Table of Contents

[1. Introduction 2](#_Toc14460553)

[1.1. Why Liberty PrePass? 2](#_Toc14460554)

[1.2. Scope. 3](#_Toc14460555)

[1.3. Running Liberty PrePass 3](#_Toc14460556)

[1.4. LPP Structural Overview. 3](#_Toc14460557)

[2. Declarations 3](#_Toc14460558)

[2.1. Declaration records are initially one of: 3](#_Toc14460559)

[2.2. Declarations breaking the following rules are denoted as multiple declarations 3](#_Toc14460560)

[2.3. If a multiple declaration is found, then: 4](#_Toc14460561)

[2.4. Declaration Records Output 4](#_Toc14460562)

[3. Variables 4](#_Toc14460563)

[3.1. Variable is in variable list 5](#_Toc14460564)

[3.2. Variable not in variable list 5](#_Toc14460565)

[3.3. Variable Records Output 5](#_Toc14460566)

[4. Coding Notes 5](#_Toc14460567)

[4.1. Basic LPP architecture 5](#_Toc14460568)

[4.2. Intermediate data 6](#_Toc14460569)

[4.3. Parentheses 7](#_Toc14460570)

[4.4. Tweeks 8](#_Toc14460571)

[5. Debugging Note 9](#_Toc14460572)

[6. Potential Improvements. 9](#_Toc14460573)

[7. Problems 10](#_Toc14460574)

[7.1. With version 86 10](#_Toc14460575)

[7.2. With version 87 10](#_Toc14460576)

[7.3. With version 88 10](#_Toc14460577)

[7.4. With version 100 10](#_Toc14460578)

[7.5. With version 10x 10](#_Toc14460579)

[7.6. With version 103 11](#_Toc14460580)

[7.7. With version 206 11](#_Toc14460581)

[7.8. With version 210 11](#_Toc14460582)

# Introduction

## Why Liberty PrePass?

Liberty Pre Pass (LPP) is a program designed to catch at least most errors of spelling or caSE in variable names. Liberty Basic (LB) assumes that all non identical variable names belong to different variables. So If there is a spelling mistake, LB assumes a new variable has been introduced – in effect has been implicitly declared. This can easily cause obscure bugs – as the author knows from experience. LPP introduces an explicit variable declaration mechanism **that is transparent to LB.** Based on this LPP scans your code and produces a list of undeclared variables. It also produces lists of ‘multiple declarations’, ‘unused declarations’, ‘all declarations’, and ‘all variables’

Because LB has no formal published definition, it is not possible to assert that LP has been verified against such a specification. So LPP undoubtedly contains bugs. However, think this way. Suppose your program has 1000 instances of variable names, and your typing is 95% accurate. Then there are 50 mistyped variable names. Suppose LPP just finds ‘most’ of these – say 90% - then you are directed quickly to fix 45 of the 50 errors and are only left to find 5 instead of 50 by conventional debug. A 10:1 saving in your miss-spelling debug time at the cost of running LPP on your code – taking perhaps a minute.

Since LB does not have any way of declaring simple local variables LPP identifies the string “’Loc” at the beginning of a line as being a local variable declaration. Of course, LB simply sees this as a comment. The syntax is the same as for an LB global declaration. Of course, multiple instances of the same local variable name are quite legal if they are in different scopes. 'Loc has two types of scope. If a 'Loc occurs within a function or subroutine, then it is local to that function or subroutine. Otherwise its scope is global, and there is no semantic distinction between such a 'Loc and a Global declaration as far as this program is concerned. Either is considered in scope inside or outside functions and subroutines.

Hence:

'Loc x

function fun

'Loc x

will result in a multiple declaration of x being indicated.

So why allow / use ‘Loc outside of a function / subroutine instead of Global? The idea is to allow for a distinction – where the coder sees it useful – between truly global variables likely to be encountered anywhere in large areas of code, and those that are not local to a single function or subroutine but are only used within a small area of code – perhaps within a page. It can make sense to put global declarations at the beginning of the code, but ‘Loc declarations at the head of the code area their variables are used in.

## Scope.

This version of the Manual is compatible with Pre-Pass 211, which is designed for LB 4.5.1

## Running Liberty PrePass

The Pre-Pass nnn.bas file can be copied into any convenient location. When Run it will create a subfolder to that location called “LBPrePasOutput”, open an explorer window asking which file you want analysed, create the four output files in the “LBPrePasOutput” folder, and signal completion. These output files are comma delimited text and may be read as is, or imported into Excel for nicer formatting or sorting. The line numbers quoted in the output are compatible with the line numbers in Notepad++.

And that’s all you need to know to run LPP. If, and only if, you are interested in LPP’s internals, read on.

## LPP Structural Overview.

LPP has a conventional two pass structure that first captures the declarations, and then attempts to match variables against these declarations. Since the author has no background in compiler writing, LPP’s code may not follow standard compiler coding practise. The code is essentially driven by the next two main sections. The first defines the format of the records used to save the information on the declarations, and then goes on to deal with duplicate / multiple declarations. The next section records what is known about variable instances prior to trying to match the variable instance against the declaration data and then defines the rules used to determine if a variable instance matches a declaration.

Once this basis is established, succeeding sections deal with special cases, coding, etc..

# Declarations

## Declaration records are initially one of:

* + 1. <variable name>", is an Array, "
    2. <variable name>", is a Local, " "Declared in line, "<Line number>
    3. <variable name>", is a Global, " "Declared in line, "<Line number>
    4. <variable name>", is a Function, " "Declared in line, "<Line number>
    5. <variable name>", is a Subroutine," "Declared in line, "<Line number>
    6. <variable name>", is a local to Function, "<Function name>", and is Declared in line, "<Line number>
    7. <variable name>", is a local to Subroutine, "<Subroutine name>", and is Declared in line, "<Line number>
    8. <variable name>", is a parameter of Function, "<Function name>", and is Declared in line, "<Line number>
    9. <variable name>", is a parameter of Subroutine, "<Subroutine name>", and is Declared in line, "<Line number>

## Declarations breaking the following rules are denoted as multiple declarations

LB PrePass distinguishes two types of multiple declaration. Declarations of the same name and of the same scope are simple multiple declarations. Two types of “Scope” are defined here: outside all funSubs, or within each specific funSub. So the number of scopes in a program is the number of funSubs plus one. Since “Global” declarations, and “’Loc” declarations outside a funSub are visible within all funSubs, use of the same name as one of these for a ‘Loc or parameter declaration within a funSub is also identified as a multiple declaration and categorised as of ‘overlapping scopes’. Declaring a name as a Global and as a ‘Loc outside of all funsubs is categorised as of ‘overlapping scopes’. This could be considered as a multi declaration, since the scope of a ‘Loc outside a funSub and a global are the same. The choice of ‘overlapping scopes’ is made to be consistent with section 1.1’s rational for allowing ‘Loc outside of funsubs.

* + 1. Item
    2. Items and must not have the same <variable name>
    3. The same <variable name>" may be used for either 1.1 or 1.2 and any other declaration
    4. Declarations 1.5 – 1.8 may use the same <variable name> in differing funsubs but not within a funSub

## If a multiple declaration is found, then:

* + 1. If the found record has the same format as section 2.1, then this is its first duplication, and:
       1. “Declared in line,” is replaced by “First declared in line,”
       2. One of two strings is appended to the record:
    2. If two declarations of the same name are in overlapping scopes, then the string is ", but is declared in overlapping scopes"
    3. In all other cases the string is ", is declared multiple times."
       1. Subsequent occurrences of a specific multiple declaration are listed.

## Declaration Records Output

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 . . . |
| <variable name> | is an Array | - | “is Used” if used, else “is NOT Used” | “and is Declared in line” | <Line number> | “-“ or  “but is declared in overlapping scopes”  or  “is declared multiple times” | List of multiple occurrences or “-“ |
| is a Local |
| is a Global |
| is a Function |
| is a Subroutine |
| is a local to Function | <Function name> |
| is a local to Subroutine | <Subroutine name> |
| is a parameter of Function | <Function name> |
| is a parameter of Subroutine | <Subroutine name> |

# Variables

(Note the final format of variable records is defined in section 3.3. The sections following show how this is built up.)

When a token is identified as a legal variable name, it is checked to see if it is in the variable record list. This includes both the “name” of the variable, and its location. This location classification is saved in the variable VarType$, which can have one of three values:

* + - “, is outside all FunSub, “
    - ", is in Function, "<Function Name>
    - ", is in Subroutine, "<Subroutine Name>

## Variable is in variable list

If “name” is in the var list, it is checked to see if where it was found is consistent with current entry in the var list entries. Note that there should only be one entry with a given name and location, so if a matching name / location pair is found the entry is amended by incrementing the ‘used counter’. If no match of both “name” and Location is found – then this variable - “name” - has not been found in the var list.

## Variable not in variable list

If the variables list does not contain an entry for “name”, then “name” is added to the variables list, together with the classification of its location information and its line number.

* + 1. Once the variable ‘name’ is added to the variable list a check is then made to see if the entry is compatible with a declarations list entry. Note that declaration records are unique. The check is repeated throughout the declarations list until either a match with qualifiers consistent with where the var was found is identified, or the declaration list is exhausted, and then the variable entry in the variables list amended as follows.
       1. If a there is a declaration for “name” as a Local or Global it is assumed to match wherever found so its entered into the var list as a declared var
       2. If a there is a declaration for “name” as a Function it matches only if it is in the “name” function, otherwise it is an un-declared variable.
       3. If a there is a declaration for “name” as a Subroutine then it is an un-declared variable.
       4. If a there is a declaration for “name” as a local to Function, <Function name>, then it matches only if it is found in the “name” function, otherwise it is an un-declared variable
       5. If a there is a declaration for “name” as a local to Subroutine, <Subroutine name>, then it matches only if it is found in the “name” subroutine, otherwise it is an un-declared variable
       6. If a there is a declaration for “name” as a parameter of Function, <Function name>, then it matches only if it is in the “name” function, otherwise it is an un-declared variable
       7. If a there is a declaration for “name” as a parameter of Subroutine, <Subroutine name>, then it matches only if it is in the “name” subroutine, otherwise it is an un-declared variable.
       8. If no matching declaration found then enter it in the var list as an undeclared var.

## Variable Records Output

Following the preceding processing the list of variables consists of a list of records each of which is of one of the following formats:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| <variable name> | is Outside all FunSub | Global | Used | <#> | times - eg in line | <Line #> | but NOT declared |
| is Outside all FunSub | Global | Declared |
| is in Function | Function name | Declared as local to Function |
| is in Subroutine | Sub name | Declared as local to subroutine |

# Coding Notes

## Basic LPP architecture

Determine from the user the input file and the <base path> of the outputs files.

Open the files

Read the input file a logical line at a time, discarding comments and empty lines.

For each line:

Check for declarations, and if found:

and already declared, add a multi declaration flag to the existing declaration record

and not already declared, add a new declaration record.

Re-read the file as before:

Check for variables. If found, and:

there is no matching declaration, & there is no matching undeclared record -

add a new undeclared record with a <undeclared once> flag

there is no matching declaration, & there is a matching undeclared record -

change the <undeclared once> flag to an <undeclared multiple times> flag to the matching record.

there is a matching declaration -

add the instance to the array as a declared variable.

("Records" here are simply elements of one of two arrays. One for declarations and one for variables.

Each array is one dimensional, and consists of a string for the token name and another string as the "flag".)

Each "record" is initially in the DecArray$ or VarArray$. These are then filtered without change as needed for the different files.

The records are formatted as one of the following forms. The comma delimiting is to facilitates import into Excel as a comma delimited file.

The output process simply:

1. Save all the Declaration records that are not multiply declared to the <base path>GoodDeclarations.txt file
2. Save all the Declaration records that are multiply declared to the <base path>MultiDeclarations.txt file
3. Save all the undeclared variable records to the <base path>UndeclaredVariables.txt file
4. Save all variable records to the <base path>AllVariables.txt file

## Intermediate data

* + 1. The variable "DecType$" consists of just the first pre-defined string in each declaration record (section 2), without the commas and spaces

(e.g. ", is a Function, ") except for the last four record types listed above, where it consists of <the first pre-defined string> ","

followed by the Function / subroutine name. (e.g. ", is a parameter of Function, “<FunctionName>)

* + 1. "tokens" are the names of either declared or used variables (note that the term is used in the code comments to identify some text which MAY

be a variable of some sort) and are identified by the program as follows:

4.2.3.1 Variable Declarations

<beginline><dec><token name(s) comma delimited>

<dec> = global | dim

<beginline>Function (<Function Name>"("<parameter name(s) comma delimited>")"

(Function and parameter names are both declarations)

<beginline>Sub <Subroutine name>" "<parameter name(s) comma delimited>

(Only the subs parameter name(s) are checked here - LB checks the calls)

<beginline>'Loc <token name(s) comma delimited>

Note 1. "global", dim, "function"", and "sub" may be of any case, but "'Loc" must be as shown.

Note 2. In the case of an "Input" command the variable(s) at the end of the command are considered uses,

not declarations

Note 3. <beginline> ignores whitespace

4.2.3.2 Variable Uses

a text string not enclosed in "" appears before or after any of, <, <=, >, >=, =, <>, and, or, xor, +, -, \*, /, ^, ","

or after "to", "step",

a text string appears before "(". (See later, for more on the use of parentheses)

a text string appears after a "call".

Note 1. Re commas. These can appear in the following circumstances:

\* multiple array declarations and print statements. These are dumped in the Var prepass code.

\* case statements with multiple conditions and in the parameter lists of subroutines and functions. These

are converted to "~". (The text "case" is dumped in the Var prepass.)

Note 2. The following are ignored as the compiler should pick up use without declaration:

Handle & label names,

GOTO destinations

## Parentheses

Parentheses occur in several circumstances, as part of the native language functions, as part of the program defined functions and their use, in array

declarations and use, within the RHS of assignments, and within the conditionals of "if" and "while" statements. The following steps are

followed in order as part of the Variable check pre pass, and hence after the declaration checks. The process enters if a

"(" is found, and exits when no more "(" exist in the line.

There may be more than one of each of these types and possibly a mixture within a line so the steps are iterated until no parentheses

exist in the line. So for each of the four steps, if the detection is positive line return to "A". Since A4 is

effectively an 'else' it automatically has a 'positive detection' and it returns to "A". Because of this return it is necessary to eliminate

multiple "~" occurrences - with possible spaces. This seems to deal with nested as well as serial parentheses - though I cannot prove this.

A. If a string "<text>(<one or more comma delimited numbers or variables>)" is encountered, it may be a function call to

a native or declared function, or an undeclared function or array reference.

A.1 Call to a native function.

A.1.1 Detection. Compare "<name>(" with the list of native functions.

A.1.2 Actions

If it is a call to a native function, delete the function name and replace the parentheses and any

internal "," with "~" . This allows the parameters to be checked in the normal 1.1 var process.

A.2 Call to a declared function.

A.2.1 Detection. Compare "<name>(" with the DecArray$ names. This must hence follow the DecArray addition for this line.

A.2.2 Actions

If it is a call to a declared function, follow the A.1.2 process above. This allows the parameters to be checked in

the normal 1.1 process.

A.3 Use of a declared array is not tracked since this is done by LB, neither are the element values checked. If an element

value is a variable LB will check if it / they are globals and if not LB assumes it / they is / are new var(s) and set

them to 0. This will then produce a runtime error if the array is accessed for any element > 0.

A.3.1 Detection. Track array declarations (names only) and compare "<name>(" with the list of declared arrays.

A.3.2 Actions

If it is a use of a declared array, delete the complete use from <name> to ")" - since we don't track use of declared arrays

A.4 'Call' to an undeclared function or array (No generic way of telling the difference).

A.4.1 Detection. We got here.

A.4.2 Actions

If 'name' is a valid Var form, add it to the var array with an 'undeclared array or function' flag and a line number.

Delete the function / array name and replace the parentheses and any internal "," with "~" . This allows the parameters to

be checked in the normal 1.1 var process.

2. Parentheses in assignment statements or following an "if" or "while" which not related to functions or arrays can be deleted.

Variables within a function or array within an assignment must be treated as though they were not inside the parentheses, so the

function / array name, including the "(" and the immediately following ")" must be deleted. The "," between () must be replaced by "~".

## Tweeks

4.4.1 Variable use inside functions and subroutines ("funsubs").

4.4.1.1 When a Subroutine or Function is declared, then the formal parameters are local declarations equivalent to a 'Loc inside the funsub.

If a 'Loc declaration occurs inside a function or subroutine, use of that variable outside the funsub is a

"NOT declared" use. To detect this, a running note is kept of FunSub starts, which is cancelled when the FunSub is

completed. This 'note' is held in variable "CurrentFunSub$" - which is either blank - we are not inside a FunSub - or

contains:

"Subroutine”| “Function” "," <name of the function or subroutine>

4.4.1.2 When a 'Loc is encountered this is first handled by the standard process of Section 1. Then it’s checked to see if this is

a unique declaration with no scope conflicts. If so it is added to the DecArray$.

The scope conflict is checked as follows:

if "CurrentFunSub$" is blank it is checked to see if this is a multiple declaration at the 'global' level – resulting in a “multiple declaration” message

else it is checked to see if this is:

a multiple declaration within the current funSub – resulting in a “multiple declaration” message

or if this funsub’s ‘Loc has the same name as either a Global declaration or a 'Loc declaration outside of funsubs – resulting in a “declared in overlapping scopes” message

or if this funsubs parameter has the same name as either a Global declaration or a 'Loc declaration outside of funsubs, or a ‘Loc within that funsub– resulting in a “declared in overlapping scopes” message

When the FunSub definition is complete CurrentFunSub$ is cleared.

Subsequently sorting into the GoodDeclarations and MultiDeclarations files happens in just the same way as the other

declarations.

4.4.1.3 When a variable is encountered this is first handled by the standard process of Section 1. This will check to see if

the declared variable exists as an existing declaration. Multiple situations can exist:

1. The variable is not in the DecArray$ list

This is an 'undeclared variable' situation handled thereafter by the basic process item c).

2. The variable is in the DecArray$ list & is not identified there as a FunSub Local declaration

This is a normal situation handled by the basic process item d).

3. The variable is in the DecArray$ list and is identified there as a FunSub Local declaration for this FunSub

This is an declared local use and is recorded as such.

4.4.2 Array Sizing

The inflexibility of Liberty's array sizing is a pain here. The current code simply defines fairly large arrays (1000) and

crashes with an error message if this is exceeded. This could be modified to keep reading the input file without storing into the problem

array, but counting occurrences. Once EOF is reached REDIM to the computed size and start over.

4.4.3 Destination folder definition

There seems to be no neat way in LB for the user to identify a folder to the program which is sticky and not

misleading - without getting into multiple non obvious (to me) API calls. Two methods are coded here. The

implemented one is not sticky, the commented out version is sticky, but is misleading in that it requires clicking a

"Save" button when a save is not wanted - and does not occur.

4.4.4 Line numbers, file dates

Line numbers are attached to some records in the output files.

All output files contain the date and time of creation in the file name. The last set of digits are the number of

seconds since midnight – just to ensure uniqueness.

# Debugging Note

In the code globally edit replace “xcopy” with ’\*ycopy to eliminate redundant statements which are only there to facilitate debug - and vice versa. Also note that some debug code uses mainwin.

# Potential Improvements.

1. Make the array size self adjusting
2. Use LB Booster per http://www.bbcbasic.co.uk/lbb/lbb.html

or GNU/Liberty Basic Compiler Collection at <http://www.filebuzz.com/files/Liberty_Basic_Dll/1.html>

1. Make real Block structure – at least for ‘Loc’s. Say ‘BEGIN and ‘BEND. ‘Loc only allowed inside funsubs and blocks.
2. Stop Sub names being recorded as variables? Means that some other way of recording unused sub declarations needed.

# Problems

## With version 86

1. **Problem**

A variable declared locally, and then multiple declared within a FunSub is then registered as a multiply declared local, e.g.

“xcopyIL$, is a Local, First declared in line, nnn, is declared multiple times,mmm”

The “nnn” refers to the first occurrence. If this was, for example a global type ‘Loc, then the first occurrence inside a funSub should have its line number listed as a multiple declaration – but it is not. Currently, “mmm” identifies the second declaration in the funSub.

1. **Change**

The meaning of “MixedGloDec” extended to cover the ‘overlapping scopes’ (e.g. global and funSub local) case and rename to “OverlappingDeclarations”. Affects Subs “AddMdFlagToTokenInDecList”, “SaveBadDecToFile”, AddMdFlagToTokenInDecList, and Function “TokenInDecList()”

## With version 87

1. Problem

“Variables Used” shows all variables as “used” just once

## With version 88

1. Problem

Still an issue with overlapping scopes, global and parameter of the same name not flagged as illegal, and declared as local in a function and in a sub is legal but flagged illegal

## With version 100

(BTW “xcopy…” entries eliminated)

MultiDec file says “count, is a local to Function, GetLine, First declared in line, 236, is declared multiple times,785” – but these are in differing subs. There are two other examples.

Some variables are shown in the VariablesUsed file with a space, e.g. w 3$ when w3$ is used.

Also Sub VarPrePass refers to “section 1.3 in the introduction” in its comments – which is no longer true.

## With version 10x

When there is a number in the variable name this is incremented instead of the multiple declarations counter.

## With version 103

No known problems.

## With version 206

No known problems.

## With version 210

No known problems – though recording subs as variables is arguable.