PHOENIX programming language

Written by: Kristine Hambardzumyan
Last revision: Nov 15, 2021

DOCUMENTATION

Hey Friend!
Phoenix is a concise, clear and interpreted language.
The interpreter of Phoenix is written in C++.

PREREQUISITES

- ✔ Any previous knowledge of other programming languages can be useful to better understand all the concepts
- ✔ A text editor like vim, or notepad to edit your code
- ✔ A command terminal

GET STARTED WITH PHOENIX

If you're using Linux:

- ✔ Open an editor and write your source code. Name the file, for example Phoenix.txt.
- ✔ Download PhoenixMain.cpp, Phoenix.cpp, and Phoenix.h files
- ✔ Open a command prompt
- ✔ Compile PhoenixMain.cpp and Phoenix.cpp files
- ✔ When executing the source code, specify the file where you've written it, for example: ./a.out Phoenix.txt

FORMATTING

Parentheses
In Phoenix control structures don't have any parentheses in their syntax.

Line length
No line length limit

Spacing
A space after each token is necessary. There can also be multiple spaces between the tokens.
Blank lines are allowed to be used to help visually separate out logical blocks of code.

End of line
No need to include semicolon(;) or any other character at the end of a statement.
\n is marks the end of the statement.

VARIABLES

DATA TYPES

In *Phoenix* each variable has a specific type and an identifier.
Phoenix's basic types are:

<u>*integer*</u>
<u>*real*</u>
<u>*boolean*</u>
<u>*text*</u>

NAME	***DESCRIPTION***	***RANGE***
integer	integer	$2^{(-31)}$- 2^{32}-1
real	floating point number	+/- 1.7e +/- 308
boolean	Boolean, takes two value: true or false	True or false
text	Non-numerical values	String of ASCII characters

Note: range of each type depends on the platform the program is compiled on. The values shown above are those found on most 32-bit systems.

VARIABLE DECLARATION
In order to declare a variable we must specify its data type followed by a valid variable identifier.
For example:

```
integer number
real r
```

UNINITIALIZED VARIABLES
Variables declared without an explicit initial value are NOT given zero or other value.
The default value of that variable is garbage (whatever value happens to already be in that memory location)

INITIALIZATION OF VARIABLES
The assignment sign(<<) is used to initialize variables.
If we want to declare an integer variable called number initialized with a value of 10 at the moment in which it is declared, we could write:

```
integer << 10
```

In order to initialize a text type variable with a value we enclose it between double quotation marks(“”) like in the following example:

```
text t << “I’m a text”
```

IDENTIFIERS
- There are a few simple variable naming rules you should follow in Phoenix:
- The only characters you can use in identifiers are alphabetic characters (a-z, A-Z), numeric digits(0-9), the underscore(_) and the asterisk(*).
- The first character in a name cannot be a numeric digit(0-9).
- Uppercase characters are considered distinct from lowercase characters.
- You cannot use Phoenix keyword for an identifier.
- No limits on the length of an identifier.

Here are some valid and invalid Phoenix identifiers:

```
integer number      //valid
real Number         //valid and distinct from number
boolean _var        //valid
text *var           //valid
boolean case        //invalid – case is a keyword in Phoenix
```

```
real 1num            //invalid – cannot start with a numeric digit
integer i-2          //invalid – no hyphen(-) is allowed
```

LITERALS

Literals are used to express particular values within the source code of a program.
We have already used these to assign concrete values to variables:

```
real  r << 5.6
```

Literals constants can be can be integer numerals, floating-point numerals, characters, strings and boolean values.
Examples:
6232, 4.0, 2.1e-12, “a”, “Hello!”, true, ….

OPERATIONS

ASSIGNMENT (<<)

Assigns value to a variable:

```
number << 55
text t << “Hi!”
number1 << number2
```

Note that the left value has to be variable, and the right value can be either a literal or a variable of the same type.

ARITHMETIC OPERATIONS

+ addition
- subtraction
* multiplication
/ division

Operations correspond with their respective mathematical operators.
Arithmetic operations are only possible for integers and reals types.

The following syntax is used on order to do any arithmetic operation:

```
integer sum << 0
sum << sum + 5
print sum          // 5
```

Note that if you do an operation in integer type variable, both operands must be integers.

```
integer sum << 0
sum << sum + 5.6   //invalid – 5.6 is not an integer type
```

```
real result << 10
result << result - 5   // valid – 5 is considered as both real and integers
```

RELATIONAL AND EQUALITY OPERATORS
(=, x=, >, <)

The result of a relational operation a Boolean value that can only be true or false.
A list of relational and equality operators that can be used in Phoenix:

=	equal
x=	not equal
>	greater
<	less

These operators can be used with types integer and real.
Here are some examples:

```
15 = 15      //evaluates to true
15 x= 15     //evaluates to false
8.2 > 8      //evaluates to true
15 < 3.4     //evaluates to false
a > 10       //will evaluate to true if the value of a is grater than 10
```

BASIC OUTPUT
print is used to print out a text message or the value of a variable on the screen.
In order to print a text message we enclose it between double quotation marks(“”) .
Here are some examples:

```
print “Hello Friend!”            // prints out Hello friend!
real var << 4.2
print “The value of var is:”     // The value of var is 4.2 (note that it also prints a space after is)
print                            //prints an empty line
```

CONTROL STRUCTURES
CONDITIONAL STRUCTURE *case*
The *case* keyword is used to execute a block only if a condition is satisfied. The syntax is:
case *condition* [
statements
]

where *condition* is expression that is being evaluated, If *condition* is true, *statement* is executed. If it is false, *statement* is not executed and the program continues right after this conditional structure.
Examples:

```
boolean b << true
case b [
print “Hi there!”
]                      // b is true, so Hi there! will be displayed on the screen.
```

```
real r << 9.7
case r = 9.7 [
r << r + 1
print “r =” r          // the condition is true, so r = 10.7 will be displayed on the screen.
]
```

ITERATION STRUCTURE *do-until*

do-until has the purpose to repeat a statement while a condition is satisfied.
Syntax:
do-until *condition* [
statements
]

It basically repeats *statements* while the *condition* is true
Examples:

```
integer number1 << 7
real number2 << 3.14
do-until number1 > number2 [
print “Hola”
number1 << number1 – 1
]
```

Hola will be printed for 4 times

KEYWORDS
integer, real, boolean, text, case, do-until