<?PastPHP ?>

Caleb Rogers

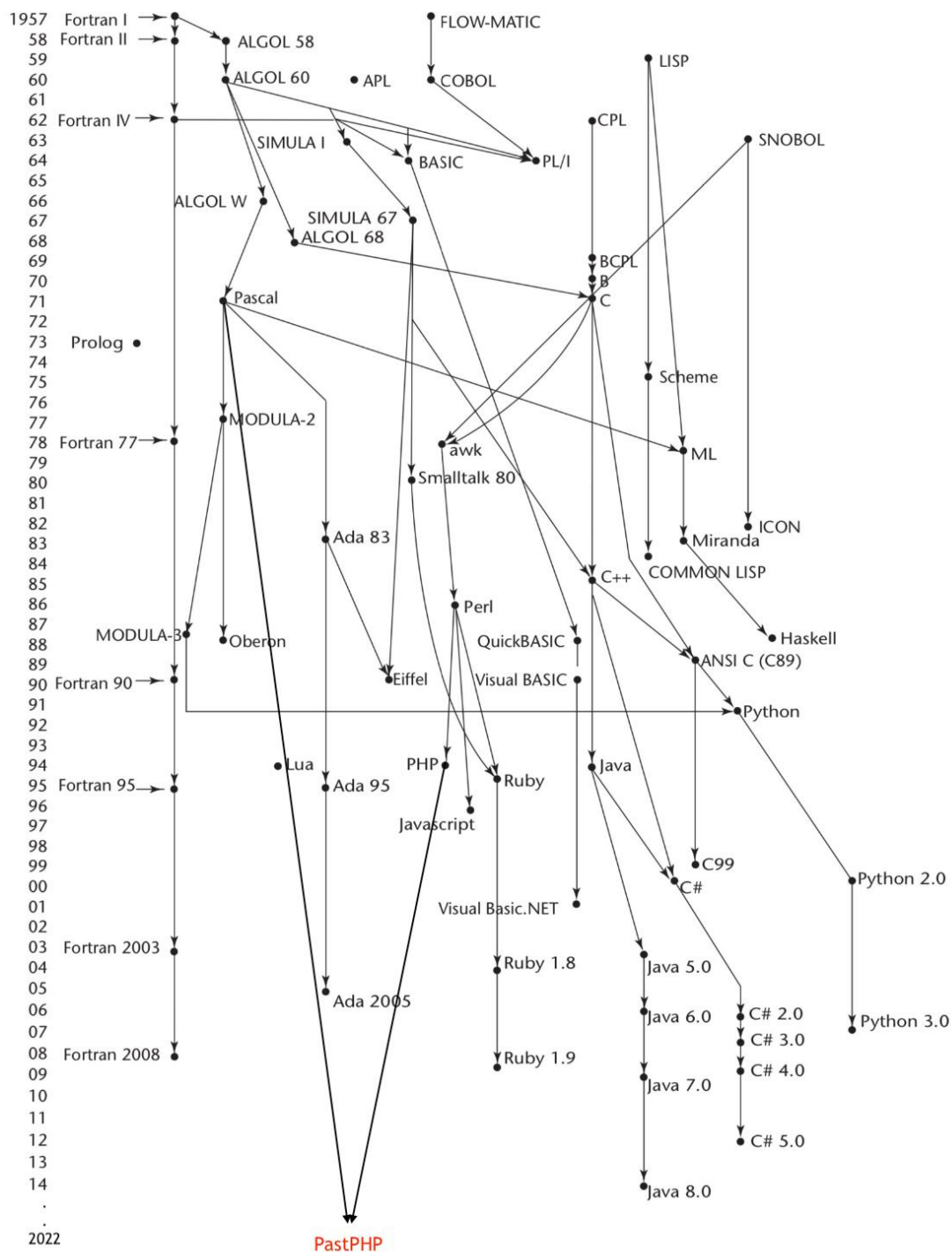CMPT 331 - Spring 2022 - Dr. Labouseur

1. Introduction

Take a blast from the past with PastPHP! Created as a strictly typed procedural programming language, PastPHP was designed by combining the favorable aspects from Pascal and PHP. The intention was to use syntax from Pascal and other languages to improve PHP and its decaying reputation in comparison to its adversary, JavaScript. Does PastPHP accomplish this? No, but its resulting syntax is intriguing and fun so let's check out in what ways PastPHP improves, and in what ways it falters.

PastPHP is more readable than PHP with its contributions from Pascal. The language is more verbose and easier to understand and recognize (maybe) but having to combine both symbols and words to make special identifiers makes the language tedious to write. This issue of writability is compounded with added strict type checking. Having to variable types, expected parameter types, return types, program names, and number of expected parameters for specified functions adds way more code to write, but then type errors will likely never happen with this added strictness.

While PastPHP is based on Pascal and PHP, if differs in the following ways:

1. Functions must be defined at the top with their expected number of parameters. This draws inspiration from Erlang.
2. Public functions are called "function" and private functions are called "method".
3. Types must be declared for every declared variable, for every incoming parameter, and for every function return.
4. Global and Local scope variables should be declared within <var > blocks.
5. Return statements receive their own <return > blocks.
6. Inputs are read with readln and outputs are performed with writeln.

1.1 Genealogy

Year	Language
1957	Fortran I
58	Fortran II, ALGOL 58
59	
60	ALGOL 60, APL, COBOL
61	
62	Fortran IV
63	SIMULA I
64	BASIC, PL/I
65	
66	ALGOL W
67	SIMULA 67
68	ALGOL 68
69	BCPL
70	B
71	Pascal, C
72	
73	Prolog
74	
75	Scheme
76	
77	MODULA-2
78	Fortran 77
79	
80	awk, Smalltalk 80, ML
81	
82	
83	Ada 83, ICON
84	Miranda
85	C++, COMMON LISP
86	
87	Perl
88	MODULA-3, Oberon, QuickBASIC, Haskell
89	ANSI C (C89)
90	Fortran 90, Eiffel, Visual BASIC
91	Python
92	
93	
94	Lua, PHP, Ruby, Java
95	Fortran 95, Ada 95
96	
97	Javascript
98	
99	C99, C#, Python 2.0
00	
01	Visual Basic.NET
02	
03	Fortran 2003
04	Ruby 1.8, Java 5.0
05	Ada 2005
06	Java 6.0, C# 2.0, Python 3.0
07	C# 3.0
08	Fortran 2008, Ruby 1.9, Java 7.0, C# 4.0
09	
10	
11	
12	C# 5.0
13	
14	Java 8.0
.	
.	
.	
2022	PastPHP

Other languages shown: FLOW-MATIC, LISP, CPL, SNOBOL

1.2 Hello World

```
1    <?PastPHP
2    <?module(HelloWorld)
3    <past
4        // Hello World example written in PastPHP
5        writeLn("Hello World!");
6    future>
7    ?>
8
```

1.3 Program Structure

The key organizational concepts in PastPHP are as follows:

1. PastPHP is strictly typed. This means variable types, function and method return types and their expected parameters all need to be declared. This allows type errors to be caught by the compiler before reaching the interpreter, thus saving run-time.
2. The less than "<" and greater than ">" operators are combined with special characters or words to create standardized identifiers. The result is unique and readable syntax using easily recognizable identifiers and wrapped sections within the signs.
3. Local scope variables are declared within "var" and initialized within "<begin" "end>".
4. Programs must declare a main method. This is done using the "<past" "future>" identifiers.
5. Programs must be wrapped within "<?PastPHP" and "?>".
6. Programs must be declared with "<?module(ProgramName)".
7. Programs must declare public functions and their expected parameter inputs.

Example Program:

```
1    <?PastPHP
2    <?module(passTheClass)
3    <?functions([passFail/1])
4
5    method avgGrades($gradeList: Arr): Float
6        <var
7            $average: Float;
8            $temp: Float;
9        >
10       <begin
11           $i := 0;
12           while(i << $gradeList.length) do
13               $temp += $gradeList[i];
14               $i++;
15           done
16           $average := $temp / $gradeList.length;
17       end>
18       <return $average>
19
20   function passFail($gradeList: Arr): Bool
21       <var
22           $passTheClass: Bool;
23       >
24       <begin
25           $passTheClass := avgGrades($gradeList);
26           if ($avgGrade << 65.0) do
27               $passTheClass := true;
28           done
29           else do
30               $passTheClass := false;
31           done
32       end>
33       <return $passTheClass>
34
```
<?example program continues next page

example program continued>

```
35    <past
36        <var
37            $numGrades: Int;
38            $grades: Arr;
39            $passed: Bool;
40        >
41        <begin
42            $numGrades := readln("How many grades would you like to enter? ");
43            for ($i:Int := 0; $i <<= $numGrades; $i++) do
44                $grades += readln("Enter Grade #$i: ");
45            done
46            $passed := passFail($grades);
47            if ($passed) do
48                writeln("After evaluation, your grade indicates that you have");
49                write(" PASSED this class! Congrats!");
50            done
51            else do
52                writeln("After evaluation, your grade indicates that you have");
53                write(" FAILED this class...");
54            done
55        end>
56    future>
57    ?>
58
```

1.4 Types and Variables

There are two types in PastPHP: value types and reference types.

1. Value type variables directly retrieves the data stored within that variable.
2. Reference type variables have indirect "references" to the data stored within that variable. An example of this would be an object variable, in which the data "referenced" is the data nested within the properties of that object.

1.5 Visibility

In PastPHP, public visibility is determined but functions being labeled as "function". Private functions are then appropriately labeled "method".

1.6 Statements Differing from Pascal and PHP

Statement	Example
Expressions	``` 1 <?PastPHP 2 <?module(ExpressionStatements) 3 <var 4 $string: Str; 5 $character: Char; 6 $integer: Int; 7 $decimal: Float; 8 $boolean: Bool; 9 $array: Arr; 10 > 11 <past 12 $string := "PastPHP is cool"; 13 $character := "P"; 14 $integer := 1991; 15 $decimal := 20.01; 16 $boolean := false; 17 $array := ["Pascal", "PHP"]; 18 future> 19 ?> ```
If	``` 1 <?PastPHP 2 <?module(IfElseIfElse) 3 <var 4 $numGrade: Float; 5 $letterGrade: Char; 6 > 7 <past 8 if ($numGrade >> 93) do 9 $letterGrade := "A"; 10 done 11 elseif ($numGrade >> 83) do 12 $letterGrade := "B"; 13 done 14 elseif ($numGrade >> 73) do 15 $letterGrade := "C"; 16 done 17 elseif ($numGrade >> 65) do 18 $letterGrade := "D"; 19 done 20 else do 21 $letterGrade := "F"; 22 future> 23 ?> 24 ```

```PastPHP
1  <?PastPHP
2  <?module(ForLoops)
3  <var
4      $grades: Arr;
5  >
6  <past
7      $grades := [91, 87, 74, 95, 85, 92];
8      for ($i:Int := 0; $i <<= $grades.length; $i++) do
9          writeln("Grade: $grades[i]");
10     done
11 future>
12 ?>
```

```PastPHP
1  <?PastPHP
2  <?module(WhileLoops)
3  <var
4      $grades: Arr;
5      $i: Int;
6  >
7  <past
8      $grades := [91, 87, 74, 95, 85, 92];
9      $i := 0;
10     while($i << $grades.length) do
11         writeln("Grade: $grades[i]");
12         $i++;
13     done
14 future>
15 ?>
16
```

```PastPHP
1  <?PastPHP
2  <?module(Comments)
3  <var
4      $a: Int;
5      $b: Int;
6      $c: Int;
7  >
8  <past
9      // Single-line comment
10     <//
11     // Muli-line Comments
12         <<///
13             /// Nested Comments
14             ///>>
15     //>
16
17     $a := 365;      // Days per Year
18     $b := 24;       // Hours per Day
19     $c := $a * $b;  // Hours per Year
20 future>
21 ?>
22
```

2.   Lexical Structure

2.1 Programs

A PastPHP program uses one or more source files which are arranged in folder structures. These files are formatted in Unicode and utilize the .pphp file extension.

2.2 Grammers

These specifications present the syntax of the PastPHP programming language where it differs from Pascal and PHP:

2.2.1 Lexical grammer (tokens) where PastPHP is different from Pascal and PHP

<Variable Operator> -> $

<Assignment Operator> → :=

<Type Operator> → :

<Mathematical Operator> → + | * | / | -

<Comparison Operator> -> <=> | <!=> | << | <<= | >> | >>=

<Input> -> readln() | read()

<Output> -> writeln() | write()

2.2.2 Syntactic ("parse") grammar where PastPHP is different from Pascal and PHP

<Program Declaration> -> <?PastPHP | ?>

<Module Declaration> -> <?module(ModName) | ?>

<functions Declaration> -> <?functions([fun1/1, fun2/2]) | ?>

<Declarations Block> -> <var | >

<Body Block> -> <begin | end>

<Return Block> -> <return | >

<Main Method Block> -> <past | future>

2.3 Lexical Analysis

2.3.1 Comments

Two forms of comments are supported: single-line comments and delimited comments:

1. Single-line comments start with the characters "//" and extend to the end of the source line.
2. Delimited comments start with the characters "<//" and end with the characters "//>". If delimited comments span multiple lines, each inner line must start with double slashes "//".
3. Nested comments start with the characters "<<///" and end with the characters "///>". If nested comments span multiple lines, each inner line must start with a single slash "///>>".

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

tokens:

- identifierkeyword
- integer-literal
- real-literal
- character-literal
- string-literal
- operator-or-punctuator

2.4.1 Keywords different from Pascal and PHP

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

New keywords:

<?PastPHP?> | <?module()?> | <?functions([])?> | <var> | <begin | end> | <return> | <past | future> | method | do done

Removed keywords:
<?php | program ProgramName | var | begin end; | do end; | return

3.   Type System

PastPHP uses a strong static type system, allowing for early binding compile-time type checking.

3.1 Type Rules

```
S ⊢ e1: T

S ⊢ e2: T

T is a primitive type

--------------------

S ⊢ e1 := e2: T



S ⊢ e1: T

S ⊢ e2: T

T is a primitive type

--------------------

S ⊢ e1 <=> e2: T



S ⊢ e1: T

S ⊢ e2: T

T is a primitive type

--------------------

S ⊢ e1 <!=> e2: T



S ⊢ e1: T

S ⊢ e2: T

T is a primitive type

--------------------

S ⊢ e1 << e2: T
```

```
S ⊢ e1: T

S ⊢ e2: T

T is a primitive type
--------------------
S ⊢ e1 >> e2: T
```

3.2 Value Types

Char: A single character.

$character: Char := "P";

Int: A whole number.

    $integer: Int := 2001;

Float: A floating point number.

$float: Float := 2.15;

Bool: A binary deciding value between true and false.

$boolean: Bool := true;

3.3 Reference Types

Str: An array of characters.

$string: Str := "PastPHP";

Arr: A mutable, comma separated collection of values with a variable length.

4.   Example Programs

4.1 Caesar Cipher

```
1    <?PastPHP
2    <?module(CaesarCipher)
3    <?functions([encrypt/2, decrypt/2])
4
5    function encrypt($toCipher: Str): Str
6        <var
7            $i: Integer;
8            $cipher: Str;
9        >
10       <begin
11           for ($i := 0; $i < $toCipher.length; $i++) do
12               // Account for uppercase letters
13               'A'..'Z': cipher[i] := chr(ord('A') + (ord(toCipher[i]) - ord('A') + shift) mod 26);
14               // Account for lowercase letters
15               'a'..'z': cipher[i] := chr(ord('a') + (ord(toCipher[i]) - ord('a') + shift) mod 26);
16           done
17       end>
18       <return $cipher>
19
20   function decrypt($ciphered: Str): Str
21       <var
22           $cipher: Str;
23       >
24       <begin
25           $cipher := encrypt($ciphered, ($shift * -1));
26       end>
27       <return $cipher>
28
```

```
29   <past
30       <var
31           // Global Declarations
32           $cipherStr: Str;
33           $shift: Int;
34       >
35       <begin
36           // Initialization
37           $cipherStr := "If my calculations are correct \n
38           when this baby hits eighty eight miles per hour \n
39           youre gonna see some serious shiii";
40           $shift := 4;
41
42           writeLn("Welcome to a Caesar Cipher in PastPHP!");
43
44           // Call Caesar Cipher Encryption Method
45           cipherStr := encrypt(cipherStr, shift);
46           Writeln('Encrypted Cipher Value: ' + cipherStr);
47
48           // Call Caesar Cipher Decryption Method
49           cipherStr := decrypt(cipherStr, shift);
50           Writeln('Decrypted Cipher Value: ' + cipherStr);
51       end>
52   future>
53   ?>
54
```

4.2 Factorial

```
1    <?PastPHP
2    <?module(Factorial)
3    <past
4        <var
5            $input: Int;
6            $factorial: Int;
7            $i: Int;
8        >
9        <begin
10           $input := readln("How many grades would you like to enter? ");
11           $factorial = 1;
12           for ($i := $input; $i >>= 1; $i--) do
13               $factorial = $factorial * $i;
14           done
15           writeln("Factorial of $input is $factorial");
16       end>
17   future>
18   ?>
19
```

4.3 QuickSort

```
1    <?PastPHP
2    <?module(QuickSort)
3    <?functions([insertionSort/2])
4
5    function insertionSort($listToSort: Arr, $n: Int): Arr
6        <var
7            $key: Float;
8            $j: Int;
9            $k: Int;
10       >
11       <begin
12           for ($j := 1; $j << $n; $j++) do
13               $key := $listToSort[$j];
14               $k := $j-1;
15
16               while ($k >= 0 && $arr[$k] > $key) do
17                   $listToSort[$k + 1] := $listToSort[$k];
18                   $k := $k - 1;
19               done
20
21               $arr[$k + 1] := $key;
22           done
23       end>
24       <return $sorted>
25
26   <past
27       <var
28           $numOfNumbers: Int;
29           $numbers: Arr;
30           $i: Int;
31           $n: Int;
32           $sortedNums: Arr;
33       >
34       <begin
35           $numOfNumbers := readln("How many numbers would you like to enter to be sorted? ");
36           for ($i := 0; $i <<= $numOfNumbers; $i++) do
37               $numbers += readln("Enter Number #$i: ");
38           done
39           $n := $arr.length;
40
41           $sortedNums := insertionSort($numbers, $n);
42
43           for($i := 0; $i << $n; $i++) do
44               write("$sortedNums[i], ");
45           done
46       end>
47   future>
48   ?>
```

```
<?quicksort program continues next page
```

quicksort program continued>

```
1    <?PastPHP
2    <?module(QuickSort)
3    <?functions([quickSort/3])
4
5    method partition($listToSort: Arr, $left: Int, $left: Int): Arr
6        <var
7            $pivotIndex: Int;
8            $pivotValue: Float;
9            $j: Int;
10           $k: int;
11       >
12       <begin
13           $pivotIndex := floor($left + ($right - $left) / 2);
14           $pivotValue := $listToSort[$pivotIndex];
15           $j := $left;
16           $k := $right;
17           while ($j <<= $k) do
18               while (($listToSort[$j] << $pivotValue) ) do
19                   $i++;
20               done
21               while (($listToSort[$k] >> $pivotValue)) do
22                   $k--;
23               done
24               if ($j <= $k ) do
25                   $temp := $listToSort[$j];
26                   $listToSort[$j] := $listToSort[$k];
27                   $listToSort[$k] := $temp;
28                   $j++;
29                   $k--;
30               done
31           done
32       end>
33       <return $listToSort & $j>
34
```

<?quicksort program continues next page

quicksort program continued>

```
35    function quickSort($listToSort: Arr, $left: Int, $left: Int): Arr & Int
36        <var
37            $pivot: Int;
38            $list: Arr;
39            $sorted: Arr;
40        >
41        <begin
42            if($left < $right) {
43                $list & $pivot := partition($listToSort, $left, $right);
44                $sorted := quicksort($list, $left, $pivot-1);
45                $sorted := quicksort($list, $pivot, $right);
46            done
47        end>a
48        <return $sorted>
49
50    <past
51        <var
52            $numOfNumbers: Int;
53            $numbers: Arr;
54            $i: Int;
55            $sortedNums: Arr;
56        >
57        <begin
58            $numOfNumbers := readln("How many numbers would you like to enter to be sorted? ");
59            for ($i := 0; $i <<= $numOfNumbers; $i++) do
60                $numbers += readln("Enter Number #$i: ");
61            done
62
63            $sortedNums := quicksort($numbers, 0, ($numbers.length)-1);
64
65            for($i := 0; $i << $sortedNums.length; $i++) do
66                write("$sortedNums[i], ");
67            done
68        end>
69    future>
70    ?>
71
```

4.3 BubbleSort

```
1    <?PastPHP
2    <?module(BubbleSort)
3    <?functions([bubbleSort/2])
4
5    function bubbleSort($listToSort: Arr): Arr
6        <var
7        $j: Int;
8        $k: Int;
9        $temp: Int;
10       >
11       <begin
12           for($j := 0; $j << $listToSort.length; $j++) do
13               for ($k := 0; $k < $listToSort.length - $i - 1; $k++) do
14                   if ($listToSort[$j] > $listToSort[$j+1]) do
15                       $temp = $listToSort[$j];
16                       $listToSort[$j] = $listToSort[$j+1];
17                       $listToSort[$j+1] = $temp;
18                   done
19               done
20           done
21       end>
22       <return $listToSort>
23
24   <past
25       <var
26           $input: Int;
27           $factorial: Int;
28           $i: Int;
29       >
30       <begin
31           $numOfNumbers := readln("How many numbers would you like to enter to be sorted? ");
32           for ($i := 0; $i <<= $numOfNumbers; $i++) do
33               $numbers += readln("Enter Number #$i: ");
34           done
35
36           $sortedNums := bubbleSort($numbers);
37
38           for($i := 0; $i << $sortedNums.length; $i++) do
39               write("$sortedNums[i], ");
40           done
41       end>
42   future>
43   ?>
44
```