Phoenix Programming Language:
User's Manual

Jonah Scheinerman

Contents

1 Introduction

The Phoenix programming language is an interpreted language written in the Java $^{\text{TM}}$ S.E. 1.5 programming language[1]. It is designed to be easy to use and to understand and to follow standard syntactical programming conventions.

1.1 Why Use Phoenix?

Phoenix by itself is not of great use. It is relatively slow, and is procedural, not object-oriented, which limits its usefulness in modern programming. However, this does not mean that Phoenix is without merit. Phoenix is meant primarily for application developers.

Phoenix is meant to be used as a scripting language for Java applications. Since Phoenix is written in Java, it can be used easily from within Java. The accompanying techinical guide (discussed in section 1.2) provides detailed information on how to integrate Phoenix programs into a Java application.

Phoenix would be useful to a Java application in which the developer wants to allow the user of the application a significant amount of flexibility. The developer can only grant a limited spectrum of flexibility through a graphical user interface of menus and buttons. There are certain situations where it would be necessary to write small sections of code to serve simple functions in the application. That is where Phoenix comes in. Phoenix is easy to use from within Java, and it is a simple matter to call a Phoenix interpretation of a file for the purposes of the application. Not only is it convenient, but Phoenix can be tailored to serve the purposes of the application. It is possible and easy to make Phoenix modules in Java that will be included in every Phoenix program as built-in functions and will provide application-specific functions that modify how the application runs.

For more information on why Phoenix is a good idea for use in Java applications, and how to use it, see the accompanying technical guide to Phoenix.

1.2 A Guide to This Guide

This guide is meant for someone attempting to learn the Phoenix programming language. Once you have learned the basic syntax of Phoenix (which should be relatively simple if you know any other languages), I encourage you to view my technical guide to Phoenix, if you are so inclined. The technical guide provides the reader with an understanding of the inner workings of Phoenix. The technical guide also provides developers with information on how to integrate Phoenix into one of their projects.

For your edification, the following is an overview of the sections of this guide, each with a brief blurb that describes what their about.

- **Getting Started** — Describes the very basics of getting started with Phoenix. Discusses how to run Phoenix programs and what the basic syntax is like.

---

[1] Java and all Java related trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S., other countries, or both.

- **Variables** — Describes the diffrent types of variables in Phoenix, how to create and manipulate them. Discusses the specific operators for each variable type, talks about variable scope, and provides naming conventions.
- **Conditionals** — Describes if / else if / else statements and switch statements in Phoenix, structures which decide what code will be executed under certain conditions.
- **Loops** — Describes the different types of loops in Phoenix. Loops allow code to be repeated until certain conditions are met.
- **Functions** — Describes how to create and use functions. Functions are code blocks that take arguments and return values.
- **Modules** — Describes how to create and import modules. Modules are Phoenix programs which contain groups of functions.

Hopefully this gives some sense of the contents of the manual. Good luck and enjoy!

2 Getting Started

2.1 The Basics

2.2 Syntax

The phoenix language syntax is quite similar to Python syntax. The code has a basic structure. Each line of the input file is a separate line of code. A single line of code cannot be split onto separate lines (i.e. no backslash concatination like in Python) nor can multiple lines of code be be combined onto a single line (i.e. no semicolon delimited lines). Like Python, Phoenix is indentation sensitive. Thus, all lines of code with the same scope must have the same amount of whitespace in front of them. On the first line of code there should be no indentation. On the lines where there is greater scope, higher indentation should be used. Besides this, Phoenix is whitespace insensitive. It does not matter how much whitespace occurs between or after keywords, literals and operators.

2.3 Comments

Comments are lines of code that are skipped over by the interpreter. Comments can either be whole lines are subsections of single lines. To make a full or partial line comment, insert two slashes (`//`). Everything before the slashes are interpreted as code. Everything after the slashes is a comment and is not interpreted. Comments cannot be within strings (see section 3.2). The following are fine uses of comments:

```
// Comment ...
Code ... // Coment ...
```

To put a comment in the middle of a line, start the comment with a slash followed by an asterix (`/*`) and end the comment with a asterix followed by a slash (`*/`). Everything between the two will become a comment. The code on either side of the comment will become concatenated together. Any whitespace surounding these comments will be preserved. These types of comments can only span a single line, not multiple lines. The following are allowed uses of in-line comments:

```
/* Comment ... */ Code ...
Code ... /* Comment ... */
Code ... /* Comment ... */ More code ...
```

3 Variables

Variables in the Phoenix language are similar to variables in other languages. Variables hold data and can be manipulated through the use of operators such as `+`, `-`, `*`, or `/`. There are two different variable types in Phoenix: numbers and strings. Numbers are positive or negative floating-point numbers and strings are groups of characters (text). All variables are declared in one of the following ways:

```
type name = value
type name
```

The first declaration creates a variable and assigns `value` to be the variables stored value. The second declaration creates a value with a default value. The default value for numbers and strings `0` and `""` (the empty string), respectively.

For the most part, one can give variables any desired name. Their are only four restrictions. The first is that variable must not be able to be confused with number literals. This means that variables must consist of other symbols besides numbers and periods. The second restriction is that the name can't include any illegal symbols. These symbols include all operators and parentheses. The third restriction is that the name cannot be equal to one of the Phoenix reserved keywords. These words can be found in appendix A. The last restriction is that the name cannot be the same as the name of an existing variable.

Variable names are exclusive and cannot be repeated. This means that after defining a variable with a certain name, it can be used, but it cannot be made again. If necessary, variables can be deleted by using the `delete` keyword. This is accomplished through the following code:

```
delete name
```

The following two sections cover the each of the two variable types.

3.1 Number Variables

A number variable is a variable that represents a real number that can be positive, negative, or zero and hold decimal values. For example `-1.2`, `220`, `2.718281828`, and `-341.3` are all number values. These are examples of number literals. A number is referred to by the keyword `num`. To declare a number variable one can do this:

```
num n = value
```

In this instance, `n` is the name of the variable and `value` is the value stored in the variable. If no value is provided, the value of the variable will default to zero. The value that is provided in this case can be a number of things. It can be a variable, an literal, or an expression with multiple variables and literals that ultimately results in a number variable.

Numbers also provide the basis of Boolean (true / false) logic in Phoenix. There is no Boolean variable type, so numbers fill the role through zero and non-zero based logic. If a number variable is non-zero it is assumed to be true. If a number variable is zero then it is false. This becomes particularly important when using conditionals and loops (sections 4 and 5, respectively).

3.1.1 Number Operators

The following is a list of operators that can be used with number variables. For each of the operators that follow, the operator will either take one or two arguments. If it takes two arguments, then the first will be a number. If it takes one argument, that will be a number.

- **Addition** (`x+y`). The addition operator takes a number variable on either side and returns a number variable as the result. This provides basic number addition. For example, `3 + 4` returns `7`.
- **Subtraction** (`x-y`). The subtraction operator takes a number on either side and returns a number as the result. This provides basic number subtraction. For example, `4 - 3` returns `1`.
- **Multiplication / Repetition** (`x*y`). The multiplication operator takes a number on either side and returns a number as the result. This provides basic number multiplication. For example, `3 * 4` returns `12`. This also provides the repetition functionality of strings in the expression `n * s`, where `n` is a number and `s` is a string. See section 3.2.1 for more information on this operation.
- **Division** (`x/y`). The division operator takes a number on either side and returns a number as the result. This provides basic number division. For example, `3 / 4` returns `0.75`.
- **Modulus Arithmetic** (`x%y`). The modulus operator takes a number on either side and returns a number as the result. The modulus of two numbers is the remainder of the quotient of the numbers. For example `9 % 5` is equal to `4` because the remainder of 9 / 5 is 4.
- **Exponentiation** (`x^y`). The exponentiation operator takes a number on either side and returns a number as the result. The exponentiation operator raises one number to the power of another number. For example `4 ^ 3` results in `64` because $4^3 = 64$.
- **Rounding** (`x#y`). The rounding operator takes a number on either side and returns a number as the result. The rounding operator will round the number on the left to the nearest multiple of the number on the right.
- **Equal To** (`x==y`). The equality operator takes a number on either side and returns a number, either one or zero as the result. The operator returns one if the two sides

are equal and zero if the two sides are not equal. For example, `1 == 1` returns one (true), whereas `1 == 0` returns zero (false). **Note:** This is not to be confused with the assignment operator (`=`). The assignment operator cannot be used in place of the equality operator.

- **Not Equal To** (`x!=y`). The inequality operator takes a number on either side and returns a number, either one or zero as the result. The operator returns one if the two sides do not equal and zero if the two sides are equal to each other. For example, `1 != 0` returns one (true), whereas `1 != 1` returns zero (false).
- **Greater Than** (`x>y`). The greater than operator takes a number on either side and returns a number, either one or zero, as the result. The operator returns one if the left hand operand is greater than the right and zero otherwise. For example, `5 > 2` returns one (true), and both `3 > 4` and `3 > 3` return zero (false).
- **Greater Than or Equal To** (`x>=y`). The greater than or equal to operator takes a number on either side and returns a number as the result. The operator returns one if the left hand operaand is greater than or equal to the right and zero otherwise. For example, the expressions `3 >= 1` and `2 >= 2` return one (true) and the expression `5 >= 9` returns zero (false).
- **Less Than** (`x<y`). The less than operator takes a number on either side and returns a number, either one or zero, as the result. The operator returns one if the left hand operand is less than the right and zero otherwise. For example, `2 < 7` returns one (true), and both `4 < 3` and `3 < 3` return zero (false).
- **Less Than or Equal To** (`x<=y`). The less than or equal to operator takes a number on either side and returns a number, either one or zero, as the result. The operator returns one if the left hand operaand is less than or equal to the right and zero otherwise. For example, the expressions `1 <= 3` and `2 <= 2` return one (true) and the expression `9 <= 5` returns zero (false).
- **Logical And** (`x&y`). The logical and operator takes a number on either side and returns a number, either one or zero, as the result. The operator returns the logical and of the two numbers using one and zero logic. This means that if the two numbers are both true (non-zero), the result will be one (true), otherwise the result will be zero (false). Following is a table of all possible results of use of the logical and operator. **Note:** All `1`'s in the `p` and `q` columns refer to non-zero numbers.

p	q	p & q
1	1	1
1	0	0
0	1	0
0	0	0

- **Logical Inclusive Or** (`x|y`). The logical inclusive or operator takes a number on either side and returns a number, either one or zero, as the result. The operator returns the logical inclusive or of the two numbers using one and zero logic. This means that if either of the two numbers is true, the result will be true. Following is a table of all possible results of use of the logical and operator. **Note:** All `1`'s in the `p` and `q` columns refer to non-zero numbers.

p	q	p | q
1	1	1
1	0	1
0	1	1
0	0	0

- **Logical Exclusive Or** (`x(+)y`). The logical exclusive or operator takes a number on either side and returns a number, either one or zero, as the result. The operator returns the logical exclusive or of the two numbers using one and zero logic. This means that if the one side is true and the other side is false, the result will be true. Following is a table of all possible results of use of the logical and operator. **Note:** All `1`'s in the `p` and `q` columns refer to non-zero numbers.

p	q	p (+) q
1	1	0
1	0	1
0	1	1
0	0	0

- **Logical Complement** (`!x`). The logical complement operator takes only one operand on the right side. The operator returns the logical opposite of the number. This means that if the number is true it will return false and visa versa. Following is a table of all possible results of use of the logical and operator. **Note:** All `1`'s in the `p` column refer to non-zero numbers.

p	!p
1	0
0	1

- **Postfix Increment** (`x++`). The postfix increment operator takes one operand that must be a declared variable, it cannot be a literal variable. The operator will increase the value of this value by one and assign the new value to the variable. The returned value of this operation will be the original value of the variable, before it was incremented.

- **Postfix Decrement** (`x--`). This postfix decrement operator is the same as the postfix increment operator except for that the operand is decreased by one rather than increased by one.
- **Prefix Increment** (`++x`). The prefix increment operator takes one operand that must be a declared variable, it cannot be a literal variable. The operator will increase the value of this value by one and assign the new value to the variable. The returned value of this operation will be the new value of the variable, after it is incremented.
- **Prefix Decrment** (`--x`). This prefix decrement operator is the same as the prefix increment operator except for that the operand is decreased by one rather than increased by one.

For a brief list of all operators, see appendix B. Appendix B also gives the precedence of the operators, describing the order of operations.

3.2 String Variables

A string variable is a variable that represents a sequence of alphabetical characters, numbers and symbols. All string literals are surrounded by double quote (`"`) characters. String literals have certain restrictions on their contents. For instance, since all string literals start and end with double quotes, no double quote characters are allowed in strings. However a double quote can be "escaped" by typing `\"`. The backslash (`\`) character, provides "escape" sequences out of string literals. These escape sequences consist of a `\` followed by another character to produce another character that is not normally allowed in a string. These are the escape sequences and what they produce:

Sequence	**Result**	**Sequence**	**Result**
`\\`	Backslash	`\"`	Double quote
`\n`	New line	`\r`	Carriage return
`\t`	Horizontal tab	`\a`	Bell (alarm) sound

For example, all of the following are valid string literals: `"Hello!"`, `"Where? \nThere!"`, or `"\tParagraph..."`.

Strings are referred to by the keyword `str`. Thus the declaration of a string variable could be:

```
str s = value
```

In this case, the name of the variable would be `s` and it would be assigned the value held by `value`. The value could be several things. It could be a string literal. It could be another variable. It could also be a combination of variables and literals which combine to make a variable.

3.2.1 String Operators

The following is a list of operators that can be used with string variables. For each of the operators that follow, the operator will take two or more arguments, the first of which will be a string.

- **Concatenation** (`x+y`). The concatenation operator takes a string on either side and returns a string as the result. The concatenation operator concatenates two strings together so that end of the first is attached to the beginning of the second. For example, `"That is " + "a dog."` is equal to `"That is a dog."`.
- **Repetition** (`x*y`). The repetition operator takes a string on the left and a number on the right and returns a string. This operator repeats a string a certain number of times. On the left of the operator is the string, and on the right of the operator is the number of repeats, a number variable which should be an integer. If it has any decimal, it will cause an error. For example, the expression `"ab " * 4` is equal to `"ab ab ab ab "`. However, the expression `"ba" * 3.5` will cause an error.
- **Equal To** (`x==y`). The equality operator takes a string on both sides and returns a number, one or zero. The equality operator returns a number which indicates whether the strings on either side of the operator are equal to each other. If the two sides are equal, the number one (true) is returned. If the two sides are unequal, the number zero (false) is returned. For example, the expression `"abc" == "abc"` returns one, whereas `"abc" == "xyz"` returns zero.
- **Not Equal To** (`x!=y`). The inequality operator takes a string on both sides and returns a number, one or zero. The inequality operator returns a number which indicates whether the strings on either side of the operator are not equal to each other. If the two sides are not equal, the number one (true) is returned. If the two sides are equal, the number zero (false) is returned. For example, the expression `"cat" != "dog"` returns one, whereas `"abc" != "abc"` returns zero.
- **Greater Than** (`x>y`). The greater than operator takes a string on both sides and returns a number, one or zero. The operator preforms a lexicographical comparison of the two strings and returns either one or zero. The operator returns one if the left hand operand comes alphanumerically after the right hand operand and returns zero otherwise. For example, the expressions `"ABC" > "abc"`, `"x" > "o"`, and `"ac" > "ab"` are all true. However, the expressions `"abc" > "abd"` and `"x" > "x"` are both false.
- **Greater Than or Equal To** (`x>=y`). The greater than or equal to operator takes a string on both sides and returns a number, one or zero.
- **Less Than** (`x<y`). The less than operator takes a string on both sides and returns a number, one or zero.

- **Less Than or Equal To** (`x<=y`). The less than or equal to operator takes a string on both sides and returns a number, one or zero.
- **Subscript / Slice** (`x[y:z]`). The subscript / slice operator is one of most complicated operators in Phoenix because it has so many forms. There are four main forms of this operator. These are covered below:
  - **Single Subscript** (`x[num]`). The single subscript operator takes a single number argument inside the brackets. This number should be an integer which should be between 0 and the length of the string minus one. Zero is the first index in the string, and the length of the string minus one is the last index. This operation will return the character at that location. For example evaluating `"Hello"[0]` will return `"H"`. Evaluating `"Test"[2]` will return `"s"`. The index can also be a negative number between the negative length of the string plus one and zero. Negative indices will start at the *end* of the string and go backward. For example, `"Hello"[-1]` evaluates as `"o"` and `"Test"[-3]` evaluates as `"e"`.
  - **Slice** (`x[num:num]`). The slice operator takes two number variables (which must be integers) and returns the section of the string from the index of the first variable up until, but not including the second index. For example, evaluating `"Hello!"[0:3]` returns `"Hel"` and `"What?"[2:3]` returns `"a"`. The indices have the same restrictions as those of the single subscript operator: They have to be less than the length minus one and greater than the negative length plus one. Negative numbers have the operate in the same way. For example the result of `"Hello!"[2:-2]` returns `"ll"`. If the first number is removed (`x[:z]`), then the result will be the beginning of the string up until (but not including) the second index. If the second number is removed (`x[y:]`), then the result will be the first index until the end of the string. For example, `"Who?"[:2]` returns `"Wh"` and `"Who?"[2:]` returns `"o?"`.
  - **First Index** (`x[str:num]`). The first index operator returns the first index of the string argument in the string starting after a given index. For example, the result of evaluating `"Hello!["l":0]"` would be `2` because the the first `"l"` in `"Hello!"` is at index 2. The result of `"Hello!"["l":2]` is still `2`, because the search starts at index 2 where there is an `"l"`. However, `"Hello!"["l":3]` returns `3`. The same rules of bounding apply to the index variable. Therefore, `"his hip"["h":-4]` return `4`. If the search string can't be found after the given index, the value `-1` is returned. If the index number is omitted from the operator, the search will start at the beginning of the string. Therefore `string[search:0]` is equivalent to `string[search:]`.
  - **Last Index** (`x[num:str]`). The last index operator returns the last index of the string argument in the string before or including the given index. For example, the result of evaluating `"Hello!"[-1:"l"]` is `3` because the last `"l"` is at index 3. However, `"Hello!"[2:"l"]` returns `2`. As with the first index operator, omitting

number variable is permissible. Omitting the index starts the search from the end of the string. Therefore `string[-1:search]` is equivalent to `string[:search]`.

- **Function Reference** (`@x`). See section 6.4 for more information on this operator.

For a brief list of all operators, see appendix B. Appendix B also gives the precedence of the operators, describing the order of operations.

3.3 Combined Assignment Operators

To assign the value of a variable, the equals operator (`=`) is used. Thus, to set the value of an already declared number variable `n` to 3.1, the following would be the code:

```
n = 3.1
```

Some times it is necessary to assign the value of a variable to itself, with a simple modification. For example, to increase the value of `n` by 5, the following code would work:

```
n = n + 5
```

Although this works, Phoenix provides a special kind of assignment operator to deal with cases such as this. This consists of the operation that is to be performed, accompanied by an equal sign. For example, the above code could be replaced by:

```
n += 5
```

There are seven combined assignment operators for seven operators in Phoenix, example cases, with there equivalent using the traditional operators are given below:

Standard	**Combined**	**Standard**	**Combined**
`x = x + y`	`x += y`	`x = x - y`	`x -= y`
`x = x * y`	`x *= y`	`x = x / y`	`x /= y`
`x = x % y`	`x %= y`	`x = x ^ y`	`x ^= y`
`x = x # y`	`x #= y`		

For a brief list of all operators, see appendix B. Appendix B also gives the precedence of the operators, describing the order of operations.

3.4 Variable Scope

Variable scope refers to where a variable can be seen in a program. This section will talk about basic variable scope. In Phoenix anytime some control structure occurs (described in section 4), the code is indented a single time. Every where that code is indented the same amount or more is called a block. Obviously, a variable is only usable in the code **after** it is defined (you can't use a variable before you've created the variable). This means that for some time after that variable declaration, that variable can be used. Also, a variable is usable in any subblock of the current block. This means that anywhere the indentation is greater, a variable is accesible. However, if the current block is exitted (the indentation is reduced), the variable is destroyed and is no longer accessible. Consider the following pseudo-code:

```
// Block A
Begin block B:
    // Block B
    Begin block C:
        // Block C
    // End block C
// End block B

Begin block D:
    // Block D
// End block D
```

In the above pseudo-code, a variable that is declared at the beginning, in block `A` would be visible for the rest of the code, because nothing else is of lower indentation (block `A` never goes out of scope). However, a variable declared in block `B` would only be visible in block `B` and block `C`. Even though block `D` is of the same indentation as block `B`, the variable wouldn't be accessible in block `D`. This is because before `D` can be reached, `B` goes out of scope, back up to `A`. At this point, all variables declared in `B` are destroyed. Similarly, when `C` goes out of scope, back to block `B`, all variables declared in `B` are destroyed.

The only reason a variable would not be destroyed by going out of scope would be if it is a global variable. A global variable is defined by adding the keyword `global` to the beginning of a variable declaration (e.g. `global num n = 0`). By adding `global`, the variable becomes scopeless, in other words, from then on, the variable is accessible in the rest of the program. A global variable cannot be destroyed by scope changes. The only way to destroy a global variable is to use the `delete` keyword (see section 3.1).

3.5 Constant Variables

Constant variables are variables that cannot be modified. A constant variable is declared by adding the keyword `const` before the variable declaration (e.g. `const num n = 0`). If a variable is constant and global (section 3.4), both the `const` and `global` keywords go before the variable declartion, but it does not matter in what order they appear.

Any variable that has been declared as constant can only be set on its initial creation, and after that it can never be modified. Thus any of the assignment operators (`=`,`+=`,`-=`,`*=`,`/=`,`%=`,`^=`, or `#=`), can not be used on that variable (these modify the variable). If such an operator is used on a constant variable, an error occur.

All of the following are valid constant variable declarations:

```
const num STOP = 2
global const num _PI = 3.141592645
const global num _E = 2.718281828
```

3.6 Naming Conventions

In Phoenix, when setting variable names, the following conventions are recommended. While these are not required by the interpreter, they are preferred for readability purposes.

- All variable names begin with a lowercase letter.
- If a variable name contains multiple words, these words are camel cased together. This means that the first letter is lowercase, and only the first letter of each of the subsequent words are uppercase. For example: `myNewVariable` or `doNotTouch`.
- If a variable name contains multiple words, the camel casing method is preferred over concatenating words together using underscores. For example: `myNewVariable` instead of `my_new_variable`.
- For constant variable names, all letters should be uppercase.
- For constant variable names with multiple words, the words should be concatenated together using underscores. For example: `MY_CONST_VAR` or `DO_NOT_TOUCH`.
- For variables which a copies of other variables, the phrase “skis” (pronounced sk’eez) should be appended to the end of the name. For example: `doNotTouch` and `doNotTouchskis`.
- For any variable this serves a temporary position, the following variable names are preferable: `whatevskis` or `tempskis`.

4 Conditionals

Conditionals are the first control stucture described in this manual. Control structures are blocks of code that determine how program control flows. Everything within a block of code must be indented once more than the block above it. Within each block a new variable scope is defined. For more information on variable scope see section 3.3.

The two types of control structures detailed in this section are conditional structures. Conditional structures a structures that do or do not execute certain sections of code based on the truth of various statements. Recall that in Phoenix, boolean values are zero (false) and non-zero (true). This will be important for conditionals.

4.1 If / Else If / Else

The if/else if/else structure is a simple control structure. The simplest form structure determines whether an initial statement is true or false. **If** it is true, it executes the block of code, otherwise it exits. This is a simple if statement. A simple if statement takes the following form:

```
// ... Some code ...
if condition:
    // ... Code to do if the condition is true ...
// ... More code ...
```

For example, suppose you have some variable, `n`. and you want to print it out if its between, 3 and 10, inclusive. You would do the following:

```
num n = // define num n somehow...
if n >= 3 & n <= 10:
    println("n=" + n) // print out the value of n
println("Done.")
```

In if statements, only the indented code section is executed conditionally. In the example above, whether or not the statement `println("n=" + n)` is executed depends on the value of `n`. However, the statement `println("Done.")` is always executed because it is outside of the if block because it is not indented.

In some cases, you might want to have alternate code execute if the condition is false. This is accomplished through an else statement. This is an additional clause that is added to the end of the if block, and the block of code following the `else:` statement is executed if the initial condition is false. A if/else construct takes the following form:

```
// ... Some code ...
if condition:
    // ... Code to do if the condition is true ...
else:
    // ... Code to do if the condition is false ...
// ... More code ...
```

For even more flexibility, else if statements can be inserted between the if and else statements, and the else statement can even be omitted. An else if statement provides and alternative condition if the initial condition is false. If the first condition is false, control passes to the first else if. If the first condition is true the first block is executed, and the if/else if/else construct ends. If the first else if is called and its true, its executed. If false, control passes to the second else if. A final else can be placed at the very end, after all the else ifs which executes only if none of the previous conditions are true. A full if/else if/else contruct takes the following form:

```
// ... Some code ...
if condition_1:
    // ... Code to do if the condition is true ...
else if condition_2:
    // ... First else if ...
else if condition_3:
    // ... Second else if ...

...

else if condition_N:
    // ... N-1th else if ...
else:
    // ... Code to do if none of the else if's executed ...
// ... More code ...
```

In the above template, the final else can be omitted. If this is done, then it is possible for none of the if's or else if's to execute, and the whole structure might have done nothing. Whether this is good depends on what you are trying to accomplish.

4.2 Switch

A switch block is a slightly more complicated control structure that executes certain blocks of code depending on whether or not certain cases match an initial variable. A switch is defined on a single variable, and the following block (indented) contains several subordinate case blocks and an optional default block. A switch takes the following basic form:

```
switch var:
    case a:
        // ... code to execute if var == a ...
        break
    case b:
        // ... code to execute if var == b ...
        break
    case c:
        // ... code to execute if var == c ...
        break

...

    case n:
        // ... code to execute if var == n ...
        break
    default:
        // ... code to execute no matter the value of var ...
```

For each `case`, the value of `var` is tested against the case value. If they are equal, then the case will execute. The last case is a `default` case, which will always execute if its reached. Each case ends in a `break`, which serves to break out of the whole switch. The reasonfor this is to prevent what's called fall through. Once a one of the cases matches the variable, the code will execute for that case and all subsequent cases until the switch ends, or a `break` is encountered. If none of the statements include breaks, the default case will always occur. Consider the following code:

```
num n = 0
switch n:
    case 0:
        println("n == 0")
    case 1:
        println("n == 1")
    case 2:
        println("n == 2")
    default:
        println("n != 0,1, or 2")
```

The output of the above code would be:

```
n == 0
n == 1
n == 2
n != 0,1, or 2
```

Fall through is useful if you want to create or conditions (if the variable equals a or it equals b). For example, putting to cases next to each other will cause the following block to execute if either of the cases are met. Placing a break, will prevent further code from executing.

A default case is not necessary at the end of a switch block. Nor is it necessary to include a break inside of a default case.

4.2.1 A Note About `break`

In the above section, everytime the keyword `break` was used, it was used by itself with no additional arguments. The function of a `break` statement is to exit from the currently enclosing `case` statement or loop (see section 5). However, in Phoenix a `break` can be followed by a positive integer (if it is not positive or an integer, an error will be thrown). This number specifies how many loops and case statements should be broken out of. Thus if there are 4 nested loops, and inside the innermost one, there is a `break 3` statement, all but one of the loops will be broken out of. However, if one wants to break out of all open loops and case statements, one can use the keyword `all`. By using the statement `break all` all open loops and case statements will be exited, taking the flow up to the top non loop or case. **Note:** if the number of loops is too high or a break traces back to a function (see section 6) and attempts to go past the scope of the function, an error will be thrown.

5 Loops

Loops are more complicated control structures. A loop defines a block of code that continuously executes until conditions become false.

5.1 The While Loop

The while loop is the simplest loop. The while loop simply consists of a conditional and a block of code. The block of code is repeatedly executed until the condition of the while loop becomes false. The following is the format of a while loop:

```
// ... Some code ...

while condition:
    // ... Code that is executed during the loop ...
// ... More code ...
```

Thus if one wanted to loop until a variable equaled 5, printing out intermediate values, the program would look like this:

```
num n = 0
while n < 5:
    println("n = " + n)
    n++
println("Done")
```

The loop starts with `n` equal to zero. As the loop continues, it prints out the current version of `n` and then increments it by one. As soon as `n` equals 5, the statement `n<5` is no longer true, and the loop ends; the string `"Done"` is printed. Therefore the output of the above code would be:

```
n = 0
n = 1
n = 2
n = 3
n = 4
Done
```

Because of the nature of the while loop, a while loop might never execute. If the condition of the while loop is false, prior to the loop starting, the loop will never execute. Thus the following code will output nothing:

```
num n = 10
while n < 5:
    println("n = " + n)
    n++
```

5.2 The Do While Loop

The do while loop is almost identical to the while loop except for the fact that the while is placed at the end of the loop. The syntax of the do while loop takes this basic format:

```
// ... Some code ...
do:
    // ... Code that executes during the loop ...
while condition
// ... More code ...
```

The loop operates in the same manner: the code inside the loop is executed until the condition becomes false. When the condition becomes false, the loop ends and the flow of the program continues at the end of the loop. The only difference is that in a do while loop, the condition is evaluated at the end of the loop. This means that a do while loop will **always** executed on the first go around. Consider the following code:

```
num n = 10
do:
    println("n = " + n)
    n++
while n < 5
println("Done")
```

The code above will be able to execute the first time even though the initial condition is false. Thus the output will be:

```
n = 10
Done
```

5.3 The For Loop

The for loop is a different type of loop that simplifies the process of looping while incrementing a variable. The for loop clause, consists of three separate pieces: an assignment, an end condition, and an increment statement. The assignment sets a variable equal to a starting value. The end condition defines when the loop is going to end, and the increment statement increments or decrements the value on every iteration of the loop. The three statements (assignment, end condition, and increment statement), follow the keyword `for`, and are delimited by semicolons. The for loop takes the following syntax:

```
// ... Some code ...
for assignment; end_condition; increment_statement:
    // ... Code that executes during the loop ...
// ... More code ...
```

The for loop makes it easy to loop over an interval of a number variable. For example, suppose you wanted to loop over all the integer values of a number `x` from 1 to 5. This can be accomplished easily with a for loop in the following way:

```
num x
for x = 1; x <= 5; x++:
    println("x = " + x)
println("At the end, x = " + x)
```

On the first iteration of the loop, the value of `x` is set to 4 and the condition, `x <= 20`, is checked. Then the value of `x` is printed. At the end of the loop, the increment statement, `x++`, is executed and **then** the condition is checked to see if it is still true. Because the increment occurs **before** the end condition is checked, the following is the output of the above code:

```
x = 1
x = 2
x = 3
x = 4
x = 5
At the end, x = 6
```

Note that you can't define any variable inside the assignment clause. This is the case in other languages, but not the case in Phoenix. Thus the following code would case an error:

```
for num n = 0; n < 10; n++:
```

However, the above code should be replaced as:

```
num n
for n = 0; n < 10; n++:
```

5.4 Variable Scope in Loops

Variable scope is an interesting problem in loops. One would think that since a loop is a single scope, any variable defined within a loop would get redefined on any future iterations of the loop, thus causing an error. Since the variable would be defined twice, without being deleted, an error would occur. However, this is not the case. What technically occurs is at the beginning of each loop, a new scope is defined, and at the end of each loop, the current scope is left, and all created variables are destroyed. Thus, it is possible to create new variables within a loop. These variables will be destroyed at the end of the loop and thus when they are defined on the second iteration of the loop, no error occurs. Variables that are defined outside of the loop are accessible in the loop and after it finishes.

5.5 Using `continue` in Loops

The keyword `continue` in loops moves the control to the end of the loop. This means that the current iteration is ended and the next iteration occurs, if it can occur. If the end condition would be false at the end of the loop, then the loop exits like it normally would, otherwise the next loop iteration starts like it normally would. If the keyword `continue` is found inside a conditional (if / else if / else or switch) inside the loop, it will end the smallest enclosing loop.

The keyword `break` in loops simply ends the loop, no matter the ending condition. If `break` occurs inside of a conditional, then the smallest enclosing loop will be the one ended.

6 Functions

Functions are sections of code that can be repeated whenever wanted, with variable input and output. Consider the mathematical function $y = f(x)$. The function has the name f, takes an input variable x, and based on the value of x, returns another value y. Phoenix functions work in the same manner, but tend to be more complex. Phoenix functions can take input in the form of `num` and `str` variables, and can take more than one variable as input. They can also return a single `num` or `str` variable, or return nothing (`void`).

Most importantly, Phoenix functions can have left-hand and right-hand arguments. This means that all of the folowing functions are possible $n = f(x)$, $n = (x)f$, or $n = (x)f(y)$. The fact that arguments can be on both sides of a function means that functions can make more logical sense. For example, consider the factorial function $x!$ (pronounced x factorial). In Phoenix instead of typing `factorial(x)`, one can type `(x)factorial`, which is how it should be said. As another example, consider the binomial coefficient function, sometimes written ${}_nC_k$ (pronounced n choose k). In Phoenix, instead of typing `choose(n,k)`, one can type `(n)choose(k)` or possibly `(n)C(k)`.

6.1 Declaration

A function declaration creates a new function. Following the function definition is a block of code which is executed when the function is called. A function declaration consists of six distinct parts:

1. The `function` keyword. This must start a function declaration; if it is not present Phoenix won't recognize this as a function declaration.

2. The return type of the function. This is what variable type will be returned by the function. This can be `num` or `str`, if returning a variable. If the function returns nothing, this should be `void`.

3. The left-hand arguments. These should be enclosed by parentheses, and consist of a list of variable types each followed by names, separated by commas. For example this could be `(num a, str b)`, which would mean that the left-hand arguments would be a number variable `a` and a string variable `b`. All names should be standard variable names and not repeat each other. If no left-hand arguments are desired, this should be omitted.

4. The name of the function. This should follow all the same rules as variable names and **cannot** be the same name as any variable currently instantiated. The conventions for variable names are the same for function names except for the rules about constant variables. These convention rules can be found in section 3.6.

5. The right-hand arguments. These follow the same rules as the left-hand arguments, and cannot repeat any of the lef-hand arguments' names. If no right-hand arguments are desired, this should be omitted.

6. A colon.

Since both the left-hand and right-hand arguments are not requisite, functions can have no arguments. This means that they will look just like variables when using them. This is why functions cannot have the same names as variables: because there is no way to differentiate them. Thus all of the following are valid function definitions:

```
function num power (num x, num y):
function num (num x) squared:
function num (num a) plus (num b):
function void printOut(str s)
function num random:
```

To call (execute) the above functions one would do something like this:

```
square (3)
(5) squared
add (8, 1)
(2) plus (-9)
random
```

The block of a function (the section that comes after the function definition), then defines what the function will do. The block of a function must be indented an extra time. Functions that return values (the return type is not `void`), must have a `return` statement. The `return` statement is followed by a variable that is going to be returned by the function. For example, this is a possible function definiton:

```
function num (num x) squared:
    return x ^ 2
```

6.2 Function Scope

A local function is defined through methods described above. A local function is accessible in its current scope and any scope below it (any place the indentation is greater). However, once the current scope exits (the indentation is reduced), the function will be deleted, and never accessible. Therefore, functions that are defined on the top level of a program (no indentation) will always be accesible and will never go out of scope. Function scope is identical in behavior to variable scope (see sections 3.4 and 5.4).

Global functions are defined by adding the keyword `global` prior to the function definition. (For example: `global function num (num x) squared:`). A global function will always be accessible no matter where it is created. Even if a global function is created at a very low scope, it will be accessible at every scope. Global functions will never go out of scope and therefore will never be destroyed. However a global function (or any function for that matter) can be removed using the `delete` keyword. This is just like how its done for variables:

```
delete name
```

6.3 Variable and Function Scope in Functions

Almost all variables are not in scope inside functions of any scope. The only variables that are in scope inside of a function are global variables and the variables passed into the function via the arguments. All other variables cannot be accessed inside a function. This does not change based on whether the function is global or local, on any level.

Many functions are in scope within functions. If a function is local, then every function that is on the same, or higher scope is accessible within that functions scope. If a function is global, only global functions are accessible within that function. Global functions are thus accessible in every function. A function can also call itself, this is called recursion. However, if a function calls itself, it must have an ending condition, otherwise it will keep calling itself, continuing on indefinitely.

To demonstrate function scope within functions, consider the following code:

```
global function num func1:
    return 0

function num func2:
    return func1

// Begin a new scope (if, while, switch, etc.):
    function num func3:
        return func2

    function num func4:
        return func3

   global function num func5:
       return func4
```

The only function above that will cause an error is `func5`. This is because a global function cannot make a call to a local function, because it is out of scope. The rest of the functinos make valid calls to functions that are within the correct scope.

6.4 Function References

Besides variables of type `num` and `str`, there is one more type of argument that can be passed to a function. This argument is a function reference. In other words, it is possible to pass a function, as a argument to another function. By passing a function, the internal function can call the passed function and use it inside the function. An argument to a function that is a function, looks like a normal function declaration without any of the keywords `local`, `global`, or `native`. Here is an example of a function declaration that involves a function argument:

```
function num (function num x (num a)) evaluatedOn (num n):
    return x(n)
```

The function above takes a left-hand argument that is a function, returning a `num` and taking a single `num` as a right-hand argument. The right-hand argument of the above function is a single `num`. The body of the function simply calls the function passed through the left-hand parameter on the variable passed through the right hand parameter. Note that the function parameter on the left must provide names for its arguments, even though it doesn't really matter. The names can intersect with each other or regular function arguments, but *must* be present.

To call the above function, one must use the special function reference operator, `@`. The function reference operator must be followed immediately by a `str` that contains the name of the function that is being referenced. This string with the preceding `@` is then passed to a function as the passed function. For example, here is an example of how the above function would be called:

```
function num square (num x):
return x ^ 2

(@"square") evaluatedOn (4)
```

In the above code, it is first necessary to define a function that will be passed. Note that the function `square` has the same form as the one that the function `evaluatedOn` takes as its left-hand argument. Both function return a `num` and a take a `num` as their right-hand argument. However, their names do not match, nor do the names of their arguments. This does not matter. As long as they match in form, the names are irrelevant.

After the `square` function is defined, we then make the call to `evaluatedOn`. For the left-hand we pass a string whose name is *exactly* equal to the name of the `square` function, with a function reference operator appended to the beginning as the left-hand argument, and the number `4` as the right-hand argument. The output of the above program would be `16`.

6.5 Native Functions

Native functions are functions that call Java programs instead of executing Phoenix code. A function is declared as native using the keyword `native` in the function declaration. The keyword `native` appears before the keyword `function`. If the function is also declared as `global`, the modifiers `native` and `global` can appear in either order. The only content of the function should be a relative or absolute path to a Java class. The return type of a native function must by `void`. All of the arguments passed to native functions are passed as command line arguments to the Java programs, in left to right order. Thus, all of the following are valid native function definitions:

```
native function void func1 (num x, num y, str z):
    Foo.class

global native function void (str x) func2 (num y, num z):
    Java/src/Bar.class

native global function void (num x, str y, num z) func3:
    ../Classes/Whatevskis.class
```

The above functions and equivalent calls to the `java` command are given below:

```
func1(3,4,"Hello There")     ⇒  java Foo 3 4 "Hello There"
("Hello There") func2 (3,4)  ⇒  java Java/src/Bar "Hello There" 3 4
(3, "Hello", 4) func3        ⇒  java ../Classes/Whatevskis 3 "Hello" 4
```

A Reserved Keywords

`all`	Indicates to a `break` that all loops and breaks are to be exited (Section 4.2.1)
`break`	Exits a loop or a `case` statement (Section 4.2.1)
`case`	Defines a case of a `switch` block (Section 4.2)
`const`	Modifies a variable to make it constant (Section 3.5)
`continue`	Ends the current iteration of a loop and starts the next (Section 5.5)
`default`	Defines a default case of a `switch` block (Section 4.2)
`delete`	Deletes a variable or a function (Sections 3.0 and 6.2)
`do`	Starts a do while loop ending at a given condition (Section 5.2)
`else`	Defines an alternative clause in an `if` statement (Section 4.1)
`for`	Defines a for loop which loops over the range of a variable (Section 5.3)
`function`	Defines a function (Section 6)
`global`	Modifies a variable or function to make it global in scope (Sections 3.4, and 6.2)
`if`	Defines an if clause (Section 4.1)
`local`	Modifies a variable or function to make it local in scope ()
`native`	Modifies a function to make it call a Java class (Section 6.4)
`num`	Specifies a variable as a number variable (Section 3.1)
`return`	Returns a value as the result of a function (Section 6.1)
`str`	Specifies a variable as a string variable (Section 3.2)
`switch`	Creates a block which executes case statements based on a variable (Section 4.2)
`void`	Specifies a function's return type as being nothing (Section 6.1)
`while`	Defines a loop which repeats until a condition becomes false (Sections 5.1 and 5.2)

B Operators

See sections 3.1.1 and 3.2.1 for detailed information on number and string operators respectively.

Operator	**Precendence***	`num` **Function**	`str` **Function**
`+`	6	Addition	Concatenation
`-`	6	Subtraction	
`*`	7	Multiplication	Repetition
`/`	7	Division	
`%`	7	Modulus Arithmetic	
`^`	8	Exponentiation	
`#`	6	Rounding	
`==`	4	Equal to	Equal to
`!=`	4	Not equal to	Not equal to
`>`	5	Greater than	Greater than
`>=`	5	Greater than or equal to	Greater than or equal to
`<`	5	Less than	Less than
`<=`	5	Less than or equal to	Less than or equal to
`&`	3	Logical AND	
`|`	2	Logical inclusive OR	
`(+)`	1	Logical exclusive OR	
`!`	9	Logical complement	
`[]`	11		Subscript / Slice
`=`	0	Assignment	Assignment
`+=`, `-=`, etc.**	0	Combined Assignment	Combined Assignment
`x++`, `x--`	10	Postfix increment / decrement	
`++x`, `--x`	9	Prefix increment / decrement	
`@x`	9		Function reference

* The precedence of an operator tells when it is evaluated in the order of operations. Operators with a higher precedence number get evaluated sooner, those with lower precedences get evaluated later. All operators of the same precedence are occur from left to right across an expression. Parenthesis have higher precedence than any of the above operators. Functions can be thought of as operators and have a precedence of 11.

** The combined assignment operators are enumerated in full in section 3.3.

C Glossary

block — A section of code with the same or greater indentation, and all the code in that section.

global — The topmost scope of a Phoenix program. Any variable or function declared with the `global` keyword is put into this scope, which is above the base level scope (no indentation).

keyword — A certain word that has a special, predefined purpose in the programming language. See Appendix A for this list of all keywords.

literal — A value that can be interpreted, but is not a defined variable. Examples include `12.3` for a number and `"abc"` for a string.

local — Any scope that is not global in a Phoenix program. This refers to any scope that is defined by the level of indentation.

operator — A symbol typed in Phoenix that performs a specfic function on one or two operands. See Appendix B for a list of all operators.

scope — A section of code in which variables exist. When that section of code leaves, all variables in that section are destroyed.

string — A sequence of characters that forms a distinct statement. Various string literals include `""`, `"Hello"`, `"What's going on?"`, and `",./?"`.

variable — A identifier that holds a value that can be a number or a string. See section 3 for more information on variables.

D More Information

For more information about Phoenix see my website <www.scheinerman.net/jonah>. On the website you will be able to download Phoenix, the accompanying integrated development environment. If you have further questions about Phoenix, please feel free to contact me using the Phoenix language email address: phoenix.proglang@gmail.com

If you are a developer and are interested in using Phoenix in an application you are using, please peruse the accompanying technical manual. It provides details of how to use Phoenix in an application you are producing and gives details on the inner workings so that you can customize Phoenix for your uses.

If you are an educator or a student, the technical manual should also be a useful document for you as well. It provides very specific details on how all of the different Phoenix systems work, and is interesting to look at Phoenix as a case study. To go along with that please download the source code and the look at the javadocs. These will help you to understand how Phoenix works under the hood.