The Pheonix Programming Language

Neil Kingdom and Mohit Nargotra

Algonquin College

1. Introduction to Pheonix

1.1. What is Pheonix?

Pheonix is a programming language based loosely on a combination of C, Python, and Bash. Pheonix hopes to provide the end user with a simple and intuitive syntax for easy development. Due to the compiler likely skipping the optimization process, Pheonix is not as fast as it could be, but we believe that the time lost from missing optimizations is saved from not having to try and remember strange keywords or awkward function definitions. Not only does its simplicity help the end user, but also helps avoid messy logic that might lead to compile errors.

Pheonix is a strongly typed language, meaning that variables are (as best as we're able to make them) type-safe and aid in prohibiting coders and the compiler from making mistakes. Each translation unit is comprised of an include file (.inc) and the Pheonix source file (.phx).

2. Defining a New Language

2.1. Basic Syntax Staples

Every line in Pheonix ends with a semi-colon as a delimiter (with the exception of statements that execute code blocks eg. loops, functions, etc.), comments included. This may seem annoying to the programmer in question, however, we feel that this adds consistency and clarity to the code. Comments begin with the hash '#' symbol, so a comment might appear as follows: **# This is a comment;** The compiler only seeks the semi-colon ';' as the dilimiter to end comments, so multiline comments work using this method as well. If the user wishes to add a semi-colon in the comment, it must be escaped with the bakclash '\' character.

Functions are explicitely declared as such using the function keyword. A return type is expected, and parameters are parsed via whitespace before the opening brace '{'. A typical function definition might look something like the following: **function int main parameter1 parameter2 ... {** This is inspired by Bash's syntax and is intuitive for beginner programmers. Outputting to the console is as simple as using the print function, and input is recieved with the input function.

With these basic elements in place, we can create a "Hello World" program. It might look something like the following:

```
import phxio.inc

# Creating a "Hello World" Program in Pheonix;

function int main {

    print "Hello Word";

    return 0;
```

}

2.2. Datatypes and Keywords

Following the philosphy of simplicity carried over from scripting languages such as Python and Bash, Pheonix has fairly traditional keywords to maintain a sense of familiarity. Some of the basic datatypes that we'd like to include the following:

- string - Should be dynamically sized according to the length of the string litteral ie. n chars * sizeof(char)
- int - 4 bytes
- float - 4 bytes
- char - 1 byte
- bool - 1 byte (Seems like a waste of space, but allocating a single bit feels like it would lead to problems)
- void - 0 bytes. Does not hold any data

As for keywords, here are some of the important ones that we think are necesarry:

- if - For if statements
- while - For while loops
- case - For switch statements
- import - To include header/include files
- function - To declare a new function
- return - For the return value of a function
- main - The entry function of the program
- true/false - Boolean values representing 1 and 0 respectively
- $n - Dollar sign followed by a number is reserved for local parameters

2.3. Assignment

When assigning variables, we've decided to stick to the familiar method of using an assignment operator such as '=' which would store the value on the right side of the operator into the memory location of the variable on the left side of the operator. Eg:

int number = 1;

As of the time of writing this documentation, we do not have plans for casting data-types, although we are open to the idea. We feel that this is something that should be agreed upon later in the development of the Pheonix compiler.

2.4. If, While, and Case

In terms of making if, while, and case statements; we'd like to follow the convention of not using round brackets as an entry and end point ('(', ')'). This means that anything between the of the if,while,and case statements and the code block delimiter ('{') will be considered part of the boolean expression.

The way these boolean expressions will be evalutated is not entirely clear yet, although we plan on taking a numerical approach. What we mean by that is that the statement should either evaluate to a negative number, 0, or a positive number. A negative number and zero indicate false, and positive numbers indicate true. Certain operators will take precedence over others, though we haven't fully worked this out yet.

For example, consider the following if statement:

```
if 3 > 5 + 7 { ... }
```

In this example, there exists a boolean operator (’>’) and an arithmetic operator (’+’). If the order of precedence was boolean operators, then arithmetic operators, 3 > 5 would simplify to 0 since the statement is false, and then 0 + 7 = 7 which would make the if statement evaluate to true. As an aside, if, while, and case statements should all accept the true/false keywords and expand them to 1 and 0 respectively. For loops are something we’d like to implement but haven’t came up with the syntax for. Other than that, for-each loops and do-while loops are not something we’re considering as of now.

2.5. Functions, Parameters, and Return Values

As we briefly went over, functions should start with the function keyword. The order for creating a function is the following: **<keyword - function> <return type> <label> [parameter types] { ... }** Instead of creating local parameter names, Pheonix assigns each parameter the local variable $n, where n corresponds to the parameter number. Here is an example of a function declaration and function call, passing parameters to the function:

```
# function declaration;
function void foo int string char {
    print $1;
    print $2;
    print $3;
    return;
}
# function call;
foo 7 "test" ’z’
```

In this example, a function called foo is created, with a return type of void, and it takes 3 parameters of type int, string, and char. These 3 parameters are then assigned to the local variables $1, $2, and $3 respectively. Under the hood, (though we haven’t gone into the architectural aspects of the language yet), the compiler creates the variables $1, $2, and $3 with the matching data types to maintain type safety. This is what it would look like if this was visible to the user:

```
# function declaration;
function void foo int string char {
    # Compiler does this secretly;
    int $1 = 7;
```

```
    string $2 = "test";

    char $3 = 'z';

    print $1;

    print $2;

    print $3;

    return;

}
```

If no return value is given like in the example above, void is assumed, ie. return nothing. Otherwise, if a value is given after return and matches the return type, the function will return sed value. Multi-return values are not allowed in Pheonix ie. only one value can be returned in any particular function.