator such as lex, or flex. The main reason was for speed and grace and the code size was reduced from around 17k to 7k. The parser was also written in C instead of using yacc or bison for similar reasons. The governing reason was that I wanted some practise at optimizing the compile time. As of yet I have not found a file that takes over 2 seconds to compile on my 68030 machine.

Error generation and attempted correction is handled by a subset of the parser in order to continue after minor mistakes like missing = or ; 's etc... Error traps also catch illegal type assignments such as string to integer or float. Such an error will cause a FATAL signal and the compiler will close down as it assumes you will want to edit your code.

When the compiler exits, either normally or otherwise the atexit() function is called. This flushes all I/O and releases file descriptors on all open files and devices. All in use memory is cleaned and then put back on the heap so that your other programs will not be in need for those precious bytes.

If this tickles your fancy consider ordering the source code so you can improve or just learn from it.

Future Improvements

* ***More buffered stream I/O.***
* ***User defined variables.***
* ***Better compiler intelligence and correction.***
* ***Multiple source files.***
* ***Ability to #include text into data blocks.***
* ***Operands (+ - / *)***
* ***More global functions.***
* ***More embedded escape codes.***
* ***Faster, faster, faster.....***

Acknowledgements

Editor:	Cygnus-Ed Professional Release 3.5
Compiler:	Manx/Aztec C Developer v5.2a
Documentation:	WordPerfect v4.2 Amiga
PostNet:	Chris Laforet Software

History

Version	Descripion
1.0	Quick ugly label printer. Used text file for address information seperated by blank spaces. All paper sizes hard coded.
1.1	Changed hard coded paper sizes to #defines for multiple documents.
1.2	Scrapped the whole idea and re-wrote with an intelligent text processor in mind.
1.3	First working version with a seperate token scanner (Lex).
1.4	Deleted all Lex code and hand wrote the scanner from scratch.
1.5	New revised version of the scanner using character based I/O and tokens. Also included seperate parser for language grammar.
1.6	Added variables and Data functions to perform other than printing abilities.
1.7	Added ability to support external functions in the language and coded a function to generate statistical reports.
1.8	Added feature for issuing flags on the command line. Only one exists to date (-c compile only.)
1.9	Postnet Bar Coding was added and yet another change to the grammar to incorporate this into the address function.
2.0	First public release of LL and first revision of user documentation.