The Wick Manual

University: Arkansas Tech University
Course: Capstone II
Professor: Dr. Bhaskar Ghosh
Project Contributors: Braden Pierce, Evan Keathley, Lisset Luna, Evelin Cerros-Patricio

Table of Contents:

1 - Wick Introduction

The Wick language was created with the intent of serving as an educational resource not only for our team to learn about programming language and compiler design, but to serve as a resource for our peers to learn about these often neglected topics in computer science. To this end, the project involved the creation of a unique programming language, Wick; an interpreter to recognize and execute programs written in this language; and an online IDE for others to experiment with the language.

1.1 - Reason for this Project

Compiler and programming language design has become increasingly deemphasized in computer science programs. At ATU specifically, Compiler Design is no longer a required class for CS majors and has even been dropped from the course catalog entirely. This trend is sparked by the perception that compiler design has lost its utility: in days gone, one may have had to create a compiler to get their system to a functioning state (a diminishingly rare situation now). Our team, however, thinks that compiler and programming language design serves as a cornerstone for much of computer science and is valuable information by itself. With Wick we hope to fill this knowledge gap and produce a valuable resource for ourselves and our peers to learn more about how programming languages work.

1.2 - Project Overview

1.2.1 - The Wick Language

Wick is presently a simple, dynamically-typed, and multi-paradigm language, supporting. First class functions (here called subroutines) are supported as well as object-oriented programming via prototypes. While our team does not currently recommend the use of the language outside of hobbyist circles, Wick could be an excellent resource in classroom settings for those that are less adept with computer science and programming.

1.2.2 - The Wick IDE

The Wick IDE is designed for simplicity. The online IDE utilizes Node.js to establish a server. A file system is integrated into the IDE to generate file outputs, which are then shown in the terminal interface. Within the text area, you can write code in the Wick language, and the output is displayed in the designated text area. Currently, the website features only a "run" button. It's incredibly straightforward to use—all you have to do is write your code in the Wick language, then click the "run" button to see your results.

2 - The Language

2.1 - I/O

In the Wick language, the print function is used to show information, messages, or results on the console or terminal. For example, print("Hello World"); displays "Hello World" on the console. The input function lets a program get information from the user while it's running. It waits for the user to type something, and the typed data is usually saved in a variable for later use. String operators are tools used to change and manage text data (strings) in a program.This is an example in the Wick language:

```
/:
Hello, this is a multiline comment! It doesn't execute
any code and is for documentation purposes. This file
will print "Hello world!" and greet someone based on
a given name.
:/

print("Hello world!");
variable name = input("What's your name?");
print("Hello " + name + ".");
print("Nice to meet you!");
```

2.2 - State

In the Wick language interpreter, the state encompasses both variables and constants, offering a foundation for dynamic data manipulation alongside fixed values.

For example, in the given code snippet, constants like 'a', 'b', and 'c' are initialized with specific initial values. 'a' is set to 1, and 'b' is also the value of 1. That leaves constant 'c' to be computed based on these predefined constants, employing mathematical expressions such as addition and multiplication. In this case it would need to print out the answer 2 like shown below. Also shown is an example of variables and reassignment.

```
constant a = 1;
// a = 2;
constant b = 1;
constant c = a * a + b * b; // Should be two.
print(c);
```

```
variable x = c / 4;
x = x * 4;
x = x mod 4; // Should be two.
print(x);
```

2.2.1 - Basic Operators

Basic operators in the Wick language interpreter cover numeric operations, facilitating mathematical calculations within the program. These operators enable fundamental arithmetic actions such as addition, subtraction, multiplication, and division on numeric values.

For example, in the snippet below we can see basic arithmetic multiplication and addition used to calculate the answer to constant c.

```
constant c = a * a + b * b; // Should be two.
```

2.2.2 - Scope

In Wick programs all code is initially written in a global scope. Whenever a pair of curly braces appears, a nested scope is created. These nested scopes can have names occurring in the global scope (called variable shadowing) and will go out of scope when out of the two braces.

The code below demonstrates the concept of variable shadowing in nested scopes. Within the block, x is declared and initialized to 4, creating a local variable x. When print(x) is called within the block, it prints the value of the local variable x, which is 4. Outside the block, when print(x) is called again, it references a different x (as the local version has gone out of scope), which is a global variable initialized in the code with a value of 2.

```
variable x = c / 4;
x = x * 4;
x = x mod 4; // Should be two.
print(x);



// Nested scopes may contain shadowed variables.
{
    variable x = 4; // Local x.
    print(x);
}
print(x); // Local x goes out of scope and global x is printed (two).
```

2.3 - Native Constants and Subroutines

In the Wick language, native constants and subroutines are built-in features to help developers efficiently build applications. They offer fundamental functionalities that programmers can use directly in their code without the need for external libraries. Native subroutines, also known as functions or procedures, are pre-written code blocks that perform specific tasks. There are some native functions in the Wick language.

All natively-supported subroutines and constants:

- subroutine doNothing(); // Does nothing. Used mostly as a helper internally, could also be used to represent unimplemented code.
- subroutine max(numberA, numberB); // Returns the highest number between a and b.
- subroutine min(numberA, numberB); // Returns the lowest number between a and b.
- subroutine print(msg); // Prints a message.
- subroutine input(msg = “”); // Gets input from the terminal and prints out a message if one is provided.
- subroutine time() // Prints out the current time.
- subroutine abs(number); // Gets the absolute value of a number.
- subroutine round(number); // Rounds a number up.
- subroutine floor(number); // Returns the floor value of number.
- subroutine ceil(number); // Returns the ceiling value of number.
- subroutine truncate(number); // Rounds a number toward zero.
- subroutine pow(number); // Raises a number to an exponent.
- subroutine exp(number); // Raises a number to the power of e
- subroutine sqrt(number); // takes the square root of a number.

- subroutine cbrt(number); // Takes the cube root of a number.
- subroutine hypotenus(numberA, numberB); // Gets the hypotenus of two sides of a right triangle.
- subroutine log(number); // Gets the base 10 log of a number.
- subroutine lg(number); // Gets the base 2 log of a number.
- subroutine ln(number); // Gets the natural log of a number
- subroutine sin(number); // Gets the sine.
- subroutine cos(number); // Gets the cosine.
- subroutine tan(number); // Gets the tangent.
- subroutine sinh(number); // Gets the hyperbolic sine.
- subroutine cosh(number); // Gets the hyperbolic cosine.
- subroutine tanh(number); // Gets the hyperbolic tangent
- subroutine arcsin(number); // Gets the inverse sine.
- subroutine arccos(number); // Gets the inverse cosine.
- subroutine arctan(number); // Gets the inverse tangent.
- subroutine arctan(numberY, numberX); // Gets the inverse tangent, accounting for quadrant.
- subroutine arcsinh(number); // Gets the inverse hyperbolic sine.
- subroutine arccosh(number); // Gets the inverse hyperbolic cosine.
- subroutine arctanh(number); // Gets the inverse hyperbolic tangent.
- subroutine isnan(number); // Determines if the value is not-a-number.
- subroutine MAX_VALUE (number); // Maximum supported value for a number.
- subroutine MIN_VALUE(number); // Minimum supported value for a number.
- subroutine NaN // Value representing not-a-number.
- subroutine PI // Native value for PI.
- subroutine E_V // Provide native value for e.

An example of native subroutines in action:

```
variable x = PI;
print(cos(x));
print(sin(x));
print(tan(x));
```

2.3.1 - NaN

In the Wick, NaN, not a number, is used in the language. NaN is a specific value within a numeric data type, often associated with floating-point numbers, that represents an undefined or indeterminate result. NaN is a common feature across various programming languages. This is a Wick example of NaN:

```
variable y = sqrt(-1);
print(isnan(y));
```

2.4 - Conditionals

In the Wick, conditionals play an important part of the execution for the interpreter. Wick supports various types of conditionals, including the traditional if and if-else constructs, as well as ternary expressions and boolean operators. Wick supports ternary expressions, which are similar to if-then statements except they evaluate to some value. In Wick, false, zero, and empty strings evaluate to false while everything else evaluates to true. The code down below is an example of conditionals being used.

```
if !oneEqualsOne {
    print("This will never run...");
} else if(eEqualsPi) {
    print("This will also never run...");
} else {
    print("This will always run.");
}
```

```
variable x = 1 if true else -1;
print(x);
```

2.5 - Loops

In the Wick language, loops are used to execute a block of code repeatedly under certain conditions, similar to various other languages. The Wick language uses for and while loops. A for loop is used when the number of iterations is known prior to the loop's execution. It is especially useful for iterating over data structure elements (such as arrays or lists) or repeatedly running a code block. A while loop is used when the number of iterations is unknown and the loop must run until a certain condition changes. The while loop starts by evaluating a condition. If the condition is met, the code within the loop is executed. After the code block is performed, the condition is re-evaluated, and this process continues until it is false. This is an example of loops with the Wick language.

```
variable i = 0;
while i < 5 {
    print("While Loop");
    i = i + 1;
}



for k = 0; k < 5; k = k + 1 {
    print(k);
}
```

2.6 - Lambdas and Subroutines

Wick has support for anonymous subroutines called “lambda expressions.” These constructs can be used to implement the behavior of a subroutine without necessarily having to associate it with a name. Because lambda expressions and subroutines are first class data types in Wick, they can even be passed into other subroutines. Additionally, subroutines can return values that they have calculated. See below for an example of a lambda expression being passed into a subroutine:

```
subroutine distanceAt(f, g, x) {
    return abs(f(x) - g(x));
}

print(
    distanceAt(
        lambda (x) { return 2 * x; },
        lambda (x) { return x * x; },
        1
    )
);
```

Subroutines are merely syntactic sugar for a variable storing a lambda expression. These two are the same:

```
subroutine distanceAt(f, g, x) {
    return abs(f(x) - g(x));
}

variable distanceAt = lambda (f, g, x) {
    return abs(f(x), - g(x));
}
```

2.6.1 - Default Parameters

Wick also supports the use of default parameters. Simply put, if a subroutine is not provided with an argument when being called and it has a default initializer, the value will be the default given. See:

```
subroutine introduceYourself(name = "stranger") {
 print("Howdy " + name + ".");
}

introduceYourself("Evelin");
introduceYourself();
```

2.6.2 - Recursion

Subroutines can call themselves. See this standard implementation of an algorithm to find the greatest common factor between two numbers:

```
subroutine fib(n) {
    if(n <= 1) {
        return n;
    }
    return fib(n - 2) + fib(n - 1);
}

print(fib(10));
```

2.6.3 - Stopping Early

Subroutines may stop themselves early using an empty return statement if desired.

```
subroutine onlyPrintIfOne(x) {
    if x != 1 {
        return;
    }
    print("It's 1!");
}

onlyPrintIfOne(0);
onlyPrintIfOne(1);
```

2.7 - Prototypes

Wick provides support for object-oriented programming via the use of prototypes. Prototypes are similar to both the concept of a class and an object - as if they were combined. The declaration of a prototype creates a “prototypical object” that can be cloned. This cloning process is done via the prototype’s constructor. Javascript developers and those familiar with the prototype pattern will see the resemblance of prototypes in Wick to other languages. Similar to subroutines, prototypes are simply syntactic sugar for variables storing anonymous prototypes. This means Wick allows statements such as:

```
introducePerson(prototype from Person {
public:
    subroutine introduce() {
        print("Hi, I'm " + name + " and I'm in witness protection!");
    }
private:
    variable name = "[REDACTED]";
});
```

2.7.1 - Visibility

Unlike some other implementations of prototypes, Wick provides support for visibility modifiers. This means certain properties of prototypes can be given additional protection. There are two relevant visibility modifiers: public, allowing clients to access, and private, allowing only the prototype itself to access.

```
prototype Evan {
public:
    variable name = "Evan";

private:
    variable ssn = 110305901;
}

print(Evan.name);
print(Evan.ssn); // Private variable requested!
```

2.7.2 - Constructors

Prototypes are cloned via their constructor. The constructor creates a new prototype, copying the public and private fields into their appropriate location, before calling its associated behavior to initialize the prototype.

```
prototype Person {
constructor(iName) {
    name = iName;
}
public:
    subroutine introduce() {
        print("Hello, my name is " + name + ".");
    }

private:
    variable name;
}

variable everett = Person("Everett");
variable preston = Person("Preston");
everett.introduce();
preston.introduce();
```

2.7.3 - Inheritance

Prototypes can inherit from other prototypes. This means that the behavior associated with the parent is pulled into the implementation for the child prototype. These behaviors can even be overridden.

```
prototype Doctor from Person {
constructor(iName) {
    name = iName;
}
public:
    subroutine introduce() {
        print("Salutations, my name is Dr. " + name + ".");
    }
}

variable ghosh = Doctor("Ghosh");
ghosh.introduce();
```

2.7.5.1 - Parent

Inheritance in Wick is very similar to cloning via a constructor, but with mostly an implied difference of intent. However, inheritance does set a special private variable, parent, to a reference to the parent of the prototype. This is dissimilar from the “super” keyword found in other languages in that parent refers to a literal prototype.

```
prototype LateGoodman from goodman {
public:
    subroutine introduce() {
        print("Finally, here we are!");
        parent.introduce();
    }
}

Wrapper.introduce();
```

2.7.4 - Polymorphism

The power of prototypes is primarily in the ability to make use of polymorphism. Polymorphism allows users to pass various different prototypes into a structure like a subroutine so long as they share an outward appearance, or, interface. See this example building off the Person prototype from before:

```
prototype Lawyer from Person {
constructor(iName) {
    name = iName;
}
public:
    subroutine introduce() {
        print("Objection! My name is " + name + "! Attorney-at-law, " + name + ".");
    }
}

variable goodman = Lawyer("Goodman");
goodman.introduce();

subroutine introducePerson(person) {
    person.introduce();
}

introducePerson(everett);
introducePerson(preston);
introducePerson(ghosh);
introducePerson(goodman);
```

2.7.5 - This

The “this” keyword is associated with all prototypes and stores a reference to the prototype itself.

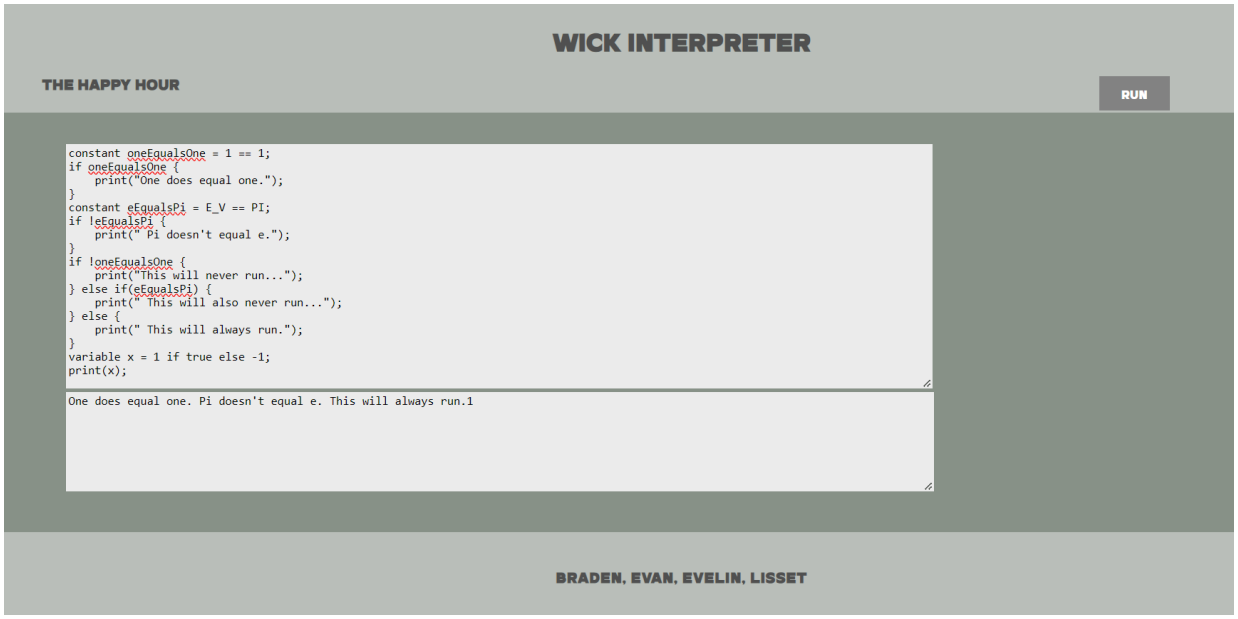
```
prototype ThisTest {
public:
    subroutine introduce() {
        print("I exist to demonstrate Wick!");
    }

    subroutine getInstance() {
        return this;
    }
}

ThisTest.getInstance().introduce();
```

3 - Online IDE and Tutorial

The online IDE is called the “Wick Interpreter”. The Integrated Development Environment purpose is designed to help you write, test, and debug/execute the Wick language effortlessly online.

3.1 - Tutorial

To initiate the Wick Interpreter IDE, guarantee you have all the appropriate files downloaded. In your terminal, navigate to the directory containing the files and run the command "node server.mjs" to start the server. Then, open the "index.html" file in your web browser to access the IDE. Once inside the IDE, you'll find the main area designated as the code editor, where you can write your Wick code. After entering your code, simply click the "RUN" button to execute it and see the results instantly. The output area below the editor displays the result of your Wick code execution in real-time. With the Wick Interpreter IDE, writing, executing, and debugging Wick language code has shown a great way to educate ourselves. Down below I will have an example of it being used with our Wick language and conditionals being involved.



4 - Appendix

4.1 - Project Timeline

8/23/2023 - Group is formed.
9/7/2023 - Decide on project.
9/8/2023 - Discord set up and everyone in.
9/12/2023 - Layout initial goals for Wick.
9/19/2023 - Decide to research interpreters and compilers in depth before work commences.
10/17/2023 - Basic description of the Wick language complete.
11/7/2023 Finish setting up tools for the project.
11/8/2023 - Create basic outline of the scanner.
11/14/2023 - Create basic outline of the parser.
1/24/2024 - First meeting of 2024; discussed remaining project timeline.
1/27/2024 - Finish scanner.
1/28/2024 - Create outline of parser.
2/6/2024 - Decide to create a web IDE.
2/12/2024 - Evaluation of basic expressions complete.
2/28/2024 - Support for variable definitions.
3/1/2024 - Support for scoping, if statements, and while-loops; the language is Turing-complete.
3/6/2024 - Support for for-loops.
3/11/2024 - Added native subroutines and subroutine calls.
3/12/2024 - Collate IDE efforts on GitHub repository.
3/19/2024 - Correct variable binding issue; add default parameters.
3/27/2024 - Finalize layout for the IDE; add in prototypes.
4/1/2024 - Begin formalizing documentation and testing efforts.
4/10/2024 - Finalize the interpreter at this time.
4/30/2024 - Basic functionality of the IDE complete.

4.2 - Special Considerations

There are several topics that are worthy of special consideration in this manual. The first being any liability concerns within the language or interpreter, for a better understanding of what this means with liability concerns is for example if a company or individual user was to utilize the project to create a very complex program to perform very strenuous tasks (for example: programs a navigation type system for a commercial passenger plane). These types of examples could possibly pose a liability concern for the project and the happy hour team.

Due to the limit of time wick has been given towards the project, wick has encountered a few obstacles, one of these being testing limitations. What the project is trying to imply about testing limitations is that it has only had a very short period to test the limits and capabilities of the language and interpreter. The best example that can compare to it is the g++ compiler, by comparison wick is attempting to demonstrate and show the length and duration this compiler has been published, tested, and debugged. The language and its interpreter has only had the opportunity to be tested for a few months and can not justify the complete testing of the project when placed next to another project like g++ that has been tested and updated for 37 years.

Legacy programs are also a special consideration that needed to be discussed. After completion of our project and the publication of the Wick language and interpreter to the public for anyone to use or

review many have to consider the inevitability of continued work being published after an individual has already employed the use of the project. This can lead to some compatibility issues that can render a user's program useless if the original source code has been altered or newer features have been released. Performance concerns have also been discussed with time limitations for the project. Due to a shorter time frame given the project has had to cut back a vast amount of the optimization step in the crafting of the interpreter, this can lead to several performance issues within the project itself. This may have several effects such as slower compile times that would generally slow things down when using the interpreter.

Wick will also have to make sure that proper credit is given to Robert Nystrom for his excellent resource Crafting Interpreters. While our project has deviated greatly from the language described in his text, his book was instrumental for the beginning stages of this project. Thank you for the wonderful source reference. A similar sentiment is extended to Michael Scott, the author of Programming Language Pragmatics for providing the perfect textbook to gather a theoretical basis of programming languages.

The last special consideration that has been discussed is the overall reliability of the online IDE. When the project is published the desire is to provide a very simple online IDE to showcase the Wick language and its interpreter for people to easily access instead of through our repository and VS code. For example this can pose a concern if the IDE is not correctly set up and polished and hosted on its own web server that can allow multiple users to access at once without disrupting the IDE or its services to all users. (slower or breaks for some if not all users).

5 - Conclusion

While Wick is a Turing-complete language that isn't to say that it is a complete language. In continuing our work, we would hope to provide additional semantic analysis including an optimization step and potentially static typing. Also necessary would be a transition from tree-traversal interpretation to compiling to bytecode or an intermediate language. Wick would also greatly benefit from meager, tiny changes such as file IO, exception handling, a package manager, a standard library, and more.

Throughout this project we created a new programming language, an interpreter to recognize and execute its programs, and an IDE to edit Wick code. In doing so, we have gained invaluable knowledge regarding languages that will serve as an excellent foundation for our computer science education. We hope that Wick and the deliverables produced alongside it will serve as a valuable learning resource for others interested in compiler and programming language design.

6 - Citations

- Thank you to Professor Bhaskar Ghosh and Professor Brian Chirgwin for guidance throughout this project.
- Node.js. “Node.js.” Node.js. 2023, nodejs.org/en.
- Michael L. Scott, Programming Language Pragmatics, 4th ed., Burlington, MA: Morgan Kaufmann, 2015
- Nystrom, Robert. Crafting Interpreters. Genever Benning, (2021).
- Roberto Lerusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua 5.1 Reference Manual. Lua. (2019, August). [Online]. Available: https://www.lua.org/manual/5.1/manual.html
- Community resource. C++ Reference. C++ Reference. (2023, September). [Online]. Available: https://en.cppreference.com/w/
- Steve Klabnik and Carol Nichols. The Rust Programming Language. The Rust Programming Language. (2023, February). [Online]. Available: https://doc.rust-lang.org/book/