# List Processing Procedural Language

*Brian Smith*
*Programming Languages and Translators*
*August 10, 2007*

**TABLE OF CONTENTS**

# *Chapter 1: Introduction*

## 1.1 Language Proposal

The List Processing Procedural Language (LPPL) is a functional language, such as C, specifically designed to benefit the processing of arrays of data (lists). The syntax caters to the list, easing the complexity of reading and manipulating them. It often takes many statements in other programming languages to accomplish what can be done in one statement in LPPL. It is also beneficial to program in LPPL when dealing with "hierarchical" lists. For example for a list that consists of "teachers" and list that consists of "students", a list might be needed to consist of "people". In any other language, whenever adding or removing from the list of "teachers", the list of "people" will also need to be maintained. In LPPL manipulation of list "teachers" or "students" does not necessitate the need to maintain the list of "people" as well.

## 1.2 Language Properties

There is a concept of a containment list. A containment list is a list which contains other lists in its definition. For example consider the following two lists defined:

```
list1 as int list;
list2 as int list;
```

A containment list could be defined like so:

```
list3 as int list contains(list1,me,list2);
```

The number of elements in list3 would be the size of list1 + the size of list2 + any elements specifically added to list3 (me). This makes these lists dynamic, such that whenever a contained list is modified, the containment list is also modified. Lists can also be defined to contain other lists returned from a function: for example consider a module with these functions:

```
Function getParents;
  input;
  output: int array;
Function getChildren;
  input;
  output: int array;
```

A containment list could be defined like so:

```
list3 as int list contains(getParents(),me,getChildren());
```

It could also be a combination of the two:

```
list3 as int list contains(me,list1,getParents());
```

If unsorted, when iterating through list3, the order will be defined by the *contains* clause, *me* being any elements exclusively added to list3. Also any operations on the containment list which affect the contained list do not directly modify the actual contained list, only how the containing list perceives it. However manipulating a contained list directly will effect how a containing list perceives its list.

The primitive data types are:

int – 32 bits
char – 8 bits
string – char list

Multi-dimensional lists are not allowed (due to time constraints). All elements in a list must be of the same type.

Some list specific statements in the language
*list1* union *list2* – combine *list1* and *list2*, producing a new array with no duplicate results
*list1* intersect *list2* – produce a new array which contains entries both *list1* and *list2*
*list1* exclude *list2* – produce a new array which contains *list1* without entries which were also contained in *list2*

# *Chapter 2: Tutorial*

This tutorial attempts to walk you through the needed steps to write and run your first LPPL program. LPPL is beneficial to list processing, so we will write an LPPL module which utilizes it best. We will write a module which takes as input a number of lists and builds a containment list to eventually print out.

First create the file testMod.lppl, note that all LPPL modules must have the extension ".lppl". Since LPPL doesn't currently have the capability of accepting input at runtime, we will resort to using constants defined in the module as input. Constants for a module must be defined after the module definition, separated by a comma. Enter the following module definition and constant declarations in the new testMod.lppl:

```
module testMod;
  teachers as string list("Stephen","Susan","Lewis"),
  students as string list("Brian","Peter","Mark","Christine"),
  faculty as string list("Bob","Carla","Steve"),
  goodRc as int(0);
```

Every executing module must have a main function, this is the starting point for every module, which takes nothing as input and as output returns one integer, for the return code. The main function definition can be entered like so:

```
function main;
input;
output:
  rc as int;
```

Notice we decided to define the return value to be stored in variable *rc*. The name of this variable can be whatever you like; the type is all that matters. Lets briefly assume there is another function in this module, which we will write later, with the signature (int,string list) *getAll*(string list, string list, string list). This signature implies *getAll* takes in three string lists as input, and returns an int and a string list as output. It is possible to define multiple outputs. Both the input and output parameters are positional, all parameters for the input list are mandatory, but if you don't care about a given output parameter, leave it blank, and continue with the parameter list.

```
do;
  out as string list,
  x as int;
  rc = goodRc;
  (rc,out) = getAll(teachers,students,faculty);
  check(rc in goodRc)
  do;
    foreach(x in out)
    do;
      >x;
    end;
  end;
  check(rc notin goodRc)
  do;
    > "BAD RC";
  end;
end;
```

At the start of every control block (do end group), you have the option to define variables local to that control block scope. The variable definitions must be the first thing in a control block, and continues until a semicolon is found. After defining the local variable, the main function calls getAll which returns a list containing all elements in the lists teachers, students, and faculty. If we didn't care about the first output parameter (the return code) we could code the function call like so:

(,out) = getAll(teachers,student,faculty)

After getting some list, we check the return code, and if it is okay we will iterate through the list and print out each element. This brings us to the getAll function, defined below:

```
function getAll;
input:
  list1 as string list,
  list2 as string list,
  list3 as string list;
output:
  rc as int,
  outList as string list;
do;
  tempOut as string list contains(list1,list2,list3);
  outList = tempOut;
  rc = goodRc;
end;
```

Though there are other ways to implement, this is the easiest. The variable *tempOut* is a containment list which contains the three lists passed in. Another way of doing this would be:

```
function getAll;
input:
  list1 as string list,
  list2 as string list,
  list3 as string list,
output:
  rc as int,
  outList as string list;
do;
  outList = list1 union list2 union list3;
  rc = goodRc;
end;
```

A side effect of this method however is there are no duplicate values. Should there be both a teacher and a student named "Bob", only one would be added to the receiving list.

The full source for testMod.lppl is as follows:

```
module testMod;
  teachers as string list("Stephen","Susan","Lewis"),
  students as string list("Brian","Peter","Mark","Christine"),
```

```
  faculty as string list("Bob","Carla","Steve"),
  goodRc as int(0);

function main;
input;
output:
  rc as int;
do;
  out as string list,
  x as int;
  rc = goodRc;
  (rc,out) = getAll(teachers,students,faculty);
  check(rc in goodRc)
  do;
    foreach(x in out)
    do;
      >x;
    end;
  end;
  check(rc notin goodRc)
  do;
    > "BAD RC";
  end;
end;

function getAll;
input:
  list1 as string list,
  list2 as string list,
  list3 as string list;
output:
  rc as int,
  outList as string list;
do;
  tempOut as string list contains(list1,list2,list3);
  outList = tempOut;
  rc = goodRc;
end;
```

Execution of the module relies upon the Java Virtual Machine and certain LPPL libraries. Run java edu.lppl.lppl testmod.lppl with the LPPL and ANTLR libraries in the java classpath and testmod.lppl in the current directory.

```
> java –classpath antlr-3.0.jar;lppl.jar edu.lppl.lppl testmod.lppl
```

Output will be printed to the system console. The output for testmod will be:

```
Stephen
Susan
Lewis
Brian
Peter
Mark
Christine
Bob
```

```
Carla
Steve
```

Currently, modules cannot reference functions from other modules, making a module a contained executed program. I hope you can now write LPPL.

# *Chapter 3: Language Reference Manual*

## 3.1 Lexical Conventions

There are 6 different types of tokens: identifiers, keywords, constants, literals, expression operators, and separators. Separators are any white space in the input stream such as blanks, tabs, and new lines. At least one of these separators is required between any two adjacent identifiers, keywords, constants, and literals.

### 3.1.1 Comments

The characters /* represent the beginning of a comment, and can span multiple lines until the terminating */ character is found.

### 3.1.2 Identifiers

An identifier is a sequence of letters, numbers, and the '_' character, with the first character being a letter. LPPL is case insensitive, 'Foobar' and 'foobar' are the same identifier. An identifier is significant up to the first 255 characters. Any list is represented in the code by an identifier.

### 3.1.3 Keywords

The following tokens are reserved to the language and cannot be used as an identifier. These reserved words are also case insensitive.

| | | | |
|---|---|---|---|
| module | function | input | output |
| as | list | contains | int |
| char | string | do | end |
| remove | keep | find | arrange |
| ascending | descending | all | foreach |
| in | notin | null | intersect |
| union | except | check | |

### 3.1.4 Constants

A constant is a static value for an identifier that can not be changed at any moment. Constants are defined at the beginning of a module. There are different types of constants:

#### *3.1.4.1 Integer Constant*

An integer constant is a sequence of digits, 0 – 9.

#### *3.1.4.2 Character Constant*

A character constant is a single representation of a character surrounded by single quotes. Certain characters not easily recordable are defined using an escape character followed by the appropriate code:

‘\n’ new line
‘\t’ tab
‘\’’ single quote
‘\\’ \

#### *3.1.4.3 String Constant*

A String constant is a sequence of characters surrounded by double quotes. Like the character constant, the same escape character sequence is used for not easily recordable characters with the exception of

“\”” double quote

#### *3.1.4.4 Defined Constant*

A defined constant is one defined by the programmer at the beginning of a module. Within the module, this constant can be substituted with its equal constant. Note that list constants can only be a defined constant, and can not be embedded throughout the module.

## 3.2 Variables

Variables represent a list defined in storage, or null (a reference to nothing). A list can be of type **int** (decimal), **char** (character), or **string** (a sequence of characters). The lifetime of a variable is determined by its surrounding Do-End block, the longest a variable can live is the time in a given function (variables cannot be global to a module). If a variable is not defined as a list, and defined as a single **int**, **char**, or **string**, it is treated as a one-element list. To define a variable as a multi-element list, the **list** keyword is appended to the declaration.

### 3.2.1 Containment variable

A variable can be defined to contain other variables transparently and dynamically using the **contains** keyword. Such a variable is called a containment variable. An update to a contained variable will be recognized by its containment variable. A containment variable can also contain the results of a function call. This function call however can only have one output variable and must have no input variables. For this case, each time the array is referenced, that function is called which is expected to return a list. A containment variable may contain any number of containing variables and other containment variables, and may also have its own explicit entries using the **me** keyword.

### 3.2.2 Receiving Variable

The receiving variable is the variable on the left of an assignment statement, which will receive a new assignment based on the right side of an assignment statement.

## 3.3 Conversions

Any combination of lists, regardless of type, may be mixed together to achieve a desired result. The resulting type depends on the receiving variable's type. When converting an int type to a char or string type, the char or string representation of that int (number) is produced. When converting a char or string type to an int type, the char or string is first checked if it represents a string int (number), and converts appropriately. If it isn't, no explicitly conversion is done and the ASCII code is used explicitly as the int.

## 3.4 Expressions

Expressions are the sequence of variables and constants split up by operations. The operations defined in LPPL follow the following precedence, from highest to lowest

( )
*
+ -
| union exclude intersect

### 3.4.1 ( expression )

A logical means of giving precedence to variables when that would not normally be the case. For example 6 + 3 * 2 normally would execute 3 * 2, then add 6. We could use ( ) to make it (6 + 3) * 2, which would execute 6 + 3 first, then multiply by 2.

### 3.4.2 expression * expression

The * operation denotes multiplication. The two expressions are multiplied together, following the conversion techniques defined in section 5.

### 3.4.3 expression + expression

The + operation denotes multiplication. The two expressions are added together, following the conversion techniques defined in section 5.

### 3.4.4 expression – expression

The – operation denotes subtraction. The second expression is subtracted from the first expression, following the conversion techniques defined in section 5.

### 3.4.5 expression union expression

The *union* operation returns a list which contains all unique entries from the first expression and the second expression, discarding duplicates, and following the conversion techniques defined in section 5.

### 3.4.6 expression exclude expression

The *except* operation returns a list which contains entries from the first expression that are NOT in the second expression, following the conversion techniques defined in section 5.

### 3.4.7 expression intersect expression

The *intersect* operation returns a list which contains entries that exist in both the first expression AND the second expression, following the conversion techniques defined in section 5.

### 3.4.8 expression | expression

The | operation denotes concatenation. It returns the second expression concatenated with the first expression, following the conversion techniques defined in section 5. Though geared toward strings, chars and lists, this can also be used for ints ( 2 | 6 would be 26).

## 3.5 Function Interface

Function interfaces are used at the beginning of a function to define a functions input and output variables. The function interface has the form:

**function** *functionName*;
*input:*
    *var-list;*
*output:*
    *var-list;*

Where *functionName* is the name function callers use to refer to the function and *var-list* is a standard variable declaration (see below). If a function has no input variables and no output variables then the function interface has the form:

*input;*
*output;*

It is possible to have multiple inputs and multiple outputs, in the case where a function call is within an expression, only function calls which return one output are allowed.

## 3.6. Statements

Statements are executed in sequence, and can be one of the following types

### 3.6.1 Assignment Statement

Most assignment statements have the form

*var = expression* ;

There is another special assignment which has the form

> *expression ;*

This outputs the result of *expression* to standard out as a string type. This should be used for debugging and learning purposes only.

### 3.6.2 Control Statement

A control statement controls the order and reoccurrences of statements, and whether statements are executed or not. There are two types of control statements: iterative (foreach) and conditional (check), both operating on lists.

#### *3.6.2.1 foreach ( var1 in var2 )*

This iterative statement will execute the following do-end block n times, n being the number of entries in *var2*, assigning *var1* as a one-element list to the current entry each iteration. *var1* exists only within the scope of this do-end block while *var2* exists in prior scopes. *var1* will get its type from the underlying type of *var2*.

#### *3.6.2.2 check ( var1 in var2 )*

This conditional statement will execute the following do-end block only if single-element *var1* is an entry in list *var2*. Instead of **in**, **notin** may be used, which will only execute the following do-end block if it is not an entry.

### 3.6.3 Do Statement / End Statement (do-end block)

The do statement marks the beginning of a scope and its associated end statement marks the end. All functions must begin with a do statement and finish with an end statement. This holds true for control statements as well. Although they are required in these two conditions, they are not limited there, and can be used throughout a function.

## 3.7. Scope

The Scope of names declared in a given do-end block extends from the beginning of the current do statement to its associated end statement. Any variables defined here override any previously defined variables with the same name. Upon leaving this scope, the previous variable becomes active. Within a current scope, a variable name can be defined only once.

# *Chapter 4: Project Planning*

## 4.1 Processes

The development process for LPPL was implemented according to the logical steps involved in compiling and interpreting LPPL. The three logical pieces are language design, AST walking, and interpreting. Language design involves defining the language syntax and building the abstract syntax tree. AST walking includes semantic analysis and manipulating the AST either for optimization or trimming (removing nodes that aren't needed for correct execution). Finally the interpreter uses the final abstract syntax tree to execute the module. Even though this was a one person project, an interface was defined to separate the language design component from the AST walking and interpreting components.

## 4.2 Programming Style

The following conventions are used to ensure the most adaptable, portable, and readable code. Since this is a one person project, the style is beneficial to that one person and doesn't take necessarily take into account common practices.

1) Program to interfaces when possible.
2) Only cast when needed, and cast to the highest type needed to accomplish what is needed.
3) Opening block brackets are at the end of the statement, code block closing bracket are on its own line, even with opening bracket line's indentation.
4) Each indentation is one tab.
5) Line length should be no longer than 100 characters. If a statement is longer, the next line should be indented logically.
6) All variable should be self-documenting. The meaning of a variable should be clear without comments.
7) Global variable begin with g_, and are private.
8) Static variables begin with s_, and are private.
9) Static final variables are all uppercase, and may be public.
10) Isolate functionality which relies on a third party library from functionality that is independent from any third party libraries.
11) Javadoc all functions and classes
12) Comment only in confusing areas to assist in decrypting the code. The javadoc should take care of most questions of functionality.

## 4.3 Project Timeline

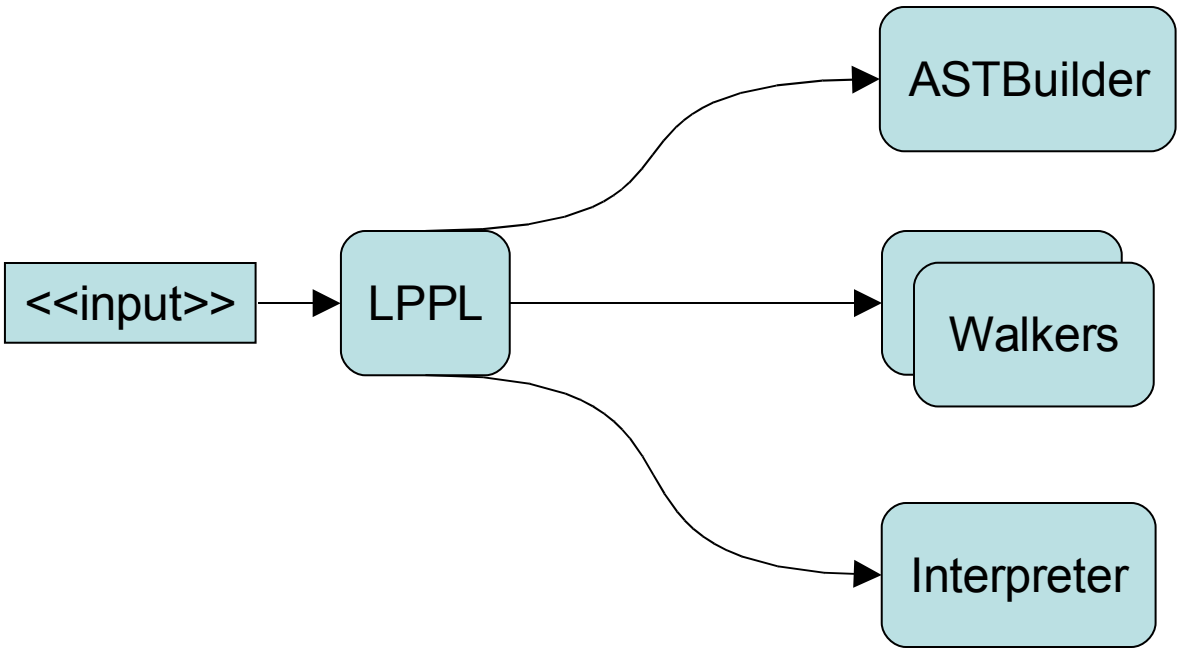
| Section | Description | Date Completed |
|---|---|---|
| White Paper | Dream the language | 6/11/2007 |
| Language Design | Design the language | 6/25/2007 |
| LRM | Language Reference Manual | 6/27/2007 |
| High Level Design | Design the interpreter | 7/14/2007 |
| Test Language | Make sure language accepts appropriate syntax | 7/15/2007 |
| AST Structure | Modify ANTLR language file to build an Abstract Syntax Tree | 7/21/2007 |

| | | |
|---|---|---|
| Test AST Structure | Make sure AST structure is accurate | 7/22/2007 |
| ANTLR to LPPL convert | Convert from ANTLR dependent tree to LPPL tree | 7/29/2007 |
| Test convert | Test to make sure nothing is lost in conversion | 7/29/2007 |
| Necessary Walkers | Create necessary walkers such as building symbol table, semantic analysis, etc. | 7/29/2007 |
| Test Necessary Walkers | Focus mostly on semantic analysis | 8/5/2007 |
| Optional Walkers | Create optional walkers, such as optimization walkers | N/A |
| Test Optional Walkers | Make sure running with optional walkers runs the same as running without optional walkers | N/A |
| Final Paper | This paper | 8/10/2007 |

## 4.4 Development Environment

The LPPL lexer, parser, and AST builder are developed using ANTLR 3.0 which generates the appropriate java source. The interpreter and other components of LPPL which aren't generated by ANTLR are written natively in the Java 1.4.2 language. The Eclipse IDE is used to create and modify the source code.

# *Chapter 5:  Architectural Design*




## 5.1 Interfaces

### 5.1.1 ASTBuilder

The ASTBuilder is the interface which separates code which is functionally dependent on third party libraries from code which doesn't rely on any third party libraries. The ASTBuilder defines the methods for an implementation to perform lexical and syntactical analysis, and to build an Abstract Syntax Tree in an implementation independent way. The ASTBuilder implementation is defined in the lppl.properties file. In this base implementation, ANTLR is the third party library, and there is an ANTLRFacade which implements ASTBuilder and converts from ANTLR's AST into the LpplNode tree.

### 5.1.2 LpplNode

LpplNode represents a node in the internal LPPL Abstract Syntax Tree. ANTLR produces an AST, but it is dependent on the ANTLR libraries. Because of this, LpplNode is used to keep LPPL independent from ANTLR. Within the ANTLR component, there is a converter which turns the ANTLR AST into an AST made up of LpplNodes (or some class which extends LpplNode, such as LpplOpNode). It is possible to do this using the language file, but for readability of the language file, it was decided to do this programmatically.

### 5.1.3 TreeWalker

The TreeWalker interface is implemented by any classes wishing to walk the LPPL AST. This allows a dynamic number of optimizations and manipulations to occur depending on the users requirements. The number of walkers, the order they are executed, and the implementing classes are defined in the lppl.properties file.

# *Chapter 6:  Test Plan*

To test LPPL, no automation was used. Instead, test modules were created to test (hopefully) all aspects of the languages. For each component, modules were created which were correct, and modules were created which had errors. These modules were then run and their expected behavior was compared to their actual behavior manually. For AST structure validation, a syntactically correct module which consisted of all aspects of the language was created, and the AST was "pretty printed" to the console. The AST structure was than manually validated. See appendix D below for a list of test modules and their description.

# *Chapter 7:  Lessons Learned*

1) The language is always evolving and changing, and should never be considered "solid" until the end of the project. Something may seem like a good idea at language design time, but during low level design of the compiler it may seem like a less than good idea.
2) A testing suite is important. When the language design changes, it is good to have a suite of tests that can be run again to confirm incompatible changes weren't made.
3) Testing never covers every possible scenario. There were times, after testing one section, I would begin testing another which relies on the previous, and come up with a scenario for the first that wasn't originally though of. This begs the question how many variations weren't uncovered
4) Sections of the compiler which look easy are usually harder than sections that look hard.
5) Allow time for understanding of third party tools. ANTLR syntax (specifically getting the AST structure just right) took longer than initially planned, and using the newer version which lacked as much documentation as the older version made things trickier.

## *Appendix A: Examples*

## A.1 Hello World type example

```
module Foo;
function Main;
input:
  argLength as int,
  args as string list;
output:
  returnCode as int;
do;
  listEntry as string;
  listArgs as string list;
  listArg = listArg Union args;  /* Copy args to listArg */
  filter args remove null;     /* Remove nulls from list */
  foreach(listEntry in listArgs)
  Do;
    < listEntry | "\n";        /* print listEntry to standard out */
  end;
end;
```

## A.2 Function Calling example

```
module SomeModule;
myInput as int list(1,2,3,4,5,6,7,8,9,10);

/* Returns a list of ints > 6, and a list of ints <= 6 based on an
function processIt;
input:
  inputList as int list;      /* The list to separate */
output:
  list1 as int list;          /* > 6 list */
  list2 as int list;          /* <= 6 list */
  rc as int;                  /* return code */
Do;
  list1 = find ">6" in inputList;
  list2 = find "<=6" in inputList;
  rc = 0;
End;

Function Main;
input;
output:
  returnCode as int;
Do;
  listEntry as int;
  retCode as int;
  inputList as int list;
  listYes as int list;
  listNo as int list;
  inputList = inputList Union myInput;
  (listYes,listNo,retCode) = processIt(inputList);
  check(retCode is 0)        /* Make sure return code okay */
  Do;
    > "Allowed Args\n";
```

```
    foreach(listEntry in listYes)
    Do;
      > listEntry | "\n";
    End;
    > "Not Allowed\n";
    foreach(listEntry in listNo)
    Do;
      > listEntry | "\n";
    End;
  End;
End;
```

## *Appendix B:  Test Cases*

## B.1 testMod.lppl

This is the LPPL module described in the tutorial. It was also used to test the runtime output of the implementer.

```
module testMod;
  teachers as string list("Stephen","Susan","Lewis"),
  students as string list("Brian","Peter","Mark","Christine"),
  faculty as string list("Bob","Carla","Steve"),
  goodRc as int(0);

function main;
input;
output:
  rc as int;
do;
  out as string list,
  x as int;
  rc = goodRc;
  (rc,out) = getAll(teachers,students,faculty);
  check(rc in goodRc)
  do;
    foreach(x in out)
    do;
      >x;
    end;
  end;
  check(rc notin goodRc)
  do;
    > "BAD RC";
  end;
end;

function getAll;
input:
  list1 as string list,
  list2 as string list,
  list3 as string list;
output:
  rc as int,
  outList as string list;
do;
  tempOut as string list contains(list1,list2,list3);
  outList = tempOut;
  rc = goodRc;
end;
```

## B.2 test.lppl

This is a test which is correct in every way, and utilizes all parts of the language. This was used to compare the built Abstract Syntax Tree as well as runtime output of the interpreter.

```
/** No Errors, hits every part of language */
```

```
module test;
  a as iNt(14),
  b AS int list(28,52),
  c as char('a'),
  d as char list('a','b','c'),
  e as string("ABC"),
  f as string list("ABC","DEF","GHI");

/* No input, 1 output */
function func1;
input;
output:
  rc as int list;
do;
  rc = 4;
end;

/* 1 input, no output */
function func2;
input:
  in1 as int;
output;
do;
  in1 = in1;
end;

/* multi input,
   multi output */
function func3;
input:
  in1 as int,
  in2 as int list;
output:
  out1 as int,
  out2 as string list;
do;
  out1 = in1;
  out2 = in2;
end;

function main;
input;
output:
  rc as int;
do;
  g as int,
  h as int list,
  i as int list contains(b,me,func1()),
  j as string list;

  (g,j) = func3(g,h);
  g = func1() + h - j;
  g=4;
  g = g union G exclude g intersect G + g - g * g;
  g = g + 4;
  > g;
end;
```

## B.3 test2.lppl

This tests the syntax of the language. It is error ridden and contains many syntactical errors. Although not all errors are currently present, this was modified as needed to expose possibly syntax problems.

```
/**
   Syntax Errors
*/
module test;
  a as int,                    /* Error -- no constant def */
  f as string list("ABC","DEF","GHI");

function func1;
input:
 rc as int;                    /* Error -- should be comma, more output
*/
 i as int;
output;
do;                                 /* Error -- no statements */
end;

function main;
input:                              /* Error -- no output parameters
but colon */
output:
  rc as int;
  i as int;
do;
  g as int,                    /* Error -- last one, should be semicolon
*/
  g = k check p;         /* Error -- just makes no sense           */
end;
```

## B.4 test3.lppl

This tests the semantics of the language. It is filled with semantic errors throughout. Because the compiler reports one semantic error at a time, this was modified as needed to expose all possible semantic problems.

```
/**
   Semantic Errors
*/
module test;
  a as int(4),
  f as string list("ABC","DEF","GHI");

function func1;
input:
 rc as int,
 i as int;
output;
do;
```

```
 h = i;                  /* Error -- h not defined */
end;

function main;
input;
output:
  rc as int;
do;
  g as int,
  g as string;    /* Error -- duplicate definition */
  g = func1(g); /* Error -- wrong function signiture */
  rc = g;
end;
```

## *Appendix C:  ANTLR Language File*

The ANTLR grammar file takes care of lexical analysis, syntax analysis, and building the Abstract Syntax Tree. The AST built is based on the ANTLR libraries and will be converted into an LpplNode tree. The lpplLexer and lpplParser files generated are expected to be in package edu.lppl.antlr.

```
grammar lppl;

options {
  k = 2;
  backtrack=true;
  memoize=true;
  output=AST;
  ASTLabelType=CommonTree;
}

tokens {
  MODULE='module';
  FUNCTION='function';
  INPUT='input';
  OUTPUT='output';
  TYPE='as';
  LIST='list';
  CONTAINS='contains';
  ME = 'me';
  TYPE_INT='int';
  TYPE_CHAR='char';
  TYPE_STRING='string';
  DO='do';
  END='end';
  FOREACH='foreach';
  IN='in';
  NOTIN='notin';
  NULL='null';
  INTERSECT='intersect';
  UNION='union';
  EXCLUDE='exclude';
  CHECK = 'check';
  LPAREN='(';
  RPAREN=')';
  COLON=':';
  COMMA=',';
  SEMICOLON=';';
  ONEQUOTE='\'';
  TWOQUOTE='"';
  ASSIGN='=';
  CONCAT='|';
  PLUS='+';
  MINUS='-';
  TIMES='*';
  SYSOUT='>';
  NODE_M;   /* Module Node */
  NODE_CL;  /* Constants List Node */
  NODE_LL; /* List List Node */
  NODE_FL;  /* Function List Node */
```

```
  NODE_DL; /* Definition List Node */
  NODE_IL; /* Input List Node */
  NODE_OL; /* Output List Node */
  NODE_CB; /* Code Block Node */
  NODE_SL; /* Statement List Node */
  NODE_FC;  /* Function Call Node */
  NODE_BLK; /* Blank Node */
}

@lexer::header {
package edu.lppl.antlr;
}

@header {
package edu.lppl.antlr;
}

@members {
public Tree parse() throws RecognitionException{
     lpplParser.module_return r = this.module();
     return r.tree;
}
}

/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

ID         : (ALPHA)(ALPHA | DIGIT |'_')*;
CHAR_LITERAL : '\'' ( ESC | ~('\''|'\\') ) '\'';
STRING_LITERAL   : '"' ( ESC | ~('\\'|'"') )* '"';
NUMBER        : DIGIT DIGIT*;

WS      : (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;};
COMMENT    : '/*' ( options {greedy=false;} : . )* '*/'
{$channel=HIDDEN;}  ;

fragment ALPHA : 'a'..'z' | 'A'..'Z';
fragment DIGIT : '0'..'9';
fragment ESC   :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') ;

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

module       :   MODULE ID ';' constList? functionList EOF -> ^(NODE_M
ID constList? functionList);
constList :(constDef ',')* constDef ';'                    ->
^(NODE_CL constDef*);
constDef  :ID TYPE varType LIST '(' constDefList ')'       -> ^(ID
varType constDefList)
           |     ID TYPE varType '(' constName ')'              ->
^(ID varType constName);
constDefList :   (constName ',')* constName                       ->
^(NODE_LL constName*);
```

```
functionList :    (function)+
     -> ^(NODE_FL function*);
function  : FUNCTION ID ';' inputFunc outputFunc block  -> ^(ID
inputFunc outputFunc block);
inputFunc : INPUT ':' parmList
     -> ^(NODE_IL parmList)
           |   INPUT ';'
           -> ^(NODE_IL);
outputFunc : OUTPUT ':' parmList                                  ->
^(NODE_OL parmList)
           |     OUTPUT ';'
           -> ^(NODE_OL);
parmList  : (parmDef ','!)* parmDef ';';
parmDef   : ID TYPE varType LIST?                                 ->
^(ID varType LIST?);
varList   : (varDef ','!)* varDef ';';
varDef    : ID TYPE varType (LIST (CONTAINS '(' containmentList ')')?)?
-> ^(ID varType (LIST (containmentList)?)?);
varType   : TYPE_INT
           |     TYPE_STRING
           |     TYPE_CHAR;
containmentList : (containmentDef ',')* containmentDef     ->
^(NODE_DL containmentDef*);
containmentDef  : ME
           |     ID
           |     funcCallNoParms;
block        :   DO ';' varList? stmtList END ';'                 ->
^(NODE_CB varList? stmtList);
stmtList  : (stmt)+
           -> ^(NODE_SL stmt*);
stmt      : condStmt
           |     asignStmt;

condStmt  : FOREACH^ condStmt2
           |     CHECK^ condStmt2;
condStmt2 : '('! ID (IN | NOTIN) ID ')'! block;
asignStmt : ID ASSIGN^ expr ';'!
           |     '('! (recID ','!)* ID ')'! ASSIGN^ funcCall ';'!
           |     SYSOUT^ expr ';'!;
recID     : ID
        | /* nothing */
     -> NODE_BLK;
expr   :   arithlow
           |     idOrConstOrNo;

arithlow  : (arithmed) (CONCAT^ arithmed | UNION^ arithmed | EXCLUDE^
arithmed | INTERSECT^ arithmed)*;
arithmed  : (arithhigh) (PLUS^ arithhigh | MINUS^ arithhigh)*;
arithhigh : (arithbase) (TIMES^ arithbase)*;
arithbase : idOrConstOrFunc
           |     '('! expr ')'!;

funcCallNoParms : ID '(' ')'                                      ->
^(NODE_FC ID);
funcCall  : ID '(' funcParams? ')'                                ->
^(NODE_FC ID funcParams?);
```

```
funcParams:(expr ','!)* expr;

idOrConstOrNo : idOrConstOrFunc
           |     NULL;
idOrConstOrFunc :ID
           |     funcCall
           |     constName;
constName :STRING_LITERAL
           |     CHAR_LITERAL
           |     NUMBER;
```

## *Appendix D:  Compiler*

## D.1 Package edu.lppl

### D.1.1 Class edu.lppl.lppl

```
package edu.lppl;

import java.util.List;

import edu.lppl.build.ExecuteCode;
import edu.lppl.interfaces.ASTBuilder;
import edu.lppl.interfaces.TreeWalker;
import edu.lppl.tree.LpplNode;

/**
 * The main entry point to the LPPL Compiler
 * @author Brian
 *
 */
public class lppl {
	/**
	 * A list of LpplNode walkers built from the lppl.properties file
	 */
	private List g_walkers;

	/**
	 * The LpplNode tree builder, built from the lppl.properties file
	 */
	private ASTBuilder g_builder;

	/**
	 * Constructor for the LPPL Compiler.  Poplulates the walkers and
	 * AST Builder using the lppl.properties file
	 */
	public lppl() {
		g_walkers = LpplProperties.getWalkers();
		g_builder = LpplProperties.getASTBuilder();
	}

	/**
	 * Execute the compiler
	 *
	 * @param fileName The filename of the module to
compile/interperet
	 * @throws Exception In case anything happens
	 */
	public void execute(String fileName) throws Exception{
		LpplNode rootNode = g_builder.buildAST(fileName);
	/* Build the LpplNode Tree, should

				also do lexical and syntactical

				analyisis
*/
		if(rootNode != null) {
			/* Make sure module was

				syntactically okay, and the
```

```
                                    LpplNode tree was built       */
                        boolean err = false;
                        for(int x=0; x<g_walkers.size() && !err; x++) {
            /* Iterate through each walker   */
                              TreeWalker tw = (TreeWalker)g_walkers.get(x);
                              if((rootNode = tw.walk(rootNode)) == null) {
/* Walk the tree, continue only

                                    if no errors were raised      */
                                    System.out.println(tw.getErrorMessage());
                                    err = true;
                              }
                        }
                        if(!err) {
                              /* Make sure we are good to

                                    execute the module            */
                              new ExecuteCode().execute(rootNode);
                        }
                  }
            }

            /**
             * The main function
             * @param args Takes one argument, the file name to
compile/interperet
             * @throws Exception In case anything happens
             */
            public static void main(String[] args) throws Exception {
                  lppl comp = new lppl();
                  if(args.length != 1) {
                        System.out.println("Syntax: edu.lppl.lppl
<fileName>");
                        System.out.println("  Where <fileName> is the name of
the lppl module to compile."
                                          + "  Must end with .lppl");
                        return;
                  }
                  if(comp.g_walkers != null && comp.g_builder != null) {
                        comp.execute(args[0]);
                  }
            }
}
```

## D.1.2 Class edu.lppl.LpplProperties

```
package edu.lppl;

import java.util.ArrayList;
import java.util.List;
import java.util.PropertyResourceBundle;
import java.util.ResourceBundle;
```

```
import edu.lppl.interfaces.ASTBuilder;

/**
 * The Properties file facade.  Loads edu.lppl.lppl.properties file into memory and retrieves
 * its values.  Possible properties are defined as constants
 *
 * @author Brian
 *
 */
public class LpplProperties {
	/**
	 * The property name for the ASTBuilder class
	 */
	private static final String PROP_BUILDER = "lppl.astBuilder";

	/**
	 * The property name for the walkers
	 * This property is expecting a number, for the number of walkers.  Then it expects
to be
	 * that many walkers in the form of "lppl.walkers.x", where x is 1 through the
number
	 * defined by "lppl.walkers".  "lppl.walkers.x" expects the class name of a specific
walker
	 */
	private static final String PROP_WALKER = "lppl.walkers";

	/**
	 * The resource bundle to load the properties
	 */
	private static PropertyResourceBundle s_props =
		(PropertyResourceBundle)ResourceBundle.getBundle("edu.lppl.lppl");

	/**
	 * A list of the walkers for easy loading
	 */
	private static List s_walkers;

	/**
	 * The builder for easy loading
	 */
	private static ASTBuilder s_builder;

	/**
	 * Return the list of the walkers.  If this is the first time called, the properties file
	 * will be read to load the walkers into memory.  Every other time the list is already
	 * in memory
	 *
	 * @return A list of the walkers defined by the properties file
```

```
     */
    public static List getWalkers() {
        if(s_walkers == null) {
            String numWalkersStr = s_props.getString(PROP_WALKER);
            try {
                int numWalkers = Integer.parseInt(numWalkersStr); /* Get
the

                    number of walkers to create   */
                boolean errFound = false;
                s_walkers = new ArrayList();
                for(int x=1; x<=numWalkers; x++) {
                    String clazz = s_props.getString(PROP_WALKER +
"." + x);
                    try {

    s_walkers.add(Class.forName(clazz).newInstance()); /* Create new instance

                      of that class and add to list */
                    } catch(Exception e) {
                        System.out.println("Walker " + clazz + " not
found in classpath");
                        errFound = true;
                    }
                }
                if(errFound) {
                    return null;
                }
            } catch(Exception e) {
                System.out.println("Property " + PROP_WALKER +
                    " should be the number of walkers defined,
not \"" + numWalkersStr + "\"");
                return null;
            }

        }
        return s_walkers;
    }

    /**
     * Return the ASTBuilder.  If this is the first time called, the properties file
     * will be read to load the ASTBuilder into memory.  Every other time the list is
     * already in memory
     *
     * @return The ASTBuilder defined by the properties file
     */
    public static ASTBuilder getASTBuilder() {
        if(s_builder == null) {
```

```
                        String clazz = s_props.getString(PROP_BUILDER);
                        try {
                                s_builder =
(ASTBuilder)Class.forName(clazz).newInstance();
                        } catch(Exception e) {
                                System.out.println("ASTBuilder " + clazz + " not found in
classpath");
                                s_builder = null;
                                return null;
                        }
                }
                return s_builder;
        }
}
```

### D.1.3 Properties File edu.lppl.lppl.properties

```
#lppl.astBuilder
#The class name of the ASTBuilder to use.  This will perform
#syntax checking, and build an LpplNode tree
lppl.astBuilder=edu.lppl.antlr.ANTLRFacade

#lppl.walkers
#The number of walkers defined followed by a list of the class names
#of the walkers
lppl.walkers=3
lppl.walkers.1=edu.lppl.walkers.BuildSymbolTableWalker
lppl.walkers.2=edu.lppl.walkers.CheckSymbolsWalker
lppl.walkers.3=edu.lppl.walkers.RemoveVarDefsWalker
#lppl.walkers.4=edu.lppl.walkers.PrintTreeWalker
```

## D.2 Package edu.lppl.interfaces

### D.2.1 Interface edu.lppl.interfaces.ASTBuilder

```
package edu.lppl.interfaces;

import edu.lppl.tree.LpplNode;

/**
 * The interface for any implementation that wants to build the
LpplNode tree.
 * @author Brian
 *
 */
public interface ASTBuilder {

       /**
        * Build the LpplNode tree and return the root LpplNode.  Before
building the tree,
        * lexical and syntactical analysis should be performed.
        *
```

```
       * @param fileName The filename of the module to parse and build
the LpplNode tree for
       *
       * @return The root node of the LpplNode tree, or null if
analyisis and
       *         tree build was unsuccessfull
       */
      public LpplNode buildAST(String fileName);
}
```

## D.2.2 Interface edu.lppl.interfaces.TreeWalker

```
package edu.lppl.interfaces;

import edu.lppl.tree.LpplNode;

/**
 * The interface any Tree Walker must implement to walk the LpplNode
tree.
 * In order to be recognized, the walker must also be put in the
lppl.properties
 * file
 *
 * @author Brian
 *
 */
public interface TreeWalker {
      /**
       * Walk the LpplNode tree
       *
       * @param node The root node of the LpplNode tree
       *
       * @return The (possibly new) root node of the LpplNode tree, or
null if there was
       *         an error.  An error here is defined as catastrophic
such that no other
       *         walkers should be run and the compile will stop.
       */
      public LpplNode walk(LpplNode node);

      /**
       * When walk returns null, the error message which explains why
the walk was
       * a failure
       *
       * @return The error message to print to the console
       */
      public String getErrorMessage();
}
```

## D.3 Package edu.lppl.tree

### D.3.1 Class edu.lppl.tree.LpplNode

```
package edu.lppl.tree;

import java.util.ArrayList;
import java.util.List;

import edu.lppl.symbols.SymbolTable;

/**
 * The base class which makes up the LpplNode tree
 *
 * @author Brian
 *
 */
public class LpplNode implements LpplNodeTypes{
	/**
	 * The current scope's symbol table this LpplNode would "execute" in
	 */
	private SymbolTable g_symbolTable;

	/**
	 * The character position in the original module text file this node was at
	 */
	private int g_charPositionInLine;

	/**
	 * The line in the original module text file this node was at
	 */
	private int g_line;

	/**
	 * The type this node represents, should be one in LpplNodeTypes
	 */
	private int g_type;

	/**
	 * A list of all child LpplNodes
	 */
	private List g_children;

	/**
	 * The text this node represents
	 */
	protected String g_text;

	/**
```

```
 * The main constructor for the LpplNode
 * @param text The text ofthis node
 * @param type The type ofthis node, one of LpplNodeTypes
 * @param line The line in the original module text file this node resides
 * @param charPosInLine The character position in the original module text
 *        file tihs node resides
 */
public LpplNode(String text, int type, int line, int charPosInLine) {
        g_text = text;
        g_type = type;
        g_line = line;
        g_charPositionInLine = charPosInLine;
        g_children = new ArrayList();
}

/**
 * Retrieve the text of this node
 * @return This node's text
 */
public String getText() {
        return g_text;
}

/**
 * Retrieve the type of this node
 * @return This node's type
 */
public int getType() {
        return g_type;
}

/**
 * Retrieve the line this node resides on
 * @return This node's line
 */
public int getLine() {
        return g_line;
}

/**
 * Retrieve the character position this node resides around
 * @return This node's character position
 */
public int getCharPositionInLine() {
        return g_charPositionInLine;
}

/**
```

```
 * Retrieve the scope's symbol table this node belongs to
 * @return This node's scope's symbol table
 */
public SymbolTable getSymbolTable() {
	return g_symbolTable;
}

/**
 * Set the symbol table this node should use for scope
 * @param symbolTable The symbol table of the scope
 */
public void setSymbolTable(SymbolTable symbolTable) {
	g_symbolTable = symbolTable;
}

/**
 * Add node as child of this node
 * @param child The node to be a child
 */
public void addChild(LpplNode child) {
	g_children.add(child);
}

/**
 * Get a specific child of this node
 * @param x The child to get
 * @return The child node
 */
public LpplNode getChild(int x) {
	return (LpplNode)g_children.get(x);
}

/**
 * Get the number of children nodes this node has
 * @return The number of children nodes this node has
 */
public int getChildCount() {
	return g_children.size();
}

/**
 * Destroy and remove a child node.  Note this will render
 * the child node specified useless
 * @param index The child to destroy and remove
 */
public void removeChild(int index) {
	((LpplNode)g_children.get(index)).destroyNode();
	g_children.remove(index);
```

```
	}

	/**
	 * Destroy this node
	 *
	 */
	public void destroyNode() {
		g_symbolTable = null;
		g_children = null;
	}
}
```

## D.3.2 Class edu.lppl.tree.LpplNodeTypes

```
package edu.lppl.tree;

/**
 * The different node types
 *
 * @author Brian
 *
 */
public interface LpplNodeTypes {
	/**
	 * Module Node
	 */
	public static final int NODE_M = 1;

	/**
	 * Definition List Node
	 */
	public static final int NODE_DL = 2;

	/**
	 * Function List Node
	 */
	public static final int NODE_FL = 3;

	/**
	 * Statement List Node
	 */
	public static final int NODE_SL = 4;

	/**
	 * Output List Node
	 */
	public static final int NODE_OL = 5;

	/**
	 * Constants List Node
	 */
	public static final int NODE_CL = 6;

	/**
	 * Input List Node
```

```
 */
public static final int NODE_IL = 7;

/**
 * Control Block Node
 */
public static final int NODE_CB = 8;

/**
 * Identifier Node
 */
public static final int NODE_ID = 9;

/**
 * Output List Node
 */
public static final int NODE_OP = 10;

/**
 * List List Node
 */
public static final int NODE_LL = 11;

/**
 * Function Call Node
 */
public static final int NODE_FC = 12;

/**
 * Literal Node
 */
public static final int NODE_LIT = 13;

/**
 * Type Node
 */
public static final int NODE_TYPE = 14;

/**
 * Conditional Statement Node
 */
public static final int NODE_COND = 15;

/**
 * List Node -- The "list" keyword
 */
public static final int NODE_LIST = 16;

/**
 * SYSOUT Node -- The ">" keyword
 */
public static final int NODE_SYS = 17;

/**
 * Me Node -- The "me" keyword
 */
public static final int NODE_ME = 18;
```

```
	/**
	 * Null Node -- The "null" keyword
	 */
	public static final int NODE_NULL = 19;

	/**
	 * Blank Node
	 */
	public static final int NODE_BLK = 20;
}
```

### D.3.3 Class edu.lppl.tree.LpplCondNode

```
package edu.lppl.tree;

/**
 * LpplNode which represents a condition statment
 *
 * @author Brian
 *
 */
public class LpplCondNode extends LpplNode{
	/**
	 * The type is a foreach statment
	 */
	public static final int COND_FOREACH = 1;

	/**
	 * The type is a check statement
	 */
	public static final int COND_CHECK   = 2;

	/**
	 * Whether we are checking if is in list or is not in list
	 */
	private boolean g_isIn;

	/**
	 * The conditional type, should be either COND_FOREACH or
COND_CHECK
	 */
	private int g_condType;

	/**
	 * The main constructor for the LpplCondNode
	 * @param text The text of this node
	 * @param type The type of this node, one of LpplNodeTypes
	 * @param line The line in the original module text file this
node resides
	 * @param charPosInLine The character position in the original
module text
	 *        file tihs node resides
	 * @param condType The condition statement type
	 * @param isIn Are we checking for inclusion
	 */
```

```
	public LpplCondNode(String text, int type, int line, int
charPosInLine, int condType, boolean isIn) {
		super(text,type,line,charPosInLine);
		g_isIn = isIn;
		g_condType = condType;
	}

	/**
	 * Retrieve whether this node is an inclusive or excludive check
	 * @return true if this node is an inclusive check, false
otherwise
	 */
	public boolean isIn() {
		return g_isIn;
	}

	/**
	 * Retrieve the type of this conditional statement
	 * @return the type of this conditional statement
	 */
	public int getConditionType() {
		return g_condType;
	}
}
```

## D.3.4 Class edu.lppl.tree.LpplLitNode

```
package edu.lppl.tree;

/**
 * LpplNode which represents a literal token
 *
 * @author Brian
 *
 */
public class LpplLitNode extends LpplNode {
	/**
	 * The literal is a char
	 */
	public static final int LIT_CHAR   = 1;

	/**
	 * The literal is an int
	 */
	public static final int LIT_INT    = 2;

	/**
	 * The literal is a string
	 */
	public static final int LIT_STRING = 3;

	/**
	 * The literal type, must be one of LIT_CHAR, LIT_INT, or
LIT_STRING
	 */
	private int g_litType;
```

```
	/**
	 * The main constructor for the LpplLitNode
	 * @param text The text of this node
	 * @param type The type of this node, one of LpplNodeTypes
	 * @param line The line in the original module text file this
node resides
	 * @param charPosInLine The character position in the original
module text
	 *        file tihs node resides
	 * @param litType The condition statement type
	 */
	public LpplLitNode(String text, int type, int line, int
charPosInLine, int litType) {
		super(text,type,line,charPosInLine);
		g_litType = litType;

		if(g_litType == LIT_CHAR || g_litType == LIT_STRING) {
			g_text = g_text.substring(1,g_text.length()-1);
		}
	}

	/**
	 * Get the literal type of this node
	 * @return The literal type of this node
	 */
	public int getLitType() {
		return g_litType;
	}
}
```

## D.3.5 Class edu.lppl.tree.LpplOpNode

```
package edu.lppl.tree;

/**
 * LpplNode which represents an operation
 *
 * @author Brian
 *
 */
public class LpplOpNode extends LpplNode {
	/**
	 * The operation is plus -- "+"
	 */
	public static final int OP_PLUS    = 1;

	/**
	 * The operation is concatenation -- "|"
	 */
	public static final int OP_CONCAT  = 2;

	/**
	 * The operation is union -- "union"
	 */
	public static final int OP_UNION   = 3;
```

```
	/**
	 * The operation is intersection -- "intersect"
	 */
	public static final int OP_INT     = 4;

	/**
	 * The operation is minus -- "-"
	 */
	public static final int OP_MINUS   = 5;

	/**
	 * The operation is times -- "*"
	 */
	public static final int OP_TIMES   = 6;

	/**
	 * The operation is exclusion -- "exclude"
	 */
	public static final int OP_EXCLUDE = 7;

	/**
	 * The operation is assignment -- "="
	 */
	public static final int OP_ASSIGN  = 8;

	/**
	 * The operation type of this node
	 */
	private int g_op;

	/**
	 * The main constructor for the LpplOpNode
	 * @param text The text of this node
	 * @param type The type of this node, one of LpplNodeTypes
	 * @param line The line in the original module text file this
node resides
	 * @param charPosInLine The character position in the original
module text
	 *        file tihs node resides
	 * @param op The operation type
	 */
	public LpplOpNode(String text, int type, int line, int
charPosInLine, int op) {
		super(text,type,line,charPosInLine);
		g_op = op;
	}

	/**
	 * Get the operation type of this node
	 * @return
	 */
	public int getOperation() {
		return g_op;
	}
}
```

## D.3.6 Class edu.lppl.tree.LpplTypeNode

```
package edu.lppl.tree;

/**
 * LpplNode which represents the type of a symbol
 *
 * @author Brian
 *
 */
public class LpplTypeNode extends LpplNode{
	/**
	 * The type is character -- "char"
	 */
	public static final int TYPE_CHAR   = 1;

	/**
	 * The type is integer -- "int"
	 */
	public static final int TYPE_INT    = 2;

	/**
	 * The type is string -- "string"
	 */
	public static final int TYPE_STRING = 3;

	/**
	 * The type type of this node
	 */
	private int g_tType;

	/**
	 * Whether it is the list ofrm of this type
	 */
	private boolean g_isList;

	/**
	 * The main constructor for the LpplTypeNode
	 * @param text The text of this node
	 * @param type The type of this node, one of LpplNodeTypes
	 * @param line The line in the original module text file this
node resides
	 * @param charPosInLine The character position in the original
module text
	 *        file tihs node resides
	 * @param tType The type type
	 * @param isList Is this the list form of the type
	 */
	public LpplTypeNode(String text, int type, int line, int
charPosInLine, int tType, boolean isList) {
		super(text,type,line,charPosInLine);
		g_tType = tType;
		g_isList = isList;
	}
```

```
	/**
	 * Get the type type of this node
	 * @return The type type
	 */
	public int getTypeType() {
		return g_tType;
	}

	/**
	 * Get whether this is the list form of the type
	 * @return true if this is the list form of the type, false
otherwise
	 */
	public boolean isList() {
		return g_isList;
	}
}
```

## D.4 Package edu.lppl.antlr

### D.4.1 Class edu.lppl.antlr.ANTLRFacade

```
package edu.lppl.antlr;

import java.io.IOException;

import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.tree.Tree;

import edu.lppl.interfaces.ASTBuilder;
import edu.lppl.tree.LpplCondNode;
import edu.lppl.tree.LpplLitNode;
import edu.lppl.tree.LpplNode;
import edu.lppl.tree.LpplOpNode;
import edu.lppl.tree.LpplTypeNode;

/**
 * The facade to ANTLR from LPPL.
 * Convert the ANTLR AST into the LPPL AST.  This could have been done in
 * the ANTLR language file, but I found it to make the language look more complicated.
 * Once converted, there should be no dependencies on any ANTLR libraries.
 *
 * @author Brian
 *
 */
public class ANTLRFacade implements ASTBuilder{

	/**
```

```
	 * The ANTLR version of ASTBuilder, see ASTBuilder.buildAST for more
information
	 */
	public LpplNode buildAST(String fileName) {
		try {
			lpplLexer lex = new lpplLexer(new
CaseInsensitiveFileStream(fileName));
			CommonTokenStream tokens = new CommonTokenStream(lex);

			lpplParser parser = new lpplParser(tokens);
			//make sure no problems before continueing!!!!
			Tree myTree = parser.parse();
			return convert(null, null, myTree);

		} catch (IOException ioe) {
			ioe.printStackTrace();
		}
		catch (RecognitionException e) {
			e.printStackTrace();
		}
		return null;
	}

	/**
	 * Convert the current ANTLR tree node into an LpplNode or some subclass,
	 * depending on the type.  The recursively call itself for each child
	 * node.
	 *
	 * @param lpplParent The LpplNode's parent
	 * @param parent The ANTLR's parent
	 * @param tree The ANTLR tree node to convert
	 * @return The LpplNode with all children already converted
	 */
	private LpplNode convert(LpplNode lpplParent, Tree parent, Tree tree) {
		LpplNode lpplNode = getLpplNode(parent, tree);				/*
Convert current to LpplNode	*/
		if(lpplNode != null) {
		/* Only process children if successfull */
		//	lpplNode.setParent(lpplParent);
		/* Set the current nodes parent to parm */
			for(int x=0; x<tree.getChildCount(); x++) {			/*
Iterate through children	*/
				LpplNode lpplChild =
convert(lpplNode,tree,tree.getChild(x));
				if(lpplChild != null) {
		/* Only append as a Lppl child if

					something was converted	*/
```

```
                                        lpplNode.addChild(lpplChild);
                                }
                        }
                }
                return lpplNode;
        }

        /**
         * Convert ANTLR node type to LPPL node type
         *
         * @param parent The parent of the node to convert.  This is needed to get siblings
of current node
         * @param tree The ANTLR node to convert
         * @return The LPPL node
         */
        private LpplNode getLpplNode(Tree parent, Tree tree) {
                LpplNode lpplNode = null;
                switch(tree.getType()) {
                        case lpplParser.NODE_M:
                                lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_M,tree.getLine(),tree.getCharPositionInLine());
                                break;
                        case lpplParser.NODE_DL:
                                lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_DL,tree.getLine(),tree.getCharPositionInLine()
);
                                break;
                        case lpplParser.NODE_FL:
                                lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_FL,tree.getLine(),tree.getCharPositionInLine())
;
                                break;
                        case lpplParser.NODE_SL:
                                lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_SL,tree.getLine(),tree.getCharPositionInLine())
;
                                break;
                        case lpplParser.NODE_OL:
                                lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_OL,tree.getLine(),tree.getCharPositionInLine()
);
                                break;
                        case lpplParser.NODE_CL:
                                lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_CL,tree.getLine(),tree.getCharPositionInLine()
);
                                break;
                        case lpplParser.NODE_IL:
```

```
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_IL,tree.getLine(),tree.getCharPositionInLine());
                    break;
                case lpplParser.NODE_LL:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_LL,tree.getLine(),tree.getCharPositionInLine())
;
                    break;
                case lpplParser.NODE_CB:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_CB,tree.getLine(),tree.getCharPositionInLine()
);
                    break;
                case lpplParser.NODE_FC:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_FC,tree.getLine(),tree.getCharPositionInLine()
);
                    break;
                case lpplParser.ME:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_ME,tree.getLine(),tree.getCharPositionInLine()
);
                    break;
                case lpplParser.SYSOUT:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_SYS,tree.getLine(),tree.getCharPositionInLine(
));
                    break;
                case lpplParser.NULL:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_NULL,tree.getLine(),tree.getCharPositionInLi
ne());
                    break;
                case lpplParser.ID:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_ID,tree.getLine(),tree.getCharPositionInLine())
;
                    break;
                case lpplParser.NODE_BLK:
                    lpplNode = new
LpplNode(tree.getText(),LpplNode.NODE_BLK,tree.getLine(),tree.getCharPositionInLine
());
                    break;
                case lpplParser.STRING_LITERAL:
                    lpplNode = new
LpplLitNode(tree.getText(),LpplNode.NODE_LIT,tree.getLine(),tree.getCharPositionInLi
ne(),LpplLitNode.LIT_STRING);
                    break;
```

```
                        case lpplParser.CHAR_LITERAL:
                                lpplNode = new
LpplLitNode(tree.getText(),LpplNode.NODE_LIT,tree.getLine(),tree.getCharPositionInLi
ne(),LpplLitNode.LIT_CHAR);
                                break;
                        case lpplParser.NUMBER:
                                lpplNode = new
LpplLitNode(tree.getText(),LpplNode.NODE_LIT,tree.getLine(),tree.getCharPositionInLi
ne(),LpplLitNode.LIT_INT);
                                break;
                        case lpplParser.TYPE_CHAR:
                                boolean isList = parent.getChild(1) != null &&
(parent.getChild(1).getType() ==lpplParser.LIST || parent.getChild(1).getType() ==
lpplParser.NODE_LL);
                                lpplNode = new
LpplTypeNode(tree.getText(),LpplNode.NODE_TYPE,treegetLine(),tree.getCharPositionI
nLine(),LpplTypeNode.TYPE_CHAR,isList);
                                break;
                        case lpplParser.TYPE_STRING:
                                isList = parent.getChild(1) != null &&
(parent.getChild(1).getType() ==lpplParser.LIST || parent.getChild(1).getType() ==
lpplParser.NODE_LL);
                                lpplNode = new
LpplTypeNode(tree.getText(),LpplNode.NODE_TYPE,treegetLine(),tree.getCharPositionI
nLine(),LpplTypeNode.TYPE_STRING,isList);
                                break;
                        case lpplParser.TYPE_INT:
                                isList = parent.getChild(1) != null &&
(parent.getChild(1).getType() ==lpplParser.LIST || parent.getChild(1).getType() ==
lpplParser.NODE_LL);
                                lpplNode = new
LpplTypeNode(tree.getText(),LpplNode.NODE_TYPE,treegetLine(),tree.getCharPositionI
nLine(),LpplTypeNode.TYPE_INT,isList);
                                break;
                        case lpplParser.FOREACH:
                                boolean isIn = tree.getChild(1).getType() == lpplParser.IN;
                                lpplNode = new
LpplCondNode(tree.getText(),LpplNode.NODE_COND,tree.getLine(),tree.getCharPositio
nInLine(),LpplCondNode.COND_FOREACH,isIn);
                                break;
                        case lpplParser.CHECK:
                                isIn = tree.getChild(1).getType() == lpplParser.IN;
                                lpplNode = new
LpplCondNode(tree.getText(),LpplNode.NODE_COND,tree.getLine(),tree.getCharPositio
nInLine(),LpplCondNode.COND_CHECK,isIn);
                                break;
                        case lpplParser.ASSIGN:
```

```
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_ASSIGN);
                                break;
                        case lpplParser.PLUS:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_PLUS);
                                break;
                        case lpplParser.EXCLUDE:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_EXCLUDE);
                                break;
                        case lpplParser.CONCAT:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_CONCAT);
                                break;
                        case lpplParser.UNION:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_UNION);
                                break;
                        case lpplParser.INTERSECT:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_INT);
                                break;
                        case lpplParser.MINUS:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_MINUS);
                                break;
                        case lpplParser.TIMES:
                                lpplNode = new
LpplOpNode(tree.getText(),LpplNode.NODE_OP,tree.getLine(),tree.getCharPositionInLin
e(),LpplOpNode.OP_TIMES);
                                break;
                }
                return lpplNode;
        }
}
```

## D.4.2 Class edu.lppl.antlr.CaseInsensitiveFileStream

```
package edu.lppl.antlr;

import java.io.IOException;
```

```
import org.antlr.runtime.ANTLRFileStream;

/**
 * The file stream used by ANTLR which makes tokens case insensitive
 *
 * @author Brian
 *
 */
public class CaseInsensitiveFileStream extends ANTLRFileStream {

        public CaseInsensitiveFileStream(String filename) throws IOException {
                super(filename);
        }

   public int LA(int i) {
      if (i == 0) {
         return 0;
      }
      if (i < 0) {
         i++;
      }
      if (((p + i) - 1) >= n) {
         return -1;
      }
      return Character.toLowerCase(data[(p + i) - 1]);
   }
}
```

### D.4.3 Class edu.lppl.antlr.lpplLexer

This class is generated by the ANTLR language file.

### D.4.4 Class edu.lppl.antlr.lpplParser

This class is generated by the ANTLR language file.

## D.5 Package edu.lppl.symbols

### D.5.1 Class edu.lppl.symbols.Symbol

```
package edu.lppl.symbols;

/**
 * Base class for all symbols in the symbol table
 *
 * @author Brian
 *
 */
public class Symbol {
```

```
	/**
	 * The name of this symbol
	 */
	private String g_name;

	/**
	 * Main constructor for Symbol
	 * @param name The name of this symbol
	 */
	public Symbol(String name) {
		g_name = name;
	}

	/**
	 * Retrieve the name of this symbol
	 * @return The name of this symbol
	 */
	public String getName() {
		return g_name;
	}
}
```

## D.5.2 Class edu.lppl.symbols.ConstSymbol

```
package edu.lppl.symbols;

import java.util.ArrayList;
import java.util.List;

import edu.lppl.build.ExecuteCode;
import edu.lppl.tree.LpplNode;

/**
 * A symbol in the symbol table which represents a variable that is constant
 *
 * @author Brian
 *
 */
public class ConstSymbol extends Symbol {
	/**
	 * The variable type
	 */
	private SymbolType g_type;

	/**
	 * The values this symbol has, "the list"
	 */
	protected List g_values;

	/**
	 * Main Constructor for a ConstSymbol
```

```
	 * @param name The name of the variable
	 * @param type The type of the variable
	 */
	public ConstSymbol(String name, SymbolType type) {
		super(name);
		g_type = type;
		g_values = new ArrayList();
	}

	/**
	 * Add a value to variable, which will never be modified or remove
	 * @param val
	 */
	public void addValue(String val) {
		g_values.add(val);
	}

	/**
	 * Retrieve this variables type
	 * @return The type of this variable
	 */
	public SymbolType getType() {
		return g_type;
	}

	/**
	 * Get the value of of this variable
	 * @param ex The execute code instance, which will be used for containment lists
that contain
	 *		a function definition to execute that function.
	 * @return The value of this variable
	 */
	public List getValue(ExecuteCode ex) {
		if(!g_type.isContainingList()) {
			return g_values;
		}
		List l = new ArrayList();
		List containees = g_type.getContainees();
		for(int x=0; x<containees.size(); x++) {
			if(g_type.getMeIndex() == x) {
				l.addAll(g_values);
			}
			if(containees.get(x) instanceof ConstSymbol) {
				ConstSymbol cs = (ConstSymbol)containees.get(x);
				l.addAll(cs.getValue(ex));
			}
			else {
				/* Instance of FuncSymbol	*/
```

```
                                FuncSymbol fs = (FuncSymbol)containees.get(x);
                                LpplNode funcNode = ex.findFunc(fs.getName());  /* Find
the function          */
                                ex.executeCodeBlock(funcNode);
                /* Execute it                */
                                l.addAll(fs.getOutParm(0).getValue(ex));                /* Add
whatever was returned    */
                        }
                }
                return l;
        }
}
```

### D.5.3 Class edu.lppl.symbols.VarSymbol

```
package edu.lppl.symbols;

import java.util.ArrayList;
import java.util.List;

/**
 * A symbol in the symbol table which represents a variable that is dynamic
 *
 * @author Brian
 *
 */
public class VarSymbol extends ConstSymbol{

        /**
         * Main Constructor for a VarSymbol
         * @param name The name of the variable
         * @param type The type ofthe symbol
         */
        public VarSymbol(String name, SymbolType type) {
                super(name,type);
        }

        /**
         * Set the value of this variable
         * @param val The list value to set this variable to
         */
        public void setValue(List val) {
                g_values = val;
                if(!getType().isList()) {
                        while(g_values.size()> 1) {
                                g_values.remove(1);
                        }
                }
        }
```

```
	/**
	 * Set the single value of this variable
	 * @param val The single value to set this variable to
	 */
	public void setValue(String val) {
		List l = new ArrayList();
		l.add(val);
		setValue(l);
	}

	/**
	 * Initialize this symbol
	 */
	public void init() {
		g_values = new ArrayList();
	}
}
```

### D.5.4 Class edu.lppl.symbols.FuncSymbol

```
package edu.lppl.symbols;

import java.util.ArrayList;
import java.util.List;

/**
 * A Symbol in the Symbol Table which represents a function signiture
 *
 * @author Brian
 *
 */
public class FuncSymbol extends Symbol{
	/**
	 * A list of the return Symbols, of type VarSymbol
	 */
	private List g_inParms;

	/**
	 * A list of the parameter Symbols, of type VarSymbol
	 */
	private List g_outParms;

	/**
	 * Main constructor for a FuncSymbol
	 * @param name The name of the symbol
	 */
	public FuncSymbol(String name) {
		super(name);
```

```
        g_inParms = new ArrayList();
        g_outParms = new ArrayList();
}

/**
 * Add a new output parameter symbol, positional by the order added
 * @param symbol The symbol that is an output parameter symbol
 */
public void addOutParm(VarSymbol symbol) {
        g_outParms.add(symbol);
}

/**
 * Add a new input parameter symbol, positional by the order added
 * @param symbol The symbol that is an input parameter symbol
 */
public void addInParm(VarSymbol symbol) {
        g_inParms.add(symbol);
}

/**
 * Retrieve the nth output parameter symbol
 * @param pos The output parameter's position to get
 * @return The VarSymbol in that position
 */
public VarSymbol getOutParm(int pos) {
        return (VarSymbol)g_outParms.get(pos);
}

/**
 * Retrieve the nth Parameter Symbol
 * @param pos The input parameter's position to get
 * @return The VarSymbol in that position
 */
public VarSymbol getInParm(int pos) {
        return (VarSymbol)g_inParms.get(pos);
}

/**
 * Retreive the number of input parameters
 * @return
 */
public int getNumInParms() {
        return g_inParms.size();
}

public int getNumOutParms() {
        return g_outParms.size();
```

```
	}

	public String getFunctionSignature() {
		String msg = "";
		if(g_outParms.size() > 0) {
			msg += "(";
			for(int x=0; x<g_outParms.size(); x++) {
				msg+=
((VarSymbol)g_outParms.get(x)).getType().getTypeString() + ",";
			}
			msg = msg.substring(0,msg.length()-1) + ")";
		}
		msg += super.getName() + "(";
		for(int x=0; x<g_inParms.size(); x++) {
			msg+= ((VarSymbol)g_inParms.get(x)).getType().getTypeString() +
",";
		}
		if(g_inParms.size() > 0) {
			msg = msg.substring(0,msg.length()-1);
		}
		msg += ")";

		return msg;
	}
}
```

### D.5.5 Class edu.lppl.symbols.SymbolType

```
package edu.lppl.symbols;

import java.util.ArrayList;
import java.util.List;

import edu.lppl.tree.LpplTypeNode;

/**
 * The type of a ConstSymbol or VarSymbol
 *
 * @author Brian
 *
 */
public class SymbolType {
	/**
	 * The global type identifier, should be one of LpplTypeNode.TYPE_XXXX
	 */
	private int g_type;

	/**
	 * Whether this symbol is defined as a list or not
```

```
	 */
	private boolean g_isList;

	/**
	 * The containment symbols of this list
	 */
	private List g_contains;

	/**
	 * The index in the containment list exclusive members of the list should be
	 */
	private int g_meIndex;

	/**
	 * The main constructor for SymbolType
	 * @param type Should be one of LpplTypeNode.TYPE_STRING,
LpplTypeNode.TYPE_CHAR,
	 *		or LpplTypeNode.TYPE_INT
	 * @param isList Whether or not this symbol is defined as a list
	 * @param isContainingList Whether or not this symbol has a containment
definintion
	 */
	public SymbolType(int type, boolean isList, boolean isContainingList) {
		g_type = type;
		g_isList = isList;
		if(isContainingList) {
			g_contains = new ArrayList();
		}
	}

	/**
	 * Add a symbol which is contained within this list, positional by order added
	 * @param s The symbol this list symbol contains
	 */
	public void addContainedSymbol(Symbol s) {
		g_contains.add(s);
	}

	/**
	 * Add me to the current containement order
	 *
	 */
	public void addMe() {
		g_meIndex = g_contains.size();
	}

	/**
	 * Determine if this symbol was defined as a list
```

```
     * @return true if this symbol is a list, false otherwise
     */
    public boolean isList() {
            return g_isList;
    }

    /**
     * Determine if this symbol was defined as a containment list
     * @return true if this symbol is a containment list, false otherwise
     */
    public boolean isContainingList() {
            return g_contains != null;
    }

    /**
     * Retrieve the list of the contained symbols within this symbol
     * @return A list of the contained symbols
     */
    public List getContainees() {
            return g_contains;
    }

    /**
     * Retrieve where in the containment list this symbol's exclusive values should be
     * @return
     */
    public int getMeIndex() {
            return g_meIndex;
    }

    /**
     * Get the string representation of the type of this symbol
     * @return
     */
    public String getTypeString() {
            String typeStr = "";
            switch(g_type) {
            case LpplTypeNode.TYPE_CHAR:
                    typeStr = "char";
                    break;
            case LpplTypeNode.TYPE_INT:
                    typeStr = "int";
                    break;
            case LpplTypeNode.TYPE_STRING:
                    typeStr = "string";
                    break;
            }
            if(g_isList) {
```

```
                typeStr += " list";
            }
            return typeStr;
        }
}
```

### D.5.6 Class edu.lppl.symbols.SymbolTable

```
package edu.lppl.symbols;

import java.util.HashMap;
import java.util.Iterator;

/**
 * A table of symbols that exist within a current scope
 *
 * @author Brian
 *
 */
public class SymbolTable {
        /**
         * A map of the symbols defined in the current scope
         */
        private HashMap g_symbols;

        /**
         * A link to the previous scope's symbol table
         */
        private SymbolTable g_parentTable;

        /**
         * The main constructor for the SymbolTable, defines the first scope
         * of a module
         */
        public SymbolTable() {
            this(null);
        }

        /**
         * Internal constructor used to define a new scope's symbol table
         * @param parentTable The previous scope's symbol table
         */
        private SymbolTable(SymbolTable parentTable) {
            g_symbols = new HashMap();
            g_parentTable = parentTable;
        }

        /**
         * Retreive the Symbol identified by the symbol name.  This will
```

```
     * first check its current scope's symbols and, if not found, will
     * delegate the request to the previous scopes symbol table
     * @param symbol The symbol name to retrieve
     * @return The Symbol object representing the symbol, or null
     *         if it isn't found in this scope or any previous scope
     */
    public Symbol getSymbol(String symbol) {
            String key = symbol.toUpperCase();
            if(g_symbols.containsKey(key)) {
                    return (Symbol)g_symbols.get(key);
            }
            else if(g_parentTable != null){
                    return g_parentTable.getSymbol(symbol);
            }
            return null;
    }

    /**
     * Add a symbol definition to the current scope
     * @param symbol The symbol
     * @return true if the symbol was added, false if
     *         this symbol is already defined in this
     *         scope
     */
    public boolean addSymbol(Symbol symbol) {
            String key = symbol.getName().toUpperCase();
            if(g_symbols.containsKey(key)) {
                    return false;
            }
            g_symbols.put(key, symbol);
            return true;
    }

    /**
     * Create new child scope's symbol table and return it
     *
     * @return The child scope's symbol table
     */
    public SymbolTable openScope() {
            SymbolTable st = new SymbolTable(this);
            return st;
    }

    /**
     * Initialize the symbols in the current scope.  This will
     * effectively zero all variables defined in this scope
     */
    public void initializeSymbolTable() {
```

```
                Iterator itr = g_symbols.keySet().iterator();
                while(itr.hasNext()) {
                        Symbol sym = (Symbol)g_symbols.get(itr.next());
                        if(sym instanceof VarSymbol) {
                                ((VarSymbol)sym).init();
                        }
                }
        }

        /**
         * Leave the current scope, and return to the previous scope
         * @return The previous scope's symbol table
         */
        public SymbolTable closeScope() {
                return g_parentTable;
        }
}
```

## D.6 Package edu.lppl.walkers

### D.6.1 Class edu.lppl.walkers.BuildSymbolTableWalker

```
package edu.lppl.walkers;

import edu.lppl.interfaces.TreeWalker;
import edu.lppl.symbols.ConstSymbol;
import edu.lppl.symbols.FuncSymbol;
import edu.lppl.symbols.Symbol;
import edu.lppl.symbols.SymbolTable;
import edu.lppl.symbols.SymbolType;
import edu.lppl.symbols.VarSymbol;
import edu.lppl.tree.LpplNode;
import edu.lppl.tree.LpplTypeNode;

/**
 * This walker is responsible for building the symbol tables, creating and
 * destroying scopes at the correct nodes in the tree.  It also checks
 * and reports on the following errors:
 *
 * 1) Duplicate symbols defined within the same scope
 * 2) Containment symbols not defined within the current scope (and parent scopes)
 *
 * This should be the first walker to traverse the LpplNode tree
 *
 * @author Brian
 *
 */
public class BuildSymbolTableWalker implements TreeWalker{
```

```
	/**
	 * The current scope's symbol table
	 */
	public SymbolTable g_symbolTable;

	/**
	 * Was an error found while building the symbol tables
	 */
	public boolean g_errFound;

	/**
	 * The error message that should printed out on error
	 */
	public String g_errMsg;

	/**
	 * Generic Constructor, initializes the first symbol table
	 *
	 */
	public BuildSymbolTableWalker() {
		g_symbolTable = new SymbolTable();		/* Create the initial
symbol table */
		g_errFound = false;
	}

	/**
	 * Get the error message that occurred while building the symbol tables
	 * @return The error message
	 */
	public String getErrorMessage() {
		return g_errMsg;
	}

	/**
	 * Walk the LpplNode tree
	 * @parm node The root LpplNode
	 * @return The root LpplNode
	 */
	public LpplNode walk(LpplNode node) {
		internalWalk(node);
		return (g_errFound ? null : node);
	}

	/**
	 * An internal walk of the tree which gets called recursively
	 * @param node The current LpplNode to process
	 */
```

```
private void internalWalk(LpplNode node) {
        if(LpplNode.NODE_CB == node.getType()) {
                g_symbolTable = g_symbolTable.openScope();
/* Create new scope and return  */
        }
        node.setSymbolTable(g_symbolTable);
        /* Set the nodes symbol table

                                        to current                */
        switch(node.getType()) {
                case LpplNode.NODE_M:
                        processModule(node);
                        break;
                case LpplNode.NODE_CB:
        /* Start of DO; END; group      */
                        processCodeBlock(node);
                        break;
                case LpplNode.NODE_CL:
                /* Process Constants of modules */
                        processConstants(node);
        }
        for(int x=0; x<node.getChildCount();x++) {
                if(node.getType() == LpplNode.NODE_FL) {
                        processFunction(node.getChild(x));
                }
                else {
                        internalWalk(node.getChild(x));
                }
        }
        if(LpplNode.NODE_CB == node.getType()) {
                g_symbolTable = g_symbolTable.closeScope();
        }
}

/**
 * Process the module node, adds the module symbol to the current symbol table
 * @param node The LpplNode of type NODE_M
 */
private void processModule(LpplNode node) {
        addSymbol(node.getChild(0),new Symbol(node.getChild(0).getText()));
}

/**
 * Process a function definition, and add the new FuncSymbol to the current
 * scope's symbol table.  For function definitions, the current scope should
 * be the "oldest" scope.
 * @param node The LpplNode of a function definition
 */
```

```
        private void processFunction(LpplNode node) {
                FuncSymbol fs = new FuncSymbol(node.getText());          /* Initialize
funcsymbol      */
                g_symbolTable = g_symbolTable.openScope();
                for(int y=0; y<2; y++) {
                /* process input and output

                                        list parms                */
                        LpplNode parmList = node.getChild(y);
                        for(int z=0; z<parmList.getChildCount(); z++) {      /* Go through
each parameter    */
                                LpplNode parmId = parmList.getChild(z);
                                LpplTypeNode parmIdType =
(LpplTypeNode)parmId.getChild(0);
                                SymbolType type = new
SymbolType(parmIdType.getTypeType(),parmIdType.isList(),false);
                                VarSymbol vs = new VarSymbol(parmId.getText(),type);
                                if(y==0) {
                                        fs.addInParm(vs);
                                }
                                else {
                                        fs.addOutParm(vs);
                                }
                                addSymbol(parmId,vs);
                        }
                }
                node.setSymbolTable(g_symbolTable);
                /* Set the nodes symbol table to

                                        current
        */
                for(int x=0; x<node.getChildCount();x++) {
                        internalWalk(node.getChild(x));
                }
                g_symbolTable = g_symbolTable.closeScope();
                addSymbol(node,fs);
                        /* add function to symbol table  */
        }

        /**
         * Process a new code block (DO END block).  At the start of every new code
block is a new
         * scope, which should have already been entered before getting here.  This is
responsible
         * for finding any newly defined variables for this block and adding them to the
scope
         * @param node The code block LpplNode
         */
```

```
	private void processCodeBlock(LpplNode node) {
		int numVars = node.getChildCount() - 1;			/* The last node is the NODE_SL,

					everyhing else is var def	*/
		for(int x=0; x<numVars; x++) {
			LpplNode varId = node.getChild(x);
			LpplTypeNode varIdType = (LpplTypeNode)varId.getChild(0);
			SymbolType varType = new SymbolType(varIdType.getTypeType(),varIdType.isList(),varId.getChildCount()==2);
			if(varType.isContainingList()) {
				LpplNode containeeList = varId.getChild(1);
				for(int y=0; y<containeeList.getChildCount(); y++) {
					if(containeeList.getChild(y).getType() == LpplNode.NODE_ME) {
						varType.addMe();
					}
					else {
						Symbol containeeSym = null;
						if(containeeList.getChild(y).getType() == LpplNode.NODE_FC) {
							containeeSym = getSymbol(containeeList.getChild(y).getChild(0).getText());
						}
						else {
							containeeSym = getSymbol(containeeList.getChild(y).getText());
						}
						if(containeeSym == null) {
							g_errFound = true;
							g_errMsg = "Containing List " + varId.getText() + " contains unknown variable at " + varId.getLine() + ":" + varId.getCharPositionInLine();
						}
						else {

	varType.addContainedSymbol(containeeSym);
						}
					}
				}
			}
			addSymbol(varId,new VarSymbol(varId.getText(),varType));
		}
	}

	/**
	 * Process the constants defined at the start of the module.
	 * @param node The constants LpplNode
```

```
 */
private void processConstants(LpplNode node) {
        int numVars = node.getChildCount();
/* Every child is constant def   */

        for(int x=0; x<numVars; x++) {
                LpplNode constId = node.getChild(x);
                LpplTypeNode constIdType = (LpplTypeNode)constId.getChild(0);

                ConstSymbol cs = new ConstSymbol(constId.getText(),
                                new SymbolType(constIdType.getTypeType(),
/* The symbol type             */
                                                constIdType.isList(),
        /* Is it a list               */
                                                constId.getChildCount()==3));
        /* Is it a containment list      */

                if(constId.getChild(1).getType()!=LpplNode.NODE_LL) {
                        cs.addValue(constId.getChild(1).getText());
                }
                else {
                        LpplNode nodell = constId.getChild(1);
/* Get the list constant node    */
                        for(int y=0; y<nodell.getChildCount(); y++) {
                                cs.addValue(nodell.getChild(y).getText());
                        }
                }
                addSymbol(constId,cs);
        }
}

/**
 * Find a specific symbol in the symbol table
 * @param symbol The symbol name to retrieve
 * @return The symbol object
 */
private Symbol getSymbol(String symbol) {
        return g_symbolTable.getSymbol(symbol);
}

/**
 * Add the symbol to the current scope's symbol table
 * @param node The node this symbol is defined
 * @param symbol The symbol to add to the symbol table
 */
private void addSymbol(LpplNode node, Symbol symbol) {
        if(!g_symbolTable.addSymbol(symbol)){
                g_errFound = true;
```

```
                        g_errMsg = "Duplicate Symbol " + node.getText() + " at " +
node.getLine() + ":" + node.getCharPositionInLine();
                }
        }
}
```

## D.6.2 Class edu.lppl.walkers.CheckSymbolsWalker

```
package edu.lppl.walkers;

import edu.lppl.interfaces.TreeWalker;
import edu.lppl.symbols.FuncSymbol;
import edu.lppl.symbols.Symbol;
import edu.lppl.symbols.VarSymbol;
import edu.lppl.tree.LpplNode;
import edu.lppl.tree.LpplOpNode;

/**
 * This walker is responsible for checking the symbols used against the symbols
 * defined in the symbol table, or the semantic analysis of a LPPL module.
 * It checks and reports on the following errors:
 *
 * 1) Function calls match existing function definition's and has correct signiture
 * 2) Variables used have been defined as variables
 * 3) Assignment of constants
 *
 * This should be the second walker to traverse the LpplNode tree, and most certainly
 * must be sometime after BuildSymbolTableWalker, which initializes the symbol tables
 * this walker requires
 *
 * @author Brian
 *
 */
public class CheckSymbolsWalker implements TreeWalker {
        /**
         * Was an error found during the walk
         */
        private boolean g_errFound;

        /**
         * The message related to the error that was found during the walk
         */
        private String g_errMsg;

        /**
         * Generic Constructor
         *
         */
        public CheckSymbolsWalker() {
```

```
		g_errFound = false;
	}

	/**
	 * Get the error message that occurred while building the symbol tables
	 * @return The error message
	 */
	public String getErrorMessage() {
		return g_errMsg;
	}

	/**
	 * Walk the LpplNode tree
	 * @parm node The root LpplNode
	 * @return The root LpplNode
	 */
	public LpplNode walk(LpplNode node) {
		internalWalk(node);
		return (g_errFound ? null : node);
	}

	/**
	 * An internal walk of the tree which gets called recursively
	 * @param node The current LpplNode to process
	 */
	private void internalWalk(LpplNode node) {
		if(LpplNode.NODE_ID ==node.getType() && node.getChildCount() == 0)
{ /*its an ID and its being referenced */
			if(node.getSymbolTable().getSymbol(node.getText()) == null) {
				g_errFound = true;
				g_errMsg = "Error with " + node.getText()  +
finishErrMsg(node);
			}
			if(node.getSymbolTable().getSymbol(node.getText()) instanceof
FuncSymbol) {
				g_errFound = true;
				g_errMsg = "Variable " + node.getText() + " is actually a
function identifier " + finishErrMsg(node);
			}
		}
		else if(LpplNode.NODE_FC == node.getType()) {
			checkFunction(node,-1);
		}
		else if(LpplNode.NODE_OP == node.getType()) {
			int numReturns = 0;
			for(int x=0; x<node.getChildCount(); x++) {
				if(node.getChild(x).getType() == LpplNode.NODE_FC){
					break;
```

```
                        }
                        numReturns++;
                    }
                    if(((LpplOpNode)node).getOperation() ==
LpplOpNode.OP_ASSIGN) {
                        Symbol sym =
node.getSymbolTable().getSymbol(node.getChild(0).getText());
                        if(node.getChild(0).getType() != LpplNode.NODE_BLK
&& !(sym instanceof VarSymbol)) {
                            g_errFound = true;
                            g_errMsg = "Attempting to assign constant " +
node.getChild(0).getText() + finishErrMsg(node);
                        }
                        if(numReturns != node.getChildCount()) {
                            for(int x=1; x<numReturns; x++) {
                                sym =
node.getSymbolTable().getSymbol(node.getChild(x).getText());
                                if(node.getChild(x).getType() !=
LpplNode.NODE_BLK && !(sym instanceof VarSymbol)) {
                                    g_errFound = true;
                                    g_errMsg = "Attempting to assign
constant " + node.getChild(0).getText() + finishErrMsg(node);
                                }
                            }

        checkFunction(node.getChild(numReturns),numReturns);
                        }
                    }
                }
                if(LpplNode.NODE_FC != node.getType()) {
                    for(int x=0; x<node.getChildCount();x++) {
                        internalWalk(node.getChild(x));
                    }
                }
        }

        /**
         * Check a function call, making sure it exists and the signiture is consistant,
         * with n output parameters
         * @param node The function call LpplNode
         * @param numReturns The number of output parameters expected
         */
        private void checkFunction(LpplNode node, int numReturns) {
            int numParms = node.getChildCount()-1;
            Symbol funcSymbol =
node.getSymbolTable().getSymbol(node.getChild(0).getText());
            if(funcSymbol == null) {
                g_errFound = true;
```

```
                        g_errMsg = "Function " + node.getChild(0).getText() + " not found"
+ finishErrMsg(node);
                }
                else if(!(funcSymbol instanceof FuncSymbol)) {
                        g_errFound = true;
                        g_errMsg = "Symbol " + node.getChild(0).getText() + " is not a
function" + finishErrMsg(node);
                }
                else {
                        FuncSymbol fs = (FuncSymbol)funcSymbol;
                        if(fs.getNumInParms() != numParms || (numReturns != -1 &&
fs.getNumOutParms() != numReturns)) {
                                g_errFound = true;
                                g_errMsg = "Signiture does not match " +
fs.getFunctionSignature() + finishErrMsg(node);
                        }
                }
        }

        /**
         * Return the append of an error message
         * @param node The LpplNode where the error was found
         * @return The tail error message
         */
        private String finishErrMsg(LpplNode node) {
                return " at " + node.getLine() + ":" + node.getCharPositionInLine();
        }
}
```

### D.6.3 Class edu.lppl.walkers.RemoveVarDefsWalker

```
package edu.lppl.walkers;

import edu.lppl.interfaces.TreeWalker;
import edu.lppl.tree.LpplNode;

/**
 * This walker is responsible for removing the now unnecessary LpplNodes which
represent
 * the variable and constants definitions.  This can be thought of as an optimization
 * to reduce the number LpplNodes required to get from one place to another during
 * execution
 *
 * This should be the third walker to traverse the LpplNode tree, and most certainly
 * must be sometime after CheckSymbolsWalker, which relies on the previous form of the
 * tree, and does some final initialization.
 *
 * @author Brian
 *
```

```
 */
public class RemoveVarDefsWalker implements TreeWalker {

	/**
	 * There is no possibility of an error message
	 */
	public String getErrorMessage() {
		return null;
	}

	/**
	 * Walk the LpplNode tree
	 * @parm node The root LpplNode
	 * @return The root LpplNode
	 */
	public LpplNode walk(LpplNode node) {
		internalWalk(node);
		return node;
	}

	/**
	 * An internal walk of the tree which gets called recursively
	 * @param node The current LpplNode to process
	 */
	private void internalWalk(LpplNode node) {
		if(node.getType() == LpplNode.NODE_FL) {
			for(int x=0; x<node.getChildCount(); x++) {
				LpplNode funcNameNode = node.getChild(x);
				funcNameNode.removeChild(0);
		/* Remove input list		*/
				funcNameNode.removeChild(0);
		/* Remove output list		*/
			}
		}
		if(node.getType() == LpplNode.NODE_CB) {
			int childCount = node.getChildCount()-1;
			for(int x=0; x<childCount; x++) {
				node.removeChild(0);
		/* Remove var def		*/
			}

			LpplNode nodeSL =  node.getChild(0);
	/* Get the last one		*/
			for(int x=0; x<nodeSL.getChildCount(); x++) {
				node.addChild(nodeSL.getChild(x));
			}
			node.removeChild(0);
		}
```

```
                if(node.getType() == LpplNode.NODE_M) {
                        node.removeChild(1);
                /* Remove Constant List         */
                }
                for(int x=0; x<node.getChildCount(); x++) {
                        internalWalk(node.getChild(x));
                }
        }
}
```

## D.6.4 Class edu.lppl.walkers.PrintTreeWalker

```
package edu.lppl.walkers;

import edu.lppl.interfaces.TreeWalker;
import edu.lppl.tree.LpplNode;

/**
 * This walker is used as a debugging tool, and simply "pretty prints" out
 * the current LpplNode tree
 *
 * @author Brian
 *
 */
public class PrintTreeWalker implements TreeWalker{

        /**
         * There is no possibility of an error message
         */
        public String getErrorMessage() {
                return null;
        }

        /**
         * Walk the LpplNode tree
         * @parm node The root LpplNode
         * @return The root LpplNode
         */
        public LpplNode walk(LpplNode node) {
                printTree(node,"",false);
                return node;
        }

        /**
         * Print the tree with the given indent
         * @param node The LpplNode to print
         * @param indent The current indentation
         * @param isLast Is this the last LpplNode in an indentation
         */
```

```
	private void printTree(LpplNode node, String indent, boolean isLast) {
		System.out.println(indent + "->" + node.getText() + ":  " +
node.getSymbolTable());
		if(!isLast) {
			indent += "  |";
		}
		else {
			indent = indent.substring(0,indent.length()-3);
			indent += "     |";
		}
		for(int x=0; x < node.getChildCount(); x++) {
			printTree(node.getChild(x),indent,x==node.getChildCount()-1);
		}
	}
}
```

## D.7 Package edu.lppl.build

### D.7.1 Class edu.lppl.build.ExecuteCode

```
package edu.lppl.build;

import java.util.ArrayList;
import java.util.List;

import edu.lppl.symbols.ConstSymbol;
import edu.lppl.symbols.FuncSymbol;
import edu.lppl.symbols.VarSymbol;
import edu.lppl.tree.LpplCondNode;
import edu.lppl.tree.LpplNode;
import edu.lppl.tree.LpplOpNode;

/**
 * Execution of the LpplNode tree.   It is assumed the tree has
 * been checked for semantic accuracy and the variable definitions
 * in the tree have been trimmed off.
 *
 * @author Brian
 *
 */
public class ExecuteCode {
	private LpplNode g_rootNode;

	/**
	 * Start the execution of the module.
	 * @param node The root module node
	 */
	public void execute(LpplNode node) {
```

```
                g_rootNode = node;
                        /* Save the root node for later  */
                LpplNode func = findFunc("main");
        /* Find the main function       */
                if(func == null) {
                        /* Bad module                   */
                        System.out.println("main function not found");
                        return;
                }
                executeCodeBlock(func);
                        /* Execute main function        */
        }

        /**
         * Execute a specific code block (DO END block)
         * @param codeBlock The code block node to execute
         */
        public void executeCodeBlock(LpplNodecodeBlock) {
                codeBlock.getSymbolTable().initializeSymbolTable();                     /*
Clear local vars of this scope*/
                for(int x=0; x<codeBlock.getChildCount(); x++) {                /* Iterate
through each statement*/
                        LpplNode stmt = codeBlock.getChild(x);
                        if(stmt instanceof LpplOpNode) {
        /* Is it an operation           */
                                executeAssignment((LpplOpNode)stmt);
                        }
                        else if(stmt instanceof LpplCondNode) {
        /* Is it a conditional statement */
                                executeConditional((LpplCondNode)stmt);
                        }
                        else {
                                /* It is a SYSOUT statement      */
                                List l = executeExpression(stmt.getChild(0)); /* Execute and
get the value of the

                                        expression following ">"      */
                                System.out.print("> ");
                        /* Start the SYSOUT line        */
                                if(l == null || l.size() == 0) {                                /* Print
"null" if nothing was

                                        returned        from the expression  */

                                        System.out.println("null");
                                }
                                else {
                                        for(int y=0; y<l.size(); y++) {
```

```
                                        String outLine = (String)l.get(y);
                                        System.out.print(outLine);
                                }
                                System.out.println();
                        }
                }
        }
}

/**
 * Execute an assignment statement
 * @param assignStmt The assignment statement node to execute
 */
private void executeAssignment(LpplOpNode assignStmt) {
        int funcCallIdx = -1;

        for(int x=1; x<assignStmt.getChildCount(); x++) {               /* Iterate
through children      */
                if(assignStmt.getChild(x).getType() == LpplNode.NODE_FC) {
                        funcCallIdx = x;
                }
        }
        if(funcCallIdx == -1) {
                VarSymbol recSym = (VarSymbol)assignStmt.getSymbolTable()

        .getSymbol(assignStmt.getChild(0).getText());
                List val = executeExpression(assignStmt.getChild(1));
                recSym.setValue(val);
        }
        else {
                LpplNode funcCall = assignStmt.getChild(funcCallIdx);
                LpplNode funcNode = findFunc(funcCall.getChild(0).getText());
                FuncSymbol funcSym = (FuncSymbol)funcCall.getSymbolTable()

         .getSymbol(funcCall.getChild(0).getText());
                for(int x=1; x<funcCall.getChildCount(); x++) {
                        List val = executeExpression(funcCall.getChild(x));
                        funcSym.getInParm(x-1).setValue(val);
                }
                executeCodeBlock(funcNode);
                for(int x=0; x<funcSym.getNumOutParms(); x++) {
                        if(assignStmt.getChild(x).getType() !=
LpplNode.NODE_BLK) {
                                VarSymbol recSym =
(VarSymbol)assignStmt.getSymbolTable()

        .getSymbol(assignStmt.getChild(x).getText());
```

```
                recSym.setValue(funcSym.getOutParm(x).getValue(this));
                        }
                    }
                }
            }

            /**
             * Execute a conditional statement - done
             * @param condNode
             */
            private void executeConditional(LpplCondNode condNode) {
                ConstSymbol var1 = (ConstSymbol)condNode.getSymbolTable()

.getSymbol(condNode.getChild(0).getText());
                ConstSymbol listSymb = (ConstSymbol)condNode.getSymbolTable()

                .getSymbol(condNode.getChild(1).getText());
                boolean isIn = condNode.isIn();
                List symbList = listSymb.getValue(this);
                List var1List = var1.getValue(this);

                if(condNode.getConditionType() == LpplCondNode.COND_CHECK){
                    boolean isActuallyIn = false;
                    for(int x=0; x<symbList.size(); x++) {
                        if(var1List.get(x).equals(symbList.get(x))){
                            isActuallyIn = true;
                        }
                    }
                    if((isIn & isActuallyIn) || (!isIn & !isActuallyIn)) {
                        executeCodeBlock(condNode.getChild(2));
                    }
                }
                else {
                /* FOR EACH */
                    for(int x=0; x<symbList.size(); x++) {
                        ((VarSymbol)var1).setValue((String)symbList.get(x));
                        executeCodeBlock(condNode.getChild(2));
                    }
                }
            }

            /**
             * Execute an expression, returning result
             * @param expr
             * @return
             */
            private List executeExpression(LpplNode expr) {
```

```
		List l = new ArrayList();
		if(expr.getType() == LpplNode.NODE_LIT) {
			l.add(expr.getText());
		}
		else if(expr.getType() == LpplNode.NODE_ID) {

l.addAll(((ConstSymbol)expr.getSymbolTable().getSymbol(expr.getText())).getValue(this)
);
		}
		else if(expr.getType() == LpplNode.NODE_OP) {
			List val1 = executeExpression(expr.getChild(0));
			List val2 = executeExpression(expr.getChild(1));
			LpplOpNode opNode = (LpplOpNode)expr;
			int x;
			switch(opNode.getOperation()) {
			case LpplOpNode.OP_CONCAT:
				l.addAll(val1);
				l.addAll(val2);
				break;
			case LpplOpNode.OP_EXCLUDE
				for(x=0; x<val1.size(); x++) {
					boolean didFind = false;
					for(int y=0; y<val2.size(); y++) {
						if(val1.get(x).equals(val2.get(y))) {
							didFind = true;
						}
					}
					if(!didFind) {
						l.add(val1.get(x));
					}
				}
				break;
			case LpplOpNode.OP_INT:
				for(x=0; x<val1.size(); x++) {
					boolean didFind = false;
					for(int y=0; y<val2.size(); y++) {
						if(val1.get(x).equals(val2.get(y))) {
							didFind = true;
						}
					}
					if(didFind) {
						l.add(val1.get(x));
					}
				}
				break;
			case LpplOpNode.OP_MINUS:

				for(x=0; x<val1.size() && x<val2.size(); x++) {
```

```
                                        try {

        l.add(String.valueOf(Integer.valueOf((String)val1.get(x)).intValue()
                                                                        -
Integer.valueOf((String)val2.get(x)).intValue()));
                                        }catch(NumberFormatException nfe) {

                                        }
                                }
                                if(val1.size() < val2.size()) {
                                        for(int y=x; y<val2.size(); y++) {
                                                l.add(val2.get(y));
                                        }
                                }
                                else {
                                        for(int y=x; y<val1.size(); y++) {
                                                l.add(val1.get(y));
                                        }
                                }
                                break;
                        case LpplOpNode.OP_PLUS:
                                for(x=0; x<val1.size() && x<val2.size(); x++) {
                                        try {

        l.add(String.valueOf(Integer.valueOf((String)val1.get(x)).intValue()
                                                                        +
Integer.valueOf((String)val2.get(x)).intValue()));
                                        }catch(NumberFormatException nfe) {

                                        }
                                }
                                if(val1.size() < val2.size()) {
                                        for(int y=x; y<val2.size(); y++) {
                                                l.add(val2.get(y));
                                        }
                                }
                                else {
                                        for(int y=x; y<val1.size(); y++) {
                                                l.add(val1.get(y));
                                        }
                                }
                                break;
                        case LpplOpNode.OP_TIMES:
                                for(x=0; x<val1.size() && x<val2.size(); x++) {
                                        try {

        l.add(String.valueOf(Integer.valueOf((String)val1.get(x)).intValue()
```

```
                                                *
Integer.valueOf((String)val2.get(x)).intValue()));
                                        }catch(NumberFormatException nfe) {

                                        }
                                }
                                if(val1.size() < val2.size()) {
                                        for(int y=x; y<val2.size(); y++) {
                                                l.add(val2.get(y));
                                        }
                                }
                                else {
                                        for(int y=x; y<val1.size(); y++) {
                                                l.add(val1.get(y));
                                        }
                                }
                                break;
                        case LpplOpNode.OP_UNION:
                                for(x=0; x<val1.size(); x++) {
                                        boolean didFind = false;
                                        for(int y=0; y<l.size(); y++) {
                                                if(val1.get(x).equals(l.get(y))) {
                                                        didFind = true;
                                                }
                                        }
                                        if(!didFind) {
                                                l.add(val1.get(x));
                                        }
                                }
                                for(x=0; x<val2.size(); x++) {
                                        boolean didFind = false;
                                        for(int y=0; y<l.size(); y++) {
                                                if(val2.get(x).equals(l.get(y))) {
                                                        didFind = true;
                                                }
                                        }
                                        if(!didFind) {
                                                l.add(val2.get(x));
                                        }
                                }
                                break;
                        }
                }
                else if(expr.getType() == LpplNode.NODE_FC){
                        LpplNode funcNode = findFunc(expr.getChild(0).getText());
                        FuncSymbol funcSym = (FuncSymbol)expr.getSymbolTable()
                                                .getSymbol(expr.getChild(0).getText());
                        for(int x=1; x<expr.getChildCount(); x++) {
```

```
                        List val = executeExpression(expr.getChild(x));
                        funcSym.getInParm(x-1).setValue(val);
                    }
                    executeCodeBlock(funcNode);
                    l.addAll(funcSym.getOutParm(0).getValue(this));
                }
                return l;
        }

        /**
         * Return the node of the function identified by the given name - done
         * @param funcName
         * @return
         */
        public LpplNode findFunc(String funcName) {
                LpplNode func = null;
                LpplNode funcList = g_rootNode.getChild(1);
        /* GET NODE_FL                  */
                for(int x=0; x<funcList.getChildCount(); x++) {
                    if(funcName.equals(funcList.getChild(x).getText())) {
                        func = funcList.getChild(x).getChild(0);;
                    }
                }
                return func;
        }
}
```