University of North Alabama

PHP

Research paper

Abdullah O Alharbi

**Programming Languages** CS-410W

Professor Patricia Roden

23 April 2019

History:

PHP is a server-side scripting language which was created in 1994 by Rasmus Lerdorf, originally called “Personal Home Page”, in order to track visits to his online resume. However, Rasmus Lerdorf kept improving and adding to PHP and in 1995, he released the source code of PHP to the public. As a consequence, developers who were interested in PHP were encouraged to contribute in improving the language. Lerdorf extended PHP to work with forms and communicate with databases. As a result, “Personal Home Page/Forms Interpreter” frequently referenced as PHP/FI was released in April 1996, but it had stability issues with the parser. At that time, it was mostly developed by one person. However, as PHP/FI attracted more users, Zeev Suraski and Andi Gutmans volunteered to rewrite the parsing engine to resolve the problems, and along with Rasmus Lerdorf they agreed to make their work form the base of PHP 3. PHP 3 was released in June 1998, changing the language name to a recursive acronym for "PHP: Hypertext Preprocessor". PHP 3 had consistent language syntax and it supported object-oriented programming. In addition, it was extendable, so it attracted many developers to write, and submit modules for it. The usage of PHP increased dramatically, so Zeev Suraski, Andi Gutmans, and many other developers continued improving PHP which led to the release of PHP 4 on May 22, 2000. Additionally, in July 2004, PHP 5 was released; it was powered by the parsing engine Zend 2.0.

Design goals:

PHP was designed to be a server-side scripting language, so users could create dynamic websites which could communicate with databases. It was designed to be simple, so there is no need to add lots of commands to do a task such as outputting HTML. However, it was designed to be a familiar language. As a result, it is a C-like language, because C was popular that time, and many developers were familiar with it. Another goal of PHP is efficiency, accordingly it is interpreted, by the Zend engine, which is faster than many scripting languages. In addition, Flexibility is a goal of PHP, therefore it is a dynamic, weakly typed language, and it could be extended to add more feature. Another important goal of PHP is security; it was designed to be a secure language for writing CGI programs.

Syntax:

PHP is embedded in HTML, so a typical PHP file will have HTML tags, as well as PHP code. The server would process the PHP code which will be within the opening and closing tags; this mechanism allows PHP code to be embedded in HTML pages. As a result, everything outside the PHP delimiters will be sent to the output and not parsed by PHP. So, a PHP file must contain an opening tag “<?php”, and a closing tag “?>”. However, if the file only contains PHP code the closing tag should be omitted to avoid having whitespaces, or new lines after the closing tag which could cause unwanted issues. In addition to the tags shown above, PHP code could be within short open tags “<?” and closed with “?>”, but it is not recommended to use the short open tags because it must be enabled using the short\_open\_tag in php\_ini file prior to using it.

A PHP program consist of statements; statements could be as simple as printing a string on the screen, or a complicated loop. Statements must end with a semicolon, unless there are curly brackets, which specifies the scope of a compound statement, there is no need for a semicolon after the closing curly bracket.

Variables in PHP are case-sensitive, but keywords, user-defined functions, classes, namespaces, and language constructs are case-insensitive. For example, $sum, and $Sum are two distinct variables, but echo “Hello World”, and Echo “Hello World” are equivalent. PHP is a loosely typed language; hence PHP do not require the developer to declare a variable type when declaring variables. Although, this adds to the flexibility of PHP, it could cause unexpected run-time errors when variables are not carefully used. To make the code more readable, the user should organize the code by have the proper number of whitespaces, and new lines, as it is ignored by the interpreter.

Comments should be used to tell the purpose of the program, subprograms within a program, or even specify the data type a variable hold, since PHP is loosely typed. PHP is influenced by other programming languages, such as C, C++, and Perl. As a result, it supports a C++ single line comments, so “//” and everything after it in the same line will be commented. In addition, Shell-style comments are supported by PHP as well, so when PHP encounters “#”, everything after it until the end of the line will be commented. On the other hand, it has a C-style multiline comments; everything between “/\*” and “\*/” is ignored during runtime. Unlike, the shell-style and C++ comments that are single line comments, C-style multiline comments could comment multiple lines until it encounters an asterisk followed by a slash “\*/”. As a consequence, it should be used carefully, or the program may have errors because some lines are commented unintentionally.

Identifiers are used to name functions, classes, or variables. The first character of an identifier must start with an underscore “\_”, an ASCII letter (upper-case, or lower-case), or characters between ASCII 0x7F and ASCII 0xFF. The other characters after the first one, could be from these characters, or any number 0-9. For instance, $\_sum, and $sum2 are valid variables names. In PHP, variables must start with a dollar sign character “$”, and they are case-sensitive. However, functions do not start with a dollar sign, and they are case-insensitive, for example “show\_names” is a valid function name. There are reserved words (or keywords), such as “echo”, that are part of the language. Therefore, functions, classes, or variables cannot be named as keywords (provided in the appendix).

Data types, type checking, and scoping:

PHP has eight different data types, four of them are scalar types which are, integer, float, boolean, and string; two of them are compound types which are array, and object, and finally two special types that are NULL, and resource. Float is a floating-point number, such as 3.14, integer is a whole number, boolean could be true or false, NULL is a special type that has the value NULL, string is a sequence of characters such as “Hello World”, array is a data structure that contains group of elements, object is an instance of a user-defined class, and resource is a special type that holds reference to an external resource.

A variable in PHP holds the value of the most recent assignment; PHP automatically does type converting when necessary. As specified earlier, PHP is loosely typed language, so a variable does not know which data type it is going to hold prior to assigning a value to it. However, if a variable is not assigned a value it holds the default value.

Additionally, variables could be used within their scopes, if a variable is declared outside the curly brackets of a function, it cannot be “seen” by the function unless the keyword global is used before it, so it refers to the global version of the variable that is declared outside the function. Otherwise, if the global keyword is not used then the variable will be considered a different variable, even if it has the same name, since it is not within the scope of the function.

Data structures:

PHP 5 and later versions have the Standard PHP Library (SPL) which contains several predefined classes of standard data structures. Such as, SplDoublyLinkedList which are nodes that are connected to each other from both ends, SplHeap which is tree-like structure, it holds the property that each parent node is greater than or equal to its children, SplPriorityQueue which is a priority queue implemented using a max heap, SplFixedArray which is a fixed size array, and SplObjectStorage which is a map that holds key-value pairs. Since PHP 5.3.0, SPL cannot be disabled, so it is always available.

Control structures:

PHP supports few traditional flow-control statements; conditional statements such as switch statements, and loops such as while loops. An if statement check whether an expression is true or false. If the expression is true the code specified by the “if statement” is executed, if it is false the alternative code in specified by the “else” keyword is executed. For example,

if ($grade >= 60){

echo “passed”;

} else {

echo “failed”;

}

Another conditional statement is the elseif statement which makes it easier, and more efficient to write nested if statements. Additionally, PHP has switch statement which is similar to the nested if statement. However, it compares the value of the given expression to all of its cases, then executes the statements of the matching case until the break keyword is encountered. If none of the cases match the value of the given expression, then if the default case is given, the statements following the default keyword are executed until a break keyword is encountered. For instance, if we have the following nested if statement:

if ($id == 1) {

echo “John”;

}

elseif ($id == 2) {

echo “Joe”;

}

else {

echo “Not found”;

}

We could replace it by the following switch statement:

switch ($id) {

case 1:

echo “John”;

break;

case 2:

echo “Joe”;

break;

default:

echo “Not found”;

break;

}

PHP supports pre-test, as well as post-test loops. The while loop in PHP is similar to the while loop in C. Therefore, it is a pre-test loop that checks the value of an expression and executes certain statements, the reevaluates the expression again and repeats the same step until it evaluates to false. In addition, it supports a do-while loop which is a post-test loop. So, it is guaranteed that the code inside the loop will execute at least once. Then, an expression is evaluated, and the same steps are repeated until the expression is evaluated to false. PHP also supports for loops which are similar to while loops except it has a counter expression; it usually easier to read than while loops. Finally, PHP supports foreach loop which allows the user to iterate over the elements of an array. So, the code inside the loop will be executed as many times as the number of elements in the array.

Subprograms including implementation:

PHP has built-in functions, and it allows the user to define functions. As shown above, the first character of a function must start with an underscore “\_”, an ASCII letter (upper-case, or lower-case), or characters between ASCII 0x7F and ASCII 0xFF. The other characters after the first one, could be from these characters, any number 0-9. The name of the user-defined functions cannot be one of the reserved words of PHP; the name of the function is case-insensitive. So, show\_name () is equivalent to Show\_name (). The number of parameters of a function could be zero, or more. However, the order of parameters must be how the function expects it; it might work if the parameters are not passed in the correct order, but it will be a matter of garbage in = garbage out. As a result, the developer should always describe how the function is expecting parameters for others to use it. So, to define a function in PHP we must use the following syntax (the EBNF is included in the appendix):

function [&] function\_name ([parameter [, …]]) {

code to execute;

}

To use variables declared outside the scope of the function, we need to use the global keyword. Otherwise, the local variable is not referring to the global variable; they will be two distinct variables even if they have the same name.

In PHP, a function could be defined inside a function. However, that does not limit the visibility of the inner function, as it could be called anywhere in the program, but after the outer function has already been called.

Data abstraction and encapsulation:

Data abstraction is the process of hiding all but essential data to the user. PHP provides data abstraction using classes. On the other hand, data encapsulation is the process of grouping data and functions that manipulate them; PHP supports encapsulation, since it allows the user to define classes that have data and functions.

Parameter passing:

There are two ways to pass parameters to a function in PHP. Firstly, pass by value which passes a copy of the value of the variable. As a result, if a variable is passed by value and within the function, we change the value it will not change the value of the original variable. The other way is to pass parameters by reference which means the function will have access to the actual variable, so changing the value of the variable inside the function will change the value of the original variable.

Concurrency:

PHP supports concurrency which is the process of handling to processes at the same time. So, the Thread class in PHP allows the user to define threads, implement the run method, then call the method Thread::start() to execute the code of the run method.

Recursion:

Recursion in this context is when a function calls itself; it could be useful when implementing some data structures such as linked list. As many other programming languages, PHP supports recursion. As a consequence, to write a recursive function, simply let the function call itself, then it is going to call itself again. However, it could be a dangerous process if not written carefully. A recursive function should always have a base case which means the end of the recursion, so it is going to return a value without any recursive calls, and a reduction step.

Exception handling:

PHP has various ways to handle exceptions. One of them is error suppression which disable the error messages for a single statement. To use this method, you should use the error suppression operator “@” before the expression. So, if error suppression is used for a statement that causes an error the program will not halt; the expression is not going to do anything, but this method might cause many issues, since the program might be in unknown state if an error occurred during runtime. However, parse error cannot be suppressed by the error suppression operator, only runtime errors. Additionally, we could use error triggering, its syntax is: trigger\_error(message [, type]);

The first parameter is the message that should be displayed when the error occurs, and the second parameter (optional) is the error type which must be one of the E\_USER errors; the default is E\_USER\_NOTICE which is runtime notice. PHP 5, and later versions have the new “try”, “catch”, “throw”, and “finally” keywords. If statements could cause an error, it should be in a try block. If the code does not have errors it executes as normal, but if the code triggers an exception, the code inside the “catch” block will be executed. The exception is thrown by using the “throw” keyword which throws an instance of the Exception class, or an instance of one of its subclasses. The code inside the “finally” block will be executed regardless of if an exception occurred; it is executed immediately after the code in the “try” block or the code in the “catch” block.

Expressions and the Assignment Statement:

“An expression is a bit of PHP that can be evaluated to produce a value. The simplest expressions are literal values and variables. A literal value evaluates to itself, while a variable evaluates to the value stored in the variable. More complex expressions can be formed using simple expressions and operators. An operator takes some values (the operands) and does something (for instance, adds them together). Operators are written as punctuation symbols—for instance, the + and – familiar to us from math. Some operators modify their operands, while most do not.” (Tatroe, Lerdorf and MacIntyre)

“The basic assignment operator (=) assigns a value to a variable. The lefthand operand is always a variable. The righthand operand can be any expression—any simple literal, variable, or complex expression. The righthand operand’s value is stored in the variable named by the lefthand operand.

Because all operators are required to return a value, the assignment operator returns the value assigned to the variable. For example, the expression $a = 5 not only assigns 5 to $a, but also behaves as the value 5 if used in a larger expression.” (Tatroe, Lerdorf and MacIntyre)

Input/Output:

Input in PHP could be gathered by the predefined “superglobals”, such as $\_GET, and $\_POST which are associative arrays of variables passed by the HTTP POST or GET methods. However, output in PHP is done by sending the result of “echo” or other outputting commands to the browser. Alternatively, the function ob\_start to turn the output buffering on, then the output that would normally be sent to the browser will be placed in a buffer. As a result, the user could send it later to the browser, or kill it.

However, there are predefined functions that are built in PHP in order to read from or write to files. Such as, the “fgets” function which reads a line from a file pointer. However, the pointer must point to a valid file which had been opened successfully by the fopen() function and not yet closed.

Other unusual features:

PHP superglobals; they are built-in variables which are available in all scopes. As a result, they could be used throughout a script without using the “global” keyword. Additionally, PHP has two ways to concatenate strings, by using the concatenation operator “.”, or by using the concatenating assignment operator “.=”

For example,

Concatenation operator:

$x = “Hello”;

$y = x . “ world”; // y will hold the string “Hello world”

Concatenating assignment operator

$x = “Hello”;

$x .= “ world”; // x will hold the string “Hello world”

Contributions to the programming language landscape:

PHP made it easier to do web development, since PHP is an “HTML-embedded scripting language”, which means PHP pages contain HTML with PHP code embedded in it; it takes less commands to do certain tasks involving HTML in PHP than C, or Perl for instance. Additionally, PHP is the most commonly used server-side programming language nowadays.

Sample program (shopping cart):

<?php

session\_start(); ?> <!-- has to be the first thing -->

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Shopping cart</title>

</head>

<body>

<?php

// add name item to our shopping cart

if (isset($\_POST ['name'])) {

if (isset($\_SESSION['cart'])) {

$cart = $\_SESSION['cart'];

} else {

// create cart

$cart = array(); // create an empty array

}

// add item to cart

$cart[] = $\_POST['name'];

$\_SESSION['cart'] = $cart;

}

?>

<form action="cart.php" method="POST">

Item: <input type="text" name="name">

<input type="submit">

</form>

<!-- display cart contents -->

<?php

if (isset($cart)) {

foreach ($cart as $item) {

echo $item . "<br>";

}

}

?>

</body>

</html>

Global issues:

Constantly changing; many PHP releases have some issues and the only way to get rid of them is by upgrading your version. Additionally, PHP has inconsistent built-in function names, such as “isset”, and “is\_null”. As a result, it could be hard for programmers to remember the name of the built-in function that they would like to use.

Promise for the future:

The Zend has already started working on developing a new JIT (Just In Time) engine which is believed to improve the performance of PHP. Since, PHP is an open source project, users could download the source code of the JIT engine. However, the engine is still not complete, and it may be included with the next major release of PHP which is PHP 8.

Appendix

Keywords:

\_\_halt\_compiler

abstract

and

array

as

break

callable

case

catch

class

clone

const

continue

declare

default

die

do

echo

else

elseif

empty

enddeclare

endfor

endforeach

endif

endswitch

endwhile

eval

exit

extends

final

for

foreach

function

global

goto

if

implements

include

include\_once

instanceof

insteadof

interface

isset

list

namespace

new

or

print

private

protected

public

require

require\_once

return

static

switch

throw

trait

try

unset

use

var

while

xor

# PHP 5.2.0 EBNF Syntax

# Converted from the yacc syntax, see file Zend/zend\_language\_parser.y

# by Umberto Salsi <salsi@icosaedro.it>.

# This file last updated: 2007-03-16

# Added the "define();" statement.

# Added some symbols from the scanner.

# FIXME: missing definition for T\_INLINE\_HTML.

# FIXME: missing definition for T\_ENCAPSED\_AND\_WHITESPACE.

# FIXME: missing definition for T\_CHARACTER.

PHP\_SOURCE\_TEXT = { inner\_statement | halt\_compiler\_statement };

halt\_compiler\_statement = "\_\_halt\_compiler" "(" ")" ";" ;

inner\_statement = statement

| function\_declaration\_statement

| class\_declaration\_statement ;

inner\_statement\_list = { inner\_statement } ;

statement = "{" inner\_statement\_list "}"

| "if" "(" expr ")" statement {elseif\_branch} [else\_single]

| "if" "(" expr ")" ":" inner\_statement\_list {new\_elseif\_branch}

[new\_else\_single] "endif" ";"

| "while" "(" expr ")" while\_statement

| "do" statement "while" "(" expr ")" ";"

| "for" "(" for\_expr ";" for\_expr ";" for\_expr ")" for\_statement

| "switch" "(" expr ")" switch\_case\_list

| "break" [expr] ";"

| "continue" [expr] ";"

| "return" [expr\_without\_variable | variable] ";"

| "global" global\_var {"," global\_var} ";"

| "static" static\_var { "," static\_var } ";"

| "echo" echo\_expr\_list ";"

| T\_INLINE\_HTML

| expr ";"

| "use" use\_filename ";" # FIXME: not implemented

| "unset" "(" variable {"," variable} ")" ";"

| "foreach" "(" (variable|expr\_without\_variable)

"as" foreach\_variable ["=>" foreach\_variable] ")"

foreach\_statement

| "declare" "(" declare\_list ")" declare\_statement

| ";" # empty statement

| "try" "{" inner\_statement\_list "}" catch\_branch {catch\_branch}

| "throw" expr ";" ;

catch\_branch = "catch" "(" fully\_qualified\_class\_name T\_VARIABLE ")" "{"

inner\_statement\_list "}" ;

use\_filename = T\_CONSTANT\_ENCAPSED\_STRING

| "(" T\_CONSTANT\_ENCAPSED\_STRING ")" ;

function\_declaration\_statement = "function" ["&"] T\_STRING

"(" parameter\_list ")" "{" inner\_statement\_list "}" ;

class\_declaration\_statement = class\_entry\_type T\_STRING

[extends\_from] [implements\_list] "{" {class\_statement} "}"

| "interface" T\_STRING [interface\_extends\_list] "{" {class\_statement} "}" ;

class\_entry\_type = [ "abstract" | "final" ] "class" ;

extends\_from = "extends" fully\_qualified\_class\_name ;

interface\_extends\_list = "extends" interface\_list ;

implements\_list = "implements" interface\_list ;

interface\_list = fully\_qualified\_class\_name { "," fully\_qualified\_class\_name } ;

foreach\_variable = ["&"] variable ;

for\_statement = statement

| ":" inner\_statement\_list "endfor" ";" ;

foreach\_statement = statement

| ":" inner\_statement\_list "endforeach" ";" ;

declare\_statement = statement

| ":" inner\_statement\_list "enddeclare" ";" ;

declare\_list = T\_STRING "=" static\_scalar { "," T\_STRING "=" static\_scalar } ;

switch\_case\_list = "{" [";"] {case\_list} "}"

| ":" [";"] {case\_list} "endswitch" ";" ;

case\_list = "case" expr [":"|";"] inner\_statement\_list

| "default" [":"|";"] inner\_statement\_list ;

while\_statement = statement

| ":" inner\_statement\_list "endwhile" ";" ;

elseif\_branch = "elseif" "(" expr ")" statement ;

new\_elseif\_branch = "elseif" "(" expr ")" ":" inner\_statement\_list ;

else\_single = "else" statement ;

new\_else\_single = "else" ":" inner\_statement\_list ;

parameter\_list = [ parameter {"," parameter} ] ;

parameter = [T\_STRING | "array"] ["&"] T\_VARIABLE ["=" static\_scalar] ;

function\_call\_parameter\_list = [ function\_call\_parameter

{ "," function\_call\_parameter } ] ;

function\_call\_parameter = expr\_without\_variable

| variable

| "&" w\_variable ;

global\_var = T\_VARIABLE

| "$" r\_variable

| "$" "{" expr "}" ;

static\_var = T\_VARIABLE [ "=" static\_scalar ] ;

class\_statement = variable\_modifiers class\_variable\_declaration

{"," class\_variable\_declaration} ";"

| "const" class\_constant\_declaration {"," class\_constant\_declaration} ";"

| {modifier} "function" ["&"] T\_STRING "(" parameter\_list ")"

method\_body ;

method\_body = ";"

| "{" inner\_statement\_list "}" ;

variable\_modifiers = "var" | modifier {modifier} ;

modifier = "public" | "protected" | "private" | "static" | "abstract"

| "final" ;

class\_variable\_declaration = ("var" | modifier {modifier}) T\_VARIABLE ["=" static\_scalar];

class\_constant\_declaration = T\_STRING "=" static\_scalar ;

echo\_expr\_list = expr {"," expr} ;

for\_expr = [ expr {"," expr} ] ;

expr\_without\_variable = "list" "(" assignment\_list ")" "=" expr

| variable "=" expr

| variable "=" "&" variable

| variable "=" "&" "new" class\_name\_reference [ctor\_arguments]

| "new" class\_name\_reference [ctor\_arguments]

| "clone" expr

| variable ("+=" | "-=" | "\*=" | "/=" | ".=" | "%=" | "&=" | "|=" |

"^=" | "<<=" | ">>=" ) expr

| rw\_variable "++"

| "++" rw\_variable

| rw\_variable "--"

| "--" rw\_variable

| expr ("||" | "&&" | "or" | "and" | "xor" | "|" | "&" | "^" | "." |

"+" | "-" | "\*" | "/" | "%" | "<<" | ">>" | "===" | "!==" |

"<" | "<=" | ">" | ">=" ) expr

| ("+" | "-" | "!" | "~") expr

| expr "instanceof" class\_name\_reference

| "(" expr ")"

| expr "?" expr ":" expr

| internal\_functions

| "(int)" expr

| "(double)" expr

| "(float)" expr

| "(real)" expr

| "(string)" expr

| "(array)" expr

| "(object)" expr

| "(bool)" expr

| "(boolean)" expr

| "(unset)" expr # FIXME: not implemented

| "exit" [exit\_expr]

| "die" [exit\_expr]

| "@" expr

| scalar

| "array" "(" [array\_pair\_list] ")"

| "`" encaps\_list "`"

| "print" expr ;

function\_call = T\_STRING "(" function\_call\_parameter\_list ")"

| fully\_qualified\_class\_name "::" T\_STRING

"(" function\_call\_parameter\_list ")"

| fully\_qualified\_class\_name "::" variable\_without\_objects

"(" function\_call\_parameter\_list ")"

| variable\_without\_objects "(" function\_call\_parameter\_list ")" ;

fully\_qualified\_class\_name = T\_STRING ;

class\_name\_reference = T\_STRING

| dynamic\_class\_name\_reference ;

dynamic\_class\_name\_reference = base\_variable "->" object\_property

{ "->" object\_property }

| base\_variable ;

exit\_expr = "(" [expr] ")" ;

ctor\_arguments = "(" function\_call\_parameter\_list ")" ;

common\_scalar = T\_LNUMBER | T\_DNUMBER | T\_CONSTANT\_ENCAPSED\_STRING

| "\_\_LINE\_\_" | "\_\_FILE\_\_" | "\_\_CLASS\_\_" | "\_\_METHOD\_\_" | "\_\_FUNCTION\_\_" ;

# FIXME: very bad syntax, $x = + + + 4; is valid!

static\_scalar = common\_scalar

| T\_STRING

| "+" static\_scalar

| "-" static\_scalar

| "array" "(" [static\_array\_pair\_list] ")"

| static\_class\_constant ;

static\_class\_constant = T\_STRING "::" T\_STRING ;

scalar = T\_STRING

| T\_STRING\_VARNAME

| class\_constant

| common\_scalar

| "\"" encaps\_list "\""

| "'" encaps\_list "'"

| T\_START\_HEREDOC encaps\_list T\_END\_HEREDOC ;

static\_array\_pair\_list = static\_array\_pair { "," static\_array\_pair } [","] ;

static\_array\_pair = static\_scalar ["=>" static\_scalar] ;

expr = r\_variable | expr\_without\_variable ;

r\_variable = variable ;

w\_variable = variable ;

rw\_variable = variable ;

variable = base\_variable\_with\_function\_calls [ "->" object\_property

method\_parameters { "->" object\_property method\_parameters } ] ;

method\_parameters = "(" function\_call\_parameter\_list ")" ;

variable\_without\_objects = reference\_variable

| simple\_indirect\_reference reference\_variable ;

static\_member = fully\_qualified\_class\_name "::" variable\_without\_objects ;

base\_variable\_with\_function\_calls = base\_variable | function\_call ;

base\_variable = reference\_variable

| simple\_indirect\_reference reference\_variable

| static\_member ;

reference\_variable = compound\_variable { selector } ;

compound\_variable = T\_VARIABLE | "$" "{" expr "}" ;

selector = "[" [expr] "]" | "{" expr "}" ;

object\_property = variable\_name { selector }

| variable\_without\_objects ;

variable\_name = T\_STRING | "{" expr "}" ;

simple\_indirect\_reference = "$" {"$"} ;

assignment\_list = [assignment\_list\_element] {"," [assignment\_list\_element]} ;

assignment\_list\_element = variable

| "list" "(" assignment\_list ")" ;

array\_pair\_list = array\_pair {"," array\_pair} [","] ;

array\_pair = "&" w\_variable

| expr "=>" "&" w\_variable

| expr "=>" expr ;

encaps\_list =

{

encaps\_var

| T\_STRING

| T\_NUM\_STRING

| T\_ENCAPSED\_AND\_WHITESPACE

| T\_CHARACTER

| T\_BAD\_CHARACTER

| "["

| "]"

| "{"

| "}"

| "->"

} ;

encaps\_var = T\_VARIABLE [ "[" encaps\_var\_offset "]" ]

| T\_VARIABLE "->" T\_STRING

| "${" expr "}"

| "${" T\_STRING\_VARNAME "[" expr "]" "}"

| T\_CURLY\_OPEN variable "}" ;

encaps\_var\_offset = T\_STRING | T\_NUM\_STRING | T\_VARIABLE ;

internal\_functions = "isset" "(" variable {"," variable} ")"

| "empty" "(" variable ")"

| "include" expr

| "include\_once" expr

| "eval" "(" expr ")"

| "require" expr

| "require\_once" expr ;

class\_constant = fully\_qualified\_class\_name "::" T\_STRING ;

#

# Some tokens from the scanner (see file Zend/zend\_language\_scanner.l):

#

LABEL = (letter | "\_") {letter | digit | "\_"} ;

T\_STRING = LABEL;

T\_BAD\_CHARACTER = "\x00".."\x08" | "\x0b" | "\x0c" | "\x0e".."\x1f" ;

T\_VARIABLE = "$" T\_STRING ;

T\_LNUMBER = octal | decimal | hexadecinal ;

octal = "0" {"0".."7"} ;

decimal = "1".."9" {digit} ;

hexadecinal = "0x" hexdigit {hexdigit} ;

digit = "0".."9" ;

hexdigit = digit | "a".."f" | "A".."F" ;

letter = "a".."z" | "A".."Z" | "\x7f".."\xff" ;

T\_DNUMBER = DNUM | EXPONENT\_DNUM;

DNUM = digit ["."] digit {digit} | digit {digit} ["."] {digit};

EXPONENT\_DNUM = (LNUM | DNUM) ("e"|"E") ["+"|"-"] LNUM;

LNUM = digit {digit};

T\_CURLY\_OPEN = "${";

T\_CONSTANT\_ENCAPSED\_STRING = single\_quoted\_constant\_string | double\_quoted\_constant\_string;

# FIXME

single\_quoted\_constant\_string =

"'" { "any char except ' and \\" | "\\" "any char" } "'";

# FIXME

double\_quoted\_constant\_string =

"\"" { "any char except $ \" and \\" | "\\" "any char" } "\"";

T\_STRING\_VARNAME = LABEL;

T\_NUM\_STRING = LNUM | hexadecinal;

T\_START\_HEREDOC = "<<<" {" "|"\t"} LABEL NEWLINE;

NEWLINE = "\r"|"\n"|"\r\n";

T\_END\_HEREDOC = "FIXME: here at the beginning of the line"

LABEL [";"] NEWLINE;

Work Cited

Tatroe, Kevin, Rasmus Lerdorf, and Peter MacIntyre. *Programming PHP*. Sebastopol, CA: O'Reilly Media, 2013. Print.

"Usage Statistics And Market Share Of Server-Side Programming Languages For Websites, April 2019." *W3techs.com*. N.p., 2019. Web.

"PHP: List Of Keywords - Manual." *Php.net*. N.p., 2019. Web. 23 Apr. 2019.

"PHP: User-Defined Functions - Manual." *Php.net*. N.p., 2019. Web. 23 Apr. 2019.

Hansen, Thomas Blom, and Jason Lengstorf. *PHP For Absolute Beginners*. Print.