LIMP: AN INTERPRETED PROGRAMMING LANGUAGE FOR STUDENTS, PROFESSORS AND PROGRAMMERS*

STUDENT PAPER

William Hawkins
Furman University Department of Computer Science
Greenville, SC
william.hawkins@furman.edu

ABSTRACT

Based upon an investigation of current interpreted and compiled programming languages, there exists a need for a new language with English-based keywords, looping constructs, functions, data type morphing and arrays. This paper describes a new language focused on these concepts: LIMP. To be useful, this new language is aimed towards users ranging from students to system administrators. LIMP was designed for ease of use and speed. This interpreted language was implemented in the fall of 2002 and an overview of the implementation is given here. LIMP succeeds by accomplishing all its design goals and executing programs at speeds comparable to or better than established languages. The result is a language for students of computer science and programmers of all skill levels.

1 INTRODUCTION

With computers increasing pervasiveness, computer science curriculum must be designed with majors and non-majors in mind - this includes mathematicians, physicists, and businesspeople who all make heavy use of computers. Recent college graduates in these and many other fields are called upon to automate repetitive tasks using the computer. In most undergraduate colleges, a computer science survey course is offered. This course usually covers

---

word processing, databases, HTML and graphic design but cannot address all the possible software combinations in use in industry today. By integrating a programming for non-computer-science-majors course into undergraduate curriculum, students gain fundamental insight into computer science that will benefit them as they undertake computer related tasks in the workplace.

LIMP[1], the new language designed by the author of this paper, is the tool for teaching such a class. While C, C++ and Java are often too complex to teach basic programming principles, LIMP is not. With its simple, English-based grammar, the concepts of programming (looping, control structures, input/output) can be taught without getting bogged down in the details of a particular programming environment. While the LIMP project was initially aimed at those programmers in search of a multi-purpose development tool, it has evolved into something that can be directly incorporated into undergraduate computer sciences courses. The initial goals of the project can be found later in section 3.

LIMP is an interpreted language. An interpreted language is one in which programs are executed immediately after it is parsed without generating any intermediate or object code. This is the opposite of a compiled language where the high-level source code is compiled into object code and then executed. The LIMP interpreter was designed from the outset for ultimate portability. The C and C++ LIMP source code will compile on any system with a C++ compiler and support for the Standard Template Library. The interpreter was written and tested on the Solaris operating system and was also compiled for testing under the Red Hat Linux operating system.

The remainder of this paper has the following format: Section 2 describes other's contributions to interpreted programming languages and how they impacted the design and implementation of LIMP. Section 3 lists the original goals of the LIMP programming language. Section 4 provides examples of tasks where LIMP would be the programmer's tool of choice. Section 5 describes LIMP's design and implementation. Section 6 describes the results of the benchmarks to compare LIMP's speed with that of other similar interpreted and compiled languages. Section 7 describes how the interpreter will be improved in the future.

2 RELATED WORK

As computing time becomes less expensive and processor speeds increase, computer scientists are becoming less concerned with the amount of time needed for algorithms to run, and more concerned with high level language constructs that make algorithm implementation easier. For this reason, many interpreted languages have been developed in the last 10 years most, notably Perl, PHP and Python.

One of the first interpreted languages to gain widespread acceptance was Perl. Invented in 1987, "Perl is a [sic] interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good

[1]LIMP is a recursively defined acronym for LIMP Is My Program.

language for many system management tasks."[4] Larry Wall, Perl's inventor, goes on to say that Perl was designed to combine the best features of C, `sed`, `awk` and `sh`. Perl is now the most popular interpreted programming language available for both the UNIX and Windows operating systems. Because so many people use Perl, many enhancements have been made over the years - most notably the support for object-oriented (OO) programming. While this is important today as programmers rely heavily on OO support, it results in an interpreter with more source code and slower response times for trivial programming tasks that do not require OO support. Over the years Perl developers have adopted the mantra that there is always more than one way to perform any operation. Unfortunately, not every method is fast or works the way the programmer expects. Often this leads to longer development time and slower running code. Since Perl was developed at the peak of C's popularity, many of the constructs in Perl are similar to those in C and do not lend themselves to high levels of source code abstraction. This makes Perl a difficult and intimidating language for novice programmers to learn and use. Overall, Perl is a good language for rapidly prototyping complex applications, but is not as impressive for smaller applications that need to run many times, as fast as possible. Section 6 shows a comparison of timing results between LIMP and Perl.

Riding Perl's coat tails, many similar languages were developed in the mid to late 1990s. The most notable is PHP. "PHP is an HTML-embedded scripting language. Much of its syntax is borrowed from C, Java and Perl with a couple of unique PHP-specific features thrown in. The goal of the language is to allow web developers to write dynamically generated pages quickly."[2] The language lives up to that description: its fast execution times have made it the primary language at large Internet companies including Yahoo!. However, PHP is not good for command line (or offline processing) work. It was designed as a web programming language, and the more recent versions are geared towards that market. Similar to Perl, what started off as a small set of CGI scripts[2] has turned into an OO programming language. Finally, PHP's syntax of combining executable statements into HTML (content markup) makes for very unreadable program source code. Overall, PHP is an expressive language for dynamic web content, but cannot be effectively used for offline processing, automation or command line scripting.

Like Perl and PHP, Python is an interpreted language. Python combines innovative programming language elements like exceptions and modules along with time-tested concepts such as classes and dynamic data types. [8] As a standalone language, Python is good for system administration tasks. However, according to the official Python website, the real power of Python lies in its ability to interoperate with heterogeneous environments. [10] "When augmented with standard extensions ... Python becomes a very convenient "glue" or "steering" language ..." [10]

The author of [10] goes on to say that several packages make Python strong enough to perform statistical calculations and even Fourier transforms. This seems like overkill for those

[2] "The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers." [1]

who are attempting to do simple and fast file, text, or number manipulation. Among non-Python programmers, there is a consensus that Python's syntax of indentation to create blocks of code is cumbersome. Python programmers refute this by saying the transition to this style of programming is easy and only takes a matter of hours.

There are many other similar languages, and some are more sophisticated than others. For instance, Ruby [11] is becoming a popular choice for scripting. No matter how many different scripting languages analyzed, the bottom line is that each language is designed to meet a specific goal. Perl is for rapid prototyping. PHP is for web development, and Python is for steering the interaction between multiple programs with easy to construct GUI components.

3 GOALS FOR LIMP

The goals of this language were set forth to fill a void left by current programming tools and create a language without the limitations of those already available.

- LIMP's syntax must be simple enough for novice programmer to learn quickly and robust enough for the expert programmer. Using more time to learn the syntax of a language leaves less time for the art of program design. With Perl and its C-like syntax, novice programmers can get lost trying to create even a simple script.
- LIMP must be relatively quick for all operations. This is not to say that all operations will be fast, just that the execution time for all operations must be respectable. With the languages detailed above and their support for OO constructs, the speed for simple operations, such as the creation of arrays, suffers. In the results section (section 6), the reader will notice that this is an area where our final product excelled in some areas and lagged in others.
- The language must support file input and output. Additionally, there must be easy constructs to open file handles and either read from or write to them. In Perl, the construct to open any kind of file handle is open The parameters differ depending on the way one wants to access the file, and are very similar to the symbols used in bash[3]. In LIMP, there are multiple constructs for opening files for read or write. This method makes it easier for programmers to be sure they have opened the file properly and that they are not going to accidentally overwrite important data by transposing a < with a >.
- The language must have built-in constructs that scale. This means that the language must have a small number of well-designed and well-implemented functions and constructs that can be used to create other commonly used functions. By avoiding feature overload, the programmer can ensure that the construct he is using is the fastest, cleanest way to accomplish a task. As was mentioned in the description of Perl above, the multitude of built-in constructs can lead to confusion about which is the easiest and fastest to use for a given problem. In the case of PHP, the online references for the built-in functions are

[3]BASH (Bourne Again SHell) is the GNU version of the original UNIX shell named sh.

helpful, but can sometimes lead the programmer astray as to what is the proper function or construct to use for a particular application.

- The LIMP interpreter must be portable enough to support all architectures with an available C++ compiler. If the tool is to be effective for the end programmer, the source code must run regardless of machine type. The author must have the freedom to take a program designed for one platform to another platform should the need arise.
- The tool should support interaction with the operating system. A language is useless if it cannot take advantage of operating system level functions.

4 MOTIVATING EXAMPLES

The primary goal of the project, as mentioned earlier, was to create a language for those interested in learning the fundamentals of computer science. Besides this academic application, there are also a number of ways a language like LIMP could benefit both novice and experienced programmers. This section outlines a few of these. The source code corresponding to these problems can be found in [5].

The Spreadsheet

In today's Microsoft-dominated desktop, most spreadsheets are created in Excel. While good for organizing data, the ability to perform operations on certain subsets of data in Excel is lacking. Written in the only programming language available, Visual Basic for Applications, these calculations can be slow and cumbersome. However, with LIMP's built-in substring operation, a number of standard functions, and easy access to files, these operations can be performed quickly.

The Email List

As corporations and departments grow, email servers become outdated and must be replaced. As a result, there exists a need to generate a new list of email addresses in the form `firstname.lastname@mailserver.example.com` from a list of email addresses in the form `firstname.lastname@example.com`. This seems like a trivial task, but the tools are not available for such a simple translation: A Perl script could do the job, but having a tool like LIMP, which is designed specifically for a task like this, would be ideal.

Shell Calls

Many tasks in the UNIX environment can be completed by making a series of shell calls with varying parameters. An obvious choice for such a task would be a shell script. However, as this programmer learned, using bash as a programming language is very difficult. Its column-oriented parsing and mathematical versus text mode processing of statements is difficult for many people to understand. However, with the `system()` call in LIMP, a shell call can

be executed and the output from that command can easily be stored in a variable for later manipulation.

Web Pages

Today, more than ever, people who do not have programming skills are attempting to publish dynamic World Wide Web homepages. Much to their chagrin, they must learn a language like Perl, PHP or Python to create such a web site. These languages are too similar to something like C (a *real* programming language) and offer little in the way of English based constructs. In LIMP, the use of English words to replace cryptic symbols common in other programming languages makes it easy for the non-programmer to create working scripts quickly. Using the provided CGI include file, access to any of the functions of PHP, Perl or Python designed for web site design are available to the LIMP programmer thereby creating a convenient language for the novice programmer to create dynamic web content.

In these four examples, plus the overall goal described in the paper's introduction, the reader can see there are plenty of practical applications for a niche language like LIMP.

5 DESIGN AND IMPLEMENTATION

The LIMP syntax is based partly on its C, C++ and Perl ancestors and partly on a theory that English-based operators will make it easier for both experienced and novice programmers to create working code that is self documenting. For a complete description of LIMP syntax, please see [5]. The interpreter for this language was implemented in the fall of 2002 using C and C++.

A complete description of the syntax and features of the language is beyond the scope of this paper. However, it is important to note a few features that make LIMP a suitable choice for classroom use.

Design and Features

1. All LIMP variables must be declared before they can be used. Not only does this reduce the risk of typos causing latent errors, but it also promotes good coding practices and leads to self-documenting code.
2. LIMP supports a single data type. Depending on the context of the operation, the data type can represent an integer, double, float, boolean, or string. Operational contexts in LIMP are straightforward so there is little possibility of inadvertently performing a meaningless operation.
3. LIMP supports arrays. To avoid problems with memory allocation, LIMP arrays are dynamically sized, and bounds are always checked for validity before performing an

access or write. This means that the formerly devastating off-by-one memory access error will no longer crash user programs.

4. LIMP supports functions. While most languages today support functions, LIMP's functions can be used to emphasize the technique of modularization.
5. LIMP supports various control structures. All LIMP control structures require { and } braces to contain the code to be conditionally executed. While this may increase typing, it has the advantage of minimizing mismatched if/else pairs and increasing code readability by grouping logically related statements.
    - *if ... else ...* LIMP includes support for the standard if statement with unlimited nesting.
    - *while, do ...* while LIMP includes support for a while, and do ... while loop.
    - *foreach* LIMP supports this looping structure for the express purpose of iterating through all the entries in an array. The concept of a foreach loop is simple: iterate through all entries in an array, assign the value from that entry into a programmer-specified variable and then execute a block of code. Upon completion of this block's execution, the process repeats for all the entries in the array.
6. LIMP supports native access to environment variables. For every defined environment variable, a identically named LIMP variable is created at program startup. Using the built-in keyword defined, the programmer can determine which environment variables are currently defined without worrying about referencing undeclared variables.
7. LIMP supports include files. The obvious consequence of this capability is to encourage code reusability and modularity.
8. LIMP includes a built-in debugger. Executing a LIMP program within the debugger allows the programmer to step the through the program and access variable contents. This simple but powerful debugger will benefit both experienced programmers and students of computer science.

Implementation

In the interest of space, implementation details of the LIMP interpreter have been omitted. A detailed description of the code for the interpreter can be found in [5]. However, a brief description of execution flow of the interpreter as a program is executed may help contextualize the following quantitative results.

1. The standard input and output channels are created.
2. The scope where "main" variables will be stored is created and entered into the list of active scopes.
3. If this is a debugger instantiation, the debugger variables are initialized.

4. Command line arguments are checked and proper variable initialization occurs. LIMP supports command line arguments to disable runtime warnings, specify a file to find LIMP code to execute and print a helpful message about LIMP's usage.
5. Control is passed to a `yacc`-generated parser. This parser relies on a flex generated lexical analyzer to tokenize LIMP source code [3] [7] [6]. During this time, the code is checked for syntax errors and converted into an execution tree. If any errors exist, they are reported and the interpreter halts. Otherwise, the interpreter continues.
6. The statements of the body of the program are compiled into an executable list and interpreter control is passed to the program evaluation engine. If during execution run-time errors occur, these errors are reported and the interpreter exits.
7. Upon successful completion of program execution, the interpreter exits.

Student Opportunities

As well as a tool for teaching computer science to non-computer scientists, LIMP offers excellent learning opportunities for advanced computer science students. The source code for the interpreter is open and available upon request. The documentation contained in the source code provides enough information for the interested programmer to make contributions. With over 5000 lines of code, modifying the LIMP interpreter provides a great starting point for students interested in compiler design. Specific suggestions for student work can be found in the Future Work section.

6 RESULTS

In order to quantitatively judge the final product, a set of benchmarks was created. These benchmarks were designed to judge LIMP's speed relative to established programming languages, compilers and interpreters. For each of the five benchmarks, results were gathered on a 250 MHz Sun4u machine running Solaris 5.8. In order to account for operating system overhead, the amount of time to execute the :[4] shell command from inside the timing program using the system() function was included in the results tables. Therefore, the actual amount of time to complete a benchmark can be inferred by subtracting the "Shell" time from the LIMP, Perl or C++ execution times in the tables below. In certain examples where a benchmark program was executed only once, the time to execute the : command was not included because it did not significantly effect the total running time. The C++ benchmarks were compiled using g++ and optimized with -O2. The full source code for the benchmark programs is in [5].

[4]The : is the null shell command. It can be compared to the empty statement ; in C.

Benchmark	Description
Recursion	The summation of the numbers 1 to 100 recursively.
Matrix Multiplication	Multiplication of a 50 by 50 matrix.
Function Calls	Execution of a function.
Arrays	Creation and manipulation of arrays.
Spreadsheet	"Substring" data from a large file.

Benchmark	Program	Real	User	System
Recursion	Shell	7.764s	2.74s	4.36s
	LIMP	41.867s	23.89s	14.31s
	Perl	38.046s	18.91s	16.94s
Matrix Multiplication	C	0.037s	0.02s	0.02s
	LIMP	16.128s	15.15s	0.81s
	Perl	1.354s	1.33s	0.02s
Function Calls	Shell	8.043s	2.90s	4.55s
	LIMP	22.887s	7.34s	12.61s
	Perl	34.986s	16.00s	16.88s
Arrays	Shell	6.454s	2.11s	3.76s
	LIMP	20.747s	6.39s	11.78s
	Perl	32.156s	14.32s	15.35s
Spreadsheet	C++	58.110s	43.490s	0.29s
	LIMP	30.163s	22.84s	1.53s
	C++ Optimized	11.131s	9.93s	0.25s

The matrix multiplication operations take a non-trivial amount of time to complete in LIMP because each value is stored as a C++ string; this string must be converted to a double before an operation can be performed and then the result must be converted back to a string. The substring operations in the Spreadsheet benchmark extract 30 characters from every line of a 30-megabyte file. Unfortunately, Perl results cannot be included for this benchmark. Perl's peephole optimizer completely eliminates the call to the substring function in our benchmark code. [9] The results from the Array benchmark prove an interesting point: the LIMP interpreter is able to handle sparsely populated n-dimensional arrays faster than other interpreters because of its underlying linked list implementation. The Function Class example shows that, calling and executing a function in LIMP is fast relative to other interpreters.

Objectively viewing these results demonstrates that the LIMP interpreter satisfies the design goal wherein all operations are interpreted relatively (to other languages) quickly.

7 FUTURE WORK

The most prominent interpreters today have been developed over a period of years. In Perl's case, development started in the late 1980s. For PHP and Python, development started

in the mid 1990s. LIMP development started in the fall of 2002. In addition to the areas of possible improvement mentioned previously, there are other ways to improve LIMP:

Eliminate the String Object

The C++ string object is at the heart of all variables, operations and literals. The overhead to carry around a complex class like this in so many structures slows down the interpreter. Additionally, there are numerous calls to convert the string class object to a standard C string. Therefore, if the string object can be changed to an array of characters, a significant amount of memory would be saved by not transferring an entire object from place to place and a significant time savings would result as calls to convert the C++ string object to a C String could be removed.

Implement Database Connectivity

As more data is migrated to databases, a language must have the capability to read and write from these data stores. LIMP needs this support. Initially support for the most popular databases will be added e.g. MySQL. This support will come in the form of functions such as `mysql_connect()` or `mysql_query()`.

Improve Speed

The methods used to execute LIMP programs could be more efficient. With an extensive audit of the source code, these problem areas can be identified and fixed to improve the overall speed. Additionally, the string base for every variable, or valuable in LIMP causes mathematical calculations to be slow (as noted in the matrix multiplication example in section 6). With the addition of a double to store the current mathematical representation of the string in all structures, repeated calls to `atof()` will be rendered unnecessary and an increase in speed should follow[5].

Remove Overflow Restrictions

Currently, a number in LIMP cannot be larger than $1x10^9$. Since LIMP is primarily a string based language, this is not a problem. However, as the general use of LIMP increases, it may be used for mathematical calculations that require support for larger numbers than LIMP currently allows.

---

[5]Based on informal tests, the LIMP interpreter will execute mathematical operations up to 50% faster with this enhancement.

Socket Connections

The ability for a LIMP program to connect to network resources via handles is an important future addition to LIMP. Since the handles were designed as a class, adding another type of resource (network) should not cause a huge change in the interpreter source code.

Other Enhancements

Because every LIMP variable can be considered a scalar *and* an array, a special index for every variable could be specified to hold information about the use of that variable. If LIMP had the ability to preprocess source files and extract this information, a program's documentation could be created automatically.

Users of the LIMP debugger will also benefit from increased access to behind-the-scenes information. This insight into the interpreter will help programmers whose complex code is infested with a subtle bug.

With any project, improvements can always be made. As people begin to use the language, this list of future improvements will surely increase.

8 SUMMARY

This paper has provided a general overview of a new programming language named LIMP, its uses and explicated possibilities for future work. This language features a simple syntax, support for file I/O, and built-in scalable constructs. Besides the accomplishment of these subjective goals, the quantitative results are promising. LIMP beat or rivaled other programming languages in all but one benchmark category. The examples of uses for LIMP show that this language is suitable for practical problem solving as well as for teaching the art and science of computing. Overall, the accomplishment of the design goals, and combination of subjective and quantitative results, shows that LIMP is a perfect tool for the classroom and industry. In the future this author looks forward to continuing work on this project, and hopes to end up with a language that programmers, non-programmers and teachers alike pick for the variety of problems they encounter and courses they teach.

Appendix: LIMP Grammar

```
delim = [ \t]
nl = [\n]
ws = {delim}+
non_digit_id_char = [_A-Za-z]
digit_char = [0-9]
variable_identifier =
\$({non_digit_id_char}|{digit_char})
+[0-9A-Z_a-z]*
regular_identifier =
{non_digit_id_char}[0-9A-Z_a-z]*
comment = ("//"|"#"){1}.*$
string_literal =
\"((\\\")|([^\"]))*\"
integer_literal =
-?[0-9]+[\.]?[0-9]*

INCLUDE = "include"
<<EOF>>   /*   Special   end   of   file
denominator*/
OPEN_BRACE = "{"
CLOSE_BRACE = "}"
OPEN_PAREN = "("
CLOSE_PAREN = ")"
SEMI = ";"
COMMA = ","
EQUAL "="
TILDE = "~"
OPEN_BRACKET = "["
```

```
CLOSE_BRACKET = "]"
L    I    T    E    R    A    L          =
{string_literal}|{integer_literal}
SUBSTR = "substr"
PRINT = "print"
READ = "read"
DATABASE = "database"
OPENR = "openr"
OPENW = "openw"
CLOSE = "close"
DECLARE = "declare"
FUNCTION = "function"
SYSTEM = "system"
RETURN = "return"
OR = "or"
AND = "and"
NOT = "not"
EQUAL_TO = "equal_to"
NOT_EQUAL_TO = "not_equal_to"
LESS_THAN = "less_than"
GREATER_THAN = "greater_than"
G R E A T E R _ T H A N _ O R _ E Q U A L     =
"greater_than_or_equal"
L E S S _ T H A N _ O R _ E Q U A L     =
"less_than_or_equal"
UNLESS = "unless"
DEFINED = "defined"
EOF = "eof"
IF = "if"
THEN = "then"
ELSE = "else"
DO = "do"
FOREACH = "foreach"
WHILE = "while"
ADD = "add"
SUBTRACT = "subtract"
MULTIPLY = "multiply"
DIVIDE = "divide"
MOD = "mod"
VAR_ID = {variable_identifier}
ID = {regular_identifier}
SYNTAX_ERROR = .

start_symbol      :  program  start_symbol
|

block : program block


else_block : program else_block
       |

program    : close_expr
      | function_expr
      | while_expr
      | if_expr
      | do_expr
      | foreach_expr
      | openr_expr
      | openw_expr
      | function_call_expr SEMI
      | system_call_expr SEMI
      | declare_expr
      | ass_expr
      | ass_expr_array
      | ret_expr
      | print_expr
      | read_expr
      | read_expr_array
      | include

include : INCLUDE LITERAL SEMI

if_expr  :    IF    OPEN_PAREN    valuable
CLOSE_PAREN    OPEN_BRACE    evaluated_block
CLOSE_BRACE
 |       IF     OPEN_PAREN     valuable
CLOSE_PAREN    OPEN_BRACE    evaluated_block
C L O S E _ B R A C E     E L S E     O P E N _ B R A C E
evaluated_else_block CLOSE_BRACE

do_expr  :       DO     OPEN_BRACE     block
CLOSE_BRACE   WHILE   OPEN_PAREN   valuable
CLOSE_PAREN SEMI

foreach_expr:    F O R E A C H      V A R _ I D
OPEN_PAREN    VAR_ID    CLOSE_PAREN
OPEN_BRACE block CLOSE_BRACE

while_expr:      W H I L E      O P E N _ P A R E N
valuable   CLOSE_PAREN   OPEN_BRACE   block
CLOSE_BRACE

function_call_expr:      ID     OPEN_PAREN
param_list CLOSE_PAREN

function_call_expr_return:       I        D
OPEN_PAREN param_list CLOSE_PAREN

system_call_expr:    SYSTEM    OPEN_PAREN
valuable CLOSE_PAREN

s y s t e m _ c a l l _ e x p r _ r e t u r n :     S Y S T E M
OPEN_PAREN valuable CLOSE_PAREN

print_expr: PRINT ID valuable SEMI

read_expr: READ ID VAR_ID SEMI

read_expr_array:    READ    ID    VAR_ID
evaluated_list SEMI

openr_expr: OPENR ID valuable SEMI

openw_expr: OPENW ID valuable SEMI

close_expr: CLOSE ID SEMI

function_expr:   FUNCTION   ID   OPEN_BRACE
block CLOSE_BRACE

partial  : partial COMMA valuable
```

```
     |

param_list        : valuable partial
          |

declare_expr:    DECLARE    var_id_list
SEMI

var_id_list       :     VAR_ID     COMMA
var_id_list
          | VAR_ID

index_list        :     OPEN_BRACKET
valuable CLOSE_BRACKET index_list
          |OPEN_BRACKET     valuable
CLOSE_BRACKET

evaluated_else_block : else_block

evaluated_block : block

evaluated_list : index_list

ret_expr : RETURN valuable SEMI

ass_expr  :  VAR_ID  EQUAL  valuable  SEMI

ass_expr_array     :     VAR_ID
evaluated_list EQUAL valuable SEMI

valuable   : valuable TILDE and_expr
     | and_expr ;

and_expr  : and_expr AND or_expr
     | or_expr ;

or_expr            :    or_expr    OR
comparison_expr
     | comparison_expr;

comparison_expr   :     comparison_expr
EQUAL_TO add_expr
 |     comparison_expr     NOT_EQUAL_TO
add_expr
         |    comparison_expr    LESS_THAN
add_expr
         |   comparison_expr   GREATER_THAN
add_expr
         |      comparison_expr
LESS_THAN_OR_EQUAL add_expr
         |      comparison_expr
GREATER_THAN_OR_EQUAL add_expr
         | add_expr ;

add_expr  : add_expr ADD mult_expr
 | add_expr SUBTRACT mult_expr
 | mult_expr ;

mult_expr    :    mult_expr    MULTIPLY
not_expr
 | mult_expr DIVIDE not_expr
 | mult_expr MOD not_expr
 | not_expr ;

not_expr  : NOT not_expr
 | EOF ID
 | DEFINED VAR_ID
 |  SUBSTR  OPEN_PAREN  last_resort  COMMA
last_resort    COMMA    last_resort
CLOSE_PAREN
 | last_resort;

last_resort      :
function_call_expr_return
         | system_call_expr_return
         | VAR_ID
         | VAR_ID index_list
         |     last_resort     OPEN_BRACE
valuable CLOSE_BRACE
         | LITERAL
         |     OPEN_PAREN     valuable
CLOSE_PAREN
```

REFERENCES

[1] *CGI: Common Gateway Interface*. http://hoohoo.ncsa.uiuc.edu/cgi/intro.html, -.

[2] *PHP Manual*. http://www.php.net/manual/en/preface.php, 2002.

[3] Alfred V. Aho. *Compilers: Principles, Techniques and Tools*. Reading, Mass: Addison-Wesley, 1986.

[4] Elaine Ashton. *The Timeline of Perl and its Culture*. http://history.perl.org/PerlTimeline.html, 2001.

[5] William Hawkins. *The Implementation and Uses of LIMP*. http://cs.furman.edu/~whawkins/Research-final.ps, 2002.

[6] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. http://dinosaur.compilertools.net/lex, -.

[7] M. E. Les and E. Schmidt. *Lex - A Lexical Analyzer Generator*. http://dinosaur.compilertools.net/lex, -.

[8] Guido Van Rossum. *What is Python?* http://www.python.org/cgi-bin/faqw.py?req=show&file=faq01.001.htp, 1997.

[9] Sriram Srinivasan. *Advanced Perl Programming.* O' Reilly, 1997.

[10] Aaron Watters. *What is Python good for?* http://www.python.org/cgi-bin/faqw.py?req=show&file=faq01.017.htp, 1997.

[11] Yukihiro Matsumoto. *What's Ruby*. http://www.ruby-lang.org/en/20020101.html, 2002.