<?PastPHP>

Caleb Rogers

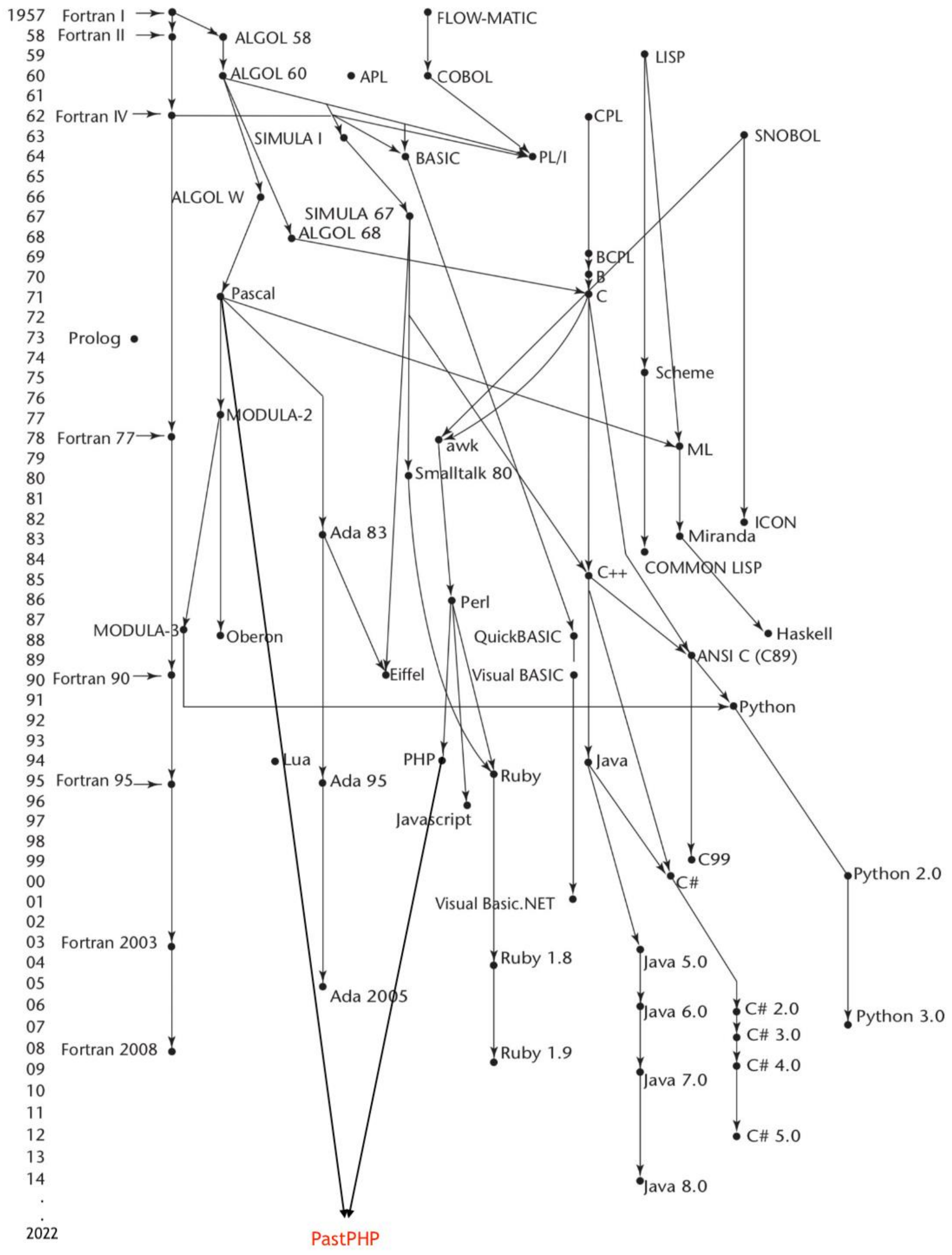
CMPT 331 - Spring 2022 - Dr. Labouseur

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.

1.1 Genealogy



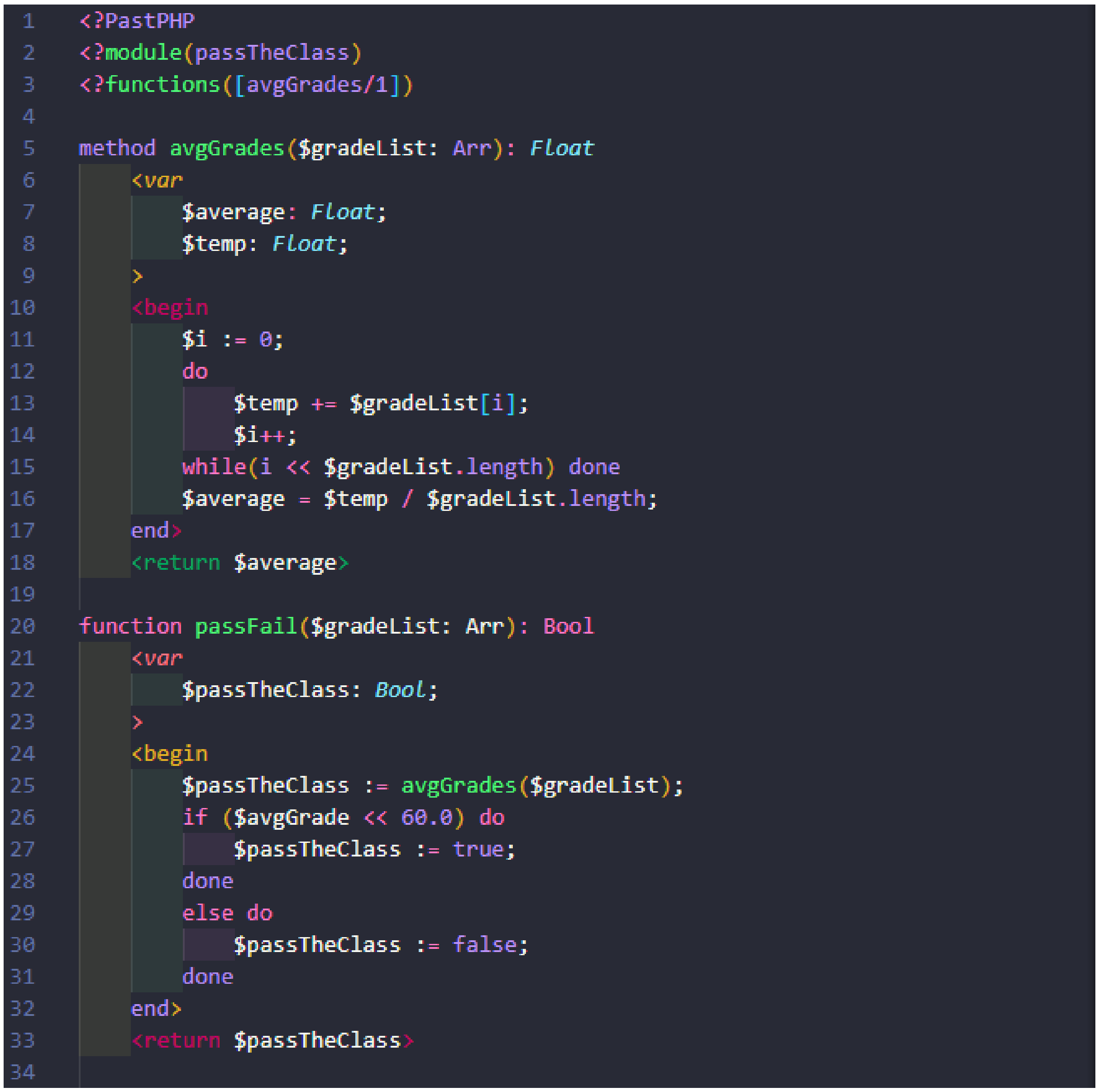
1.2 Hello World

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.

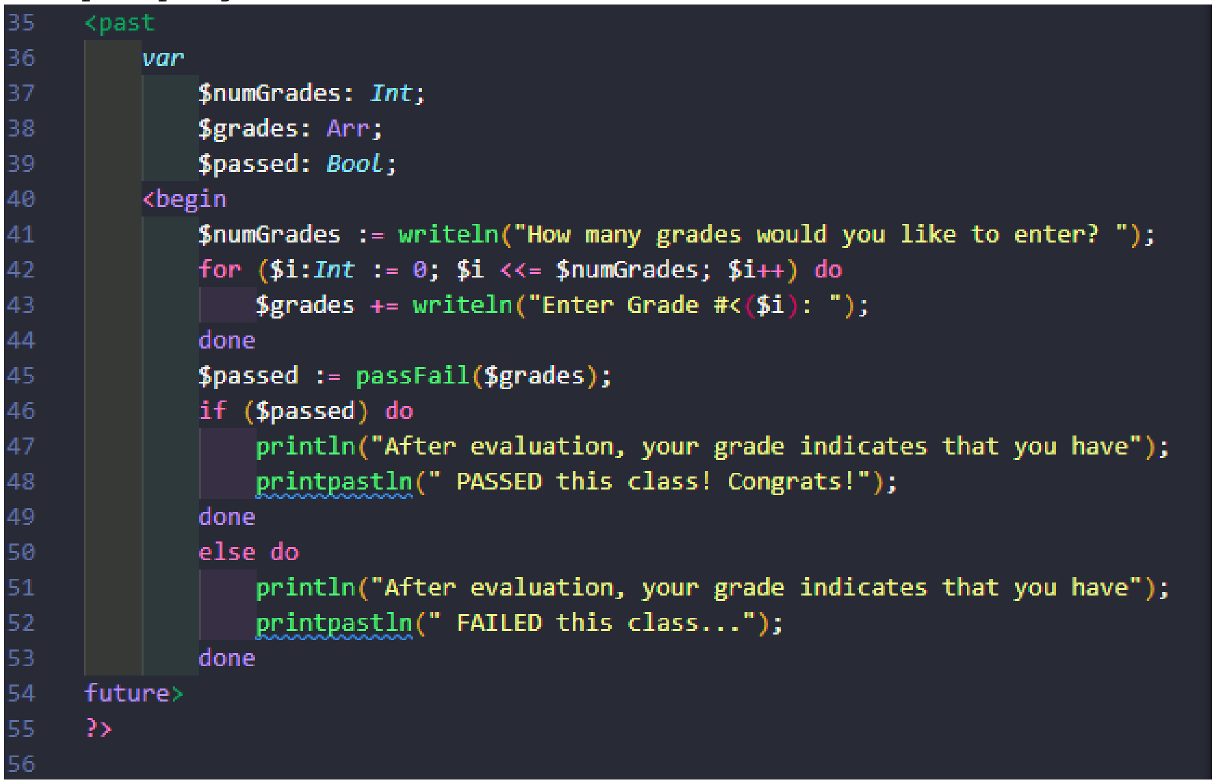
Example Program:



```
<?PastPHP
<?module(passTheClass)
<?functions([avgGrades/1])

method avgGrades($gradeList: Arr): Float
    <var
        $average: Float;
        $temp: Float;
    >
    <begin
        $i := 0;
        do
            $temp += $gradeList[i];
            $i++;
        while(i << $gradeList.length) done
        $average = $temp / $gradeList.length;
    end>
    <return $average>

function passFail($gradeList: Arr): Bool
    <var
        $passTheClass: Bool;
    >
    <begin
        $passTheClass := avgGrades($gradeList);
        if ($avgGrade << 60.0) do
            $passTheClass := true;
        done
        else do
            $passTheClass := false;
        done
    end>
    <return $passTheClass>

```

<?example program continues next page

example program continued>



```
<past
    var
        $numGrades: Int;
        $grades: Arr;
        $passed: Bool;
    <begin
        $numGrades := writeln("How many grades would you like to enter? ");
        for ($i:Int := 0; $i <<= $numGrades; $i++) do
            $grades += writeln("Enter Grade #<($i): ");
        done
        $passed := passFail($grades);
        if ($passed) do
            println("After evaluation, your grade indicates that you have");
            printpastln(" PASSED this class! Congrats!");
        done
        else do
            println("After evaluation, your grade indicates that you have");
            printpastln(" FAILED this class...");
        done
future>
?>
```

1.4 Types and Variables

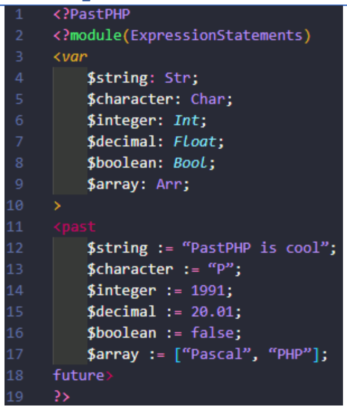
There are two types in PastPHP: value types and reference types.

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

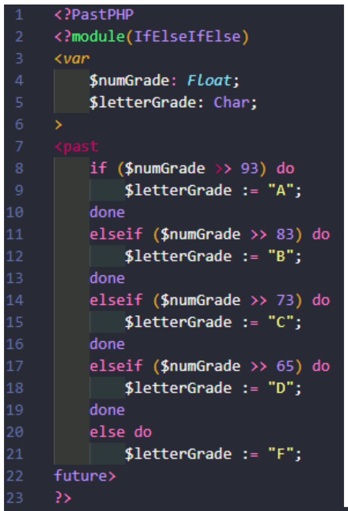
In PastPHP, public visibility is determined but functions being labeled as "function". Private functions are then appropriately labeled "method".

1.6 Statements Differing from Pascal and PHP

Statement	*Example*
Expressions	

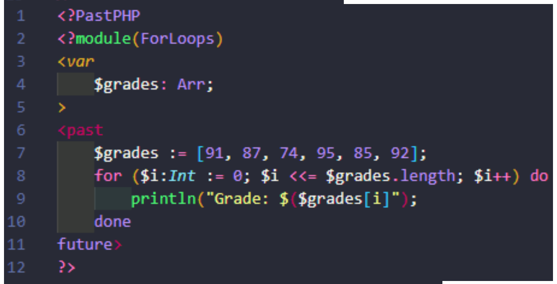
```
<?PastPHP
<?module(ExpressionStatements)
<var
    $string: Str;
    $character: Char;
    $integer: Int;
    $decimal: Float;
    $boolean: Bool;
    $array: Arr;
>
<past
    $string := "PastPHP is cool";
    $character := "P";
    $integer := 1991;
    $decimal := 20.01;
    $boolean := false;
    $array := ["Pascal", "PHP"];
future>
?>
```

If



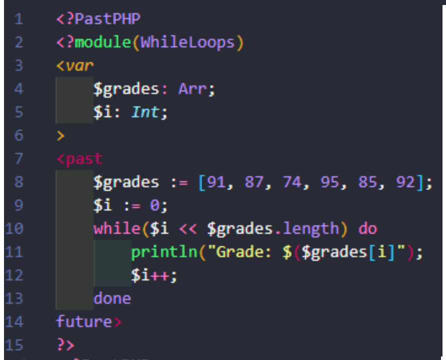
```
<?PastPHP
<?module(IfElseIfElse)
<var
    $numGrade: Float;
    $letterGrade: Char;
>
<past
    if ($numGrade >> 93) do
        $letterGrade := "A";
    done
    elseif ($numGrade >> 83) do
        $letterGrade := "B";
    done
    elseif ($numGrade >> 73) do
        $letterGrade := "C";
    done
    elseif ($numGrade >> 65) do
        $letterGrade := "D";
    done
    else do
        $letterGrade := "F";
future>
?>
```

For



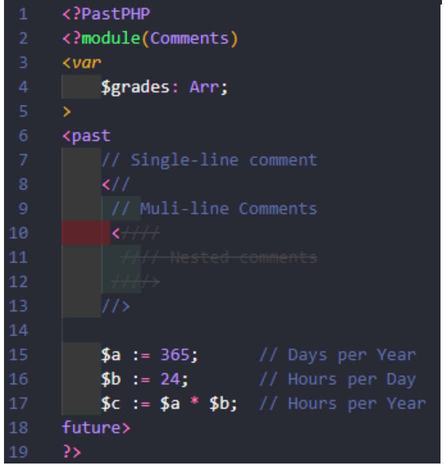
```
<?PastPHP
<?module(ForLoops)
<var
    $grades: Arr;
>
<past
    $grades := [91, 87, 74, 95, 85, 92];
    for ($i:Int := 0; $i <<= $grades.length; $i++) do
        println("Grade: $($grades[i]");
    done
future>
?>
```

While



```
<?PastPHP
<?module(WhileLoops)
<var
    $grades: Arr;
    $i: Int;
>
<past
    $grades := [91, 87, 74, 95, 85, 92];
    $i := 0;
    while($i << $grades.length) do
        println("Grade: $($grades[i]");
        $i++;
    done
future>
?>
```

Comments



```
<?PastPHP
<?module(Comments)
<var
    $grades: Arr;
>
<past
    // Single-line comment
    <//
     // Muli-line Comments
     <////
      //// Nested comments
     ////>
    //>

    $a := 365;      // Days per Year
    $b := 24;       // Hours per Day
    $c := $a * $b;  // Hours per Year
future>
?>
```

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> -> <=> | <!=> | << | <<= | >> | >>=

<Output> -> println() | printpastln() | writeln() | writelastln()

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

. . .