DESIGN AND IMPLEMENTATION OF A NOVEL PROGRAMMING LANGUAGE THAT PROMOTES CLEAN CODING

WONG JIA HAU

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Science (Hons.) Software Engineering

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

April 2019

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : _____

Name : Wong Jia Hau

ID No. : 15UEB00181

Date : _____

APPROVAL FOR SUBMISSION

I certify that this project report entitled "**DESIGN AND IMPLEMENTATION OF A NOVEL PROGRAMMING LANGUAGE THAT PROMOTES CLEAN CODING**" was prepared by WONG JIA HAU has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Science (Hons.) Software Engineering at Universiti Tunku Abdul Rahman.

Approved by ,

Signature :

Supervisor :

Date :

Signature :

Co-Supervisor :

Date :

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisors, Dr. Victor Tan and Dr. Madhavan for their invaluable advice, guidance and enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me encouragement to develop this project.

Also, I would like to thanks the Reddit Programming Languages Community (https://www.reddit.com/r/ProgrammingLanguages/) that have provided a lot of constructive criticism on my language design. Not only that, I am also inspired by their language design. Also, in that community I am exposed with a lot of materials regarding programming languages, such as garbage collections, compiler design, compiler optimization, type theory and so on.

Lastly, I would like to thank the creator of Java Generics, Mr. Philip Wadler, who had given me precious insights on how to implement generics.

ABSTRACT

Software maintenance is one of the most expensive software activity, implying that code maintainability is undeniably important in any software project. Many software engineers thus enforce strong disciplinary on writing clean code. However, such skill is hard to achieve even by experienced programmer. Therefore, this study aims to investigate various programming language features or design that causes such difficulties. The main factors discovered are difficulties in creating understandable functions, uncertainty in creating new functions, difficulties in extending existing classes, implicit coercion, implicit variable mutability etc. The primary solutions for the aforementioned factors are mixfix functions, separation of class definition from interface implementation and functional purity. In this project, a new programming language, named Keli that solves the difficulties in writing clean code was implemented. Based on the evaluation result, Keli seems to achieved its objective as most respondents thought that it is particularly comprehensible and learnable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
ES	ECMAScript, the international standard for JavaScript. For example, ES6 stands for ECMAScript 2016.
I/O	Input output operations. For example, printing string to a console screen.
RISC	Reduced Instruction Set Computer
UTAR	Universiti Tunku Abdul Rahman
VSCode	Visual Studio Code

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.0 Introduction

This chapter shall discuss the background of the problem, problem statements, project objectives, proposed solution, proposed approach and the scope of the project.

1.1 Background Research

In software maintenance, code reading is one of the most recurring activity, because developers have to understand the underlying source code before they can make any changes to it (Scalabrino et al., 2016). According to Glass (2001), software maintenance usually consumes about 60% of software cost on average. Also, there were a number of papers estimated that around 50% of programming resources goes into maintenance, with some citing as high as 75% (Lientz, Swanson and Tompkins 1978; Pearse and Oman, 1996; Galorath, 2017). On top of the project development. Since software is mostly developed with code, it is undeniable that the maintainability of the code directly affects the overall maintainability of the software.

According to Martin (2011), poorly written code caused significant loss of resources to a development organization every year. In addition, many large-scale software project suffers multi-million dollar lost due to programming error at, Jarzabek (2007) even stated that software maintenance is the most expensive software activity. Thus, it is notable that software maintenance is one of the most important aspect in softwarat the syntactic or semantic level, rather than the algorithmic or system-levels (Gopstein, et. al, 2017). Pradhan (2008) also states that one of the main culprit of software bugs is poor coding practices. Moreover, a was-popular terminal multiplexer application, namely GNU Screen, dig its own grave due to its unreadable and unmaintainable codebase (Perrin, 2010).

However, it is widely accepted that clean code is not easy to write, as most programming language design did not emphasize usability (Software Engineering StackExchange, 2013; Pane, Myers and Miller, 2002). Thus, this paper aims to investigate the reasons that causes the *difficulties of writing clean code*, and shall propose a solution, which is a new programming language (called Keli) that shall resolve the stated problems.

1.2 Problem Statements

In this section, this paper shall described the *difficulties of writing clean code* (DOWCC). In another words, it attempts to answer the following question:

Why is clean code so hard to write?

Or in other words:

Why code tend to be unclean?

Although human-factor such as discipline play a major part on code maintainability, this paper shall not discuss any of that factor, instead, programming language design shall be the center of attention.

To summarize, below are the factors or reasons that causes DOWCC, which is mostly based on direct observations upon the most widely used programming languages (see Table 1):

- Difficulties in creating understandable functions
- Uncertainty of designing function interface
- Difficulties in extending functionalities of existing class
- Implicit coercion
- Implicit mutability of variables
- Absence of type annotation
- Context-sensitive symbols

Rank	Language	Ratings (%)
1	Java	15.36
2	C	14.94
3	C++	8.34
4	Python	5.76
5	C#	4.31
6	Visual Basic .NET	3.76
7	PHP	2.88
8	JavaScript	2.50
9	SQL	2.34
10	R	1.25

Table 1: The top ten most used programming languages as of June 2018 (TIOBE, 2018).

1.2.1 Difficulties In Creating Understandable Functions (DICUF)

I believe that one of main factor that prevents programmers from writing clean code easily is due to *difficulties in creating understandable functions (DICUF)***.** This problem can actually be traced back to assembly language.

```
MOV A, B
```

Figure 1: A simple assembly operation

From the code in Figure 1, it is impossible to tell whether it means *move value of A to B* or *move value of B to A*, unless one look the definition of MOV. This problem of argument position confusion have been carried throughout the evolution of programming languages, from procedural to functional to object-oriented, and it still persist in most of the high level language.

To truly understand what DICUF is**,** one should look into two programming language paradigms, namely *procedural* and *object-oriented*. To demonstrate, examples on creating a function that carries the following meaning shall be illustrated using Python in the following section:

replace some character with a new character in a string.

A. The Procedural Approach

To define the required function in a procedural manner, there are a few variants:

```
# Variant 1
def replace(target, oldChar, newChar):
     pass

# Variant 2
def replace(target, newChar, oldChar):
     pass

# Variant 3
def replace(oldChar, newChar, target):
     pass

# Variant 4
```

```
def replace(newChar, oldChar, target):
     pass
```

Figure 2: Variants for defining the replace function in procedural manner

From Figure 2, it is evident that there are multiple ways to define the API for the *replace* function. However, none of them will be truly elegant**,** because each of the variant carries some degrees of confusion to the client (see Figure 3).

```
 comma    = ","
dot      = "."
myString = "John,James,Jane"

# Statement 1
replace(myString, comma, dot)

# Statement 2
replace(myString, dot, comma)

# Statement 3
replace(dot, comma, myString)

# Statement 4
replace(comma, dot, myString)
```

Figure 3: Confusions of invoking the procedural replace function

From Figure 3, it is almost impossible to tell which statement truly bears the meaning of *replace comma with dot in myString*. Due to this issue, programmers who intend to use this function might misplace the argument, due to the fact that they have to memorize the argument position correctly in order to bring out the desired semantics. This situation also correlates to the violation of one of the Shneiderman's Eight Golden Rules for Interface Design, which is *Reduce short-term memory load* (Interaction Design Foundation, 2018).

B. The Object-Oriented Programming (OOP) Approach

The previous section had shown that defining a clear and concise interface for the *replace*

function in procedural manner is hard. This section aims to investigate if OOP could improve the situation.

```
class String:
    # Variant 1
    def replace(self, oldChar, newChar):
        pass

    # Variant 2
    def replace(self, newChar, oldChar):
        pass
```

Figure 4: Variants for defining *replace* function using OOP

From Figure 4, although it is apparent that the number of variants decreases greatly as compared to that of procedural approach, however, there are still some space for confusion to happen, as illustrated in Figure 4.

```
# Statement 1
myString.replace(comma, dot)

# Statement 2
myString.replace(dot, comma)
```

Figure 5: Confusions of invoking the object-oriented *replace* function

From Figure 5, it is apparent that it is still impossible to tell which statements shall bear the meaning of replace *comma with dot in myString*. It is arguable that Statement 1 seems to be more correct, however, that assumption is purely based on context, thus the only way to resolve such confusion is still by looking up the definition of the function. This situation also suggests that one of the main principle of OOP, namely *encapsulation* had been broken, as the client have to unearth the function definition in order to understand how to use it properly. In a nutshell, there are still space for confusion and argument-misplacement to happen even with OOP.

Moreover, the creator of AWK, Prof. Brian Kernighan (University of Nottingham, 2015) mentioned that he could never remember the order of the arguments (for a C function

named fgets). To further strengthen this statement, I actually had done a short survey (mostly students from Software Engineering UTAR). From the results obtained, it is evident that confusion arises even with OOP, as illustrated in Figure 6 and Figure 7. Thus, it is clear that even OOP cannot resolve DICUF properly.

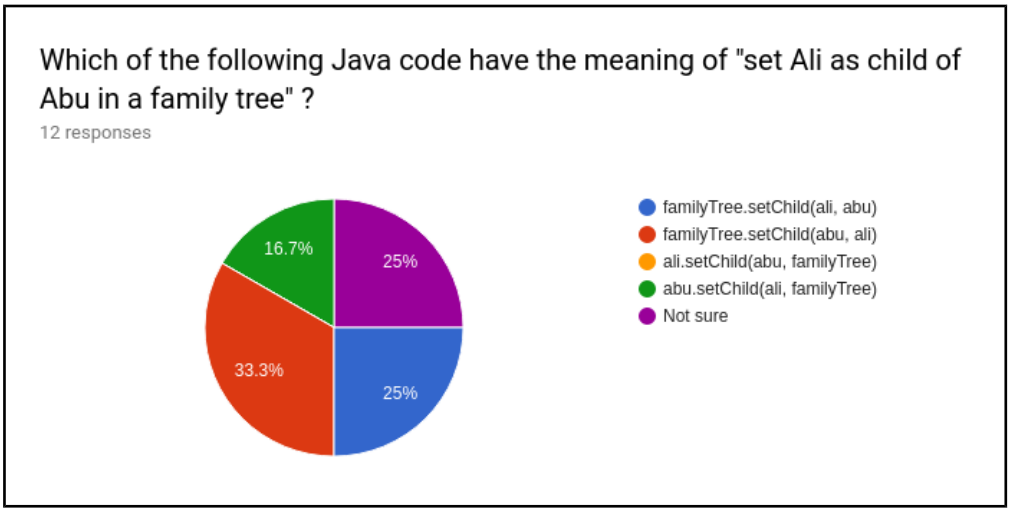

Figure 6: Respondents assume varying meanings for an OOP method invocation

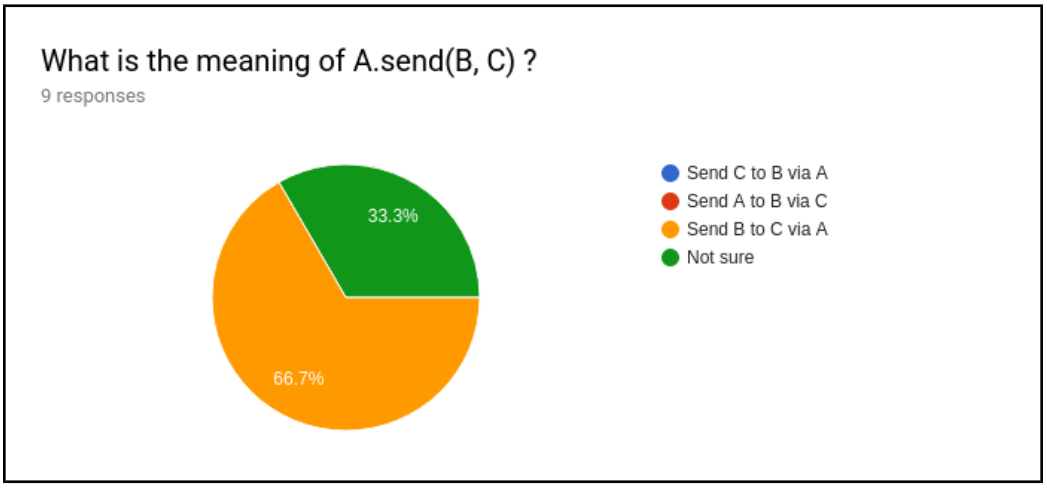

Figure 7: More than 30% of respondents could not deduce the meaning of a simple OOP method invocation statement

C. Definition Of DICUF

Based on the previous findings, one could see that the underlying problem is the inevitability of creating confusion**.** It means that there are spaces for confusion when defining a function interface that takes more than one parameters, due to the fact that the correct argument position cannot be deduced confidently during the first glance on the

function call. Thus, the terminology DICUF shall refer to such phenomena.

D. Causes Of DICUF

The main causes of DICUF is that, in most programming language, the function declaration/definition are always prefix-oriented, meaning that the function name must be at front, and the parameters must be at the back, as shown in Figure 8, which is not much different than assembly language (as illustrated in Figure 1).

```
def function_name(param1, param2, param3):
    pass
```

Figure 8: Prefix-oriented function definition.

Thus, it is notable that although high level language abstracted a lot of concept and syntax of low level language like assembly, the problematic heritage of argument position confusion still persist in modern programming languages.

1.2.2 Uncertainty Of Designing Function Interface (UDFI)

The second factor that contributes to the difficulty of writing clean code is UDFI. The following section shall elaborate further on the term UDFI by using examples from Python. (Note that the term *function* and *method* shall be used interchangeably). In layman terms, UDFI can be understood as "*It's hard to design a function interface properly"*.

A. Excessive Constructs For Defining A Function

One of the main factor that causes UDFI is due the excessive constructs for defining a function. In most popular programming languages, one is able to define a function using the following construct: Function, Instance method, Static method or Constructor (see Figure 9).

```
# Variant 1: Define it as a static method under the Integer class
class Integer:
    @staticmethod
    def parse(input):
          pass

# Variant 2: Define it as an constructor under the Integer class
class Integer:
    def __init__(input):
          pass

# Variant 3: Define it as an instance method under the String class
class String:
    def toInteger():
          pass

# Variant 4: Define it as a function
def parseIntegerToString(input):
    pass
```

Figure 9: Defining *convertToInteger* function using various construct

```
input = "123"

# Using variant 1
result = Integer.parse(input)

# Using variant 2
result = Integer(input)

# Using variant 3
result = input.toInteger()

# Using variant 4
result = parseIntegerToString(input)
```

Figure 10: Invocation of various *convertToInteger* function

As illustrated in Figure 10, the usage of each variant actually do convey a clear purpose (which is to convert integer to string), however the function designers might need to undergoes a lot of cognitive workout in order to figure which variant suits best for the project design. Unfortunately, none of the variant will be the best, as it is solely based on the designers programming style. Thus, the terminology *Uncertainty of Designing Function Interface (UDFI)* is used to describe such phenomena. Due to UDFI, some programmers might avoid refactoring their code into smaller components, as there are just too many variants for designing a function interface. As a consequences, the code base have a tendency to become dirty overtime, unless a strict discipline is enforced.

Furthermore, UDFI also causes inconsistencies in libraries function. For example (in C#), one could convert a variable to string value by calling instance method:

```
myVariable.ToString()
```

Based on that, one might thought that the following code is the right way to convert a variable to integer:

```
myVariable.ToInteger()
```

Unfortunately, it is incorrect, one should use static method instead:

```
int.Parse(myVariable)
```

Thus, this prove that UDFI do causes inconsistency, and thus violates another principle of Shneiderman's Eight Golden Rules for Interface Design*:* which is *Strive for Consistency* (Interaction Design Foundation, 2018).

B. Mutability Of Function Inputs

Another main factor that contributed to UDFI is the mutability of function inputs. In other words, it means that function are able to modify their inputs. For example, the *sort* function can be defined in two variants within Python (see Figure 11).

```
 # Variant 1
def sort(list):
     # Sort the list and return nothing
     # Mutate the input

# Variant 2
def sorted(list):
     # Does not mutate that input
     # Return a sorted list
```

Figure 11: Defining *sort* function in various way

```
# Variant 1
myList = [4,3,2,1]
sort(myList)

# Variant 2
myList = [4,3,2,1]
myList = sort(myList)
```

Figure 12: Invocation of each *sort* functions

From Figure 11 and Figure 12, it can be seen that in Python (this also applies to other top 10 most popular programming languages as in Table 1), there are 2 ways to define a function:

A. One that will mutate the input and return nothing
B. One that will not mutate the input and return a new result

Due to this reason, whenever a programmer wants to create a function, they have to consider carefully whether to use option A or option B, thus increasing the uncertainty of defining function interface. In conjunction with Section 1.2.2(A), there would be 4 x 2 = 8

ways to define a simple function.

C. Ability To Have Global Variables

The ability to have global variables, otherwise known as out-of-scope variable is another reason that contributed to UDFI. It is commonly known that global variables are bad (Stack Overflow, 2009), thus this section shall not further elaborate it. However, this section aims to illustrate how the *ability to have global variables* will causes UDFI. Consider an example where a programmer is require to write a function that calculates the addition of 2 numbers based on user input. The first approach is to use global variables (see Figure 13).

```
string userInput1; // Global variable
string userInput2; // Global variable

void main() {
      print("Enter two numbers");
      userInput1 = readLine();
      userInput2 = readLine();
      int answer = calculate();
      print("The result is" + answer);
}

int calculate() {
      int num1 = convertToInteger(userInput1);
      int num2 = convertToInteger(userInput2);
      return num1 + num2;
}
```

Figure 13: Function that uses global variables

The second approach is to write the function without using global variables (see Figure 14):

```
void main() {
      print("Enter two numbers");
      userInput1 = readLine();
      userInput2 = readLine();
      int answer = calculate(userInput1, userInput2);
      print("The result is" + answer);
}

int calculate(string userInput1, string userInput2) {
      int num1 = convertToInteger(userInput1);
      int num2 = convertToInteger(userInput2);
      return num1 + num2;
}
```

Figure 14: Function that does not use global variables

Figure 13 and Figure 14 illustrated that there are 2 ways to define such simple function. Thus, it further increases more ways to define a function. By including factors from Section 1.2.2(A) and Section 1.2.2(B), one have at least 4 x 2 x 2 = 16 ways to define a function.

D. Ambiguous Arguments Position

The other factor that causes UDFI is the problem of ambiguous arguments position that arise when one needs to declare a function that takes more than 2 arguments (this problem is also discussed in). For example, one have two ways to declare a function that takes 2 parameters (see Figure 15).

```
// Language: JavaScript
// Variant 1
function writeFile(fileStream, data) { /*body*/ };

// Variant 2
function writeFile(data, fileStream) { /*body*/ };
```

Figure 15: Variants in defining 2-arguments function

This situation certainly increases the uncertainty in designing function interface (UDFI) as the designers have to pick 1 variant out of the 2 variant which are equally understandable. By adding factors from the previous subsection of section 1.2.2, there will be at least 4 x 2 x 2 = 32 ways for defining a 2-arguments function (global variables is not included in the calculation, because the usage of global variables will causes the problem of ambiguous argument position to be nonexistent). Clearly, this problem shall get worse if there more than 2 arguments for the function.

Also, this condition might causes inconsistencies in library functions if style guide is not enforced strictly, for example in PHP, some built-in functions expect *haystack* to be the 1st argument while *needle* as the 2nd argument, yet, some expect the inverse, as shown in Figure 16 (Software Engineering StackExchange, 2010b).

```
bool array_key_exists(mixed $key, array $array);

bool property_exists(mixed $class, string $property);
```

Figure 16: Inconsistency of haystack-needle functions in PHP

Certainly, this complication can be avoided via strict discipline, but such practice requires high cognitive effort, thus it is not easy to avoid function signature inconsistencies.

1.2.3 Difficulties In Extending Functionalities Of Existing Classes

Another factor that causes DOWCC is the difficulties in extending functionalities of existing classes. This condition can also be perceived as *"It is hard to add new features for existing classes or compiled classes"*. To illustrate, assume that there is an existing class which should not be modified (see Figure 17).

```
class Person {
      public name : string;
      public age  : number;
}
```

Figure 17: Definition of the Person class

If one wish to implement the Comparable interface for the class *Person* without modifying the original class, one might need to create a subclass for Person (see Figure 18).

```
class ComparablePerson extends Person
      implements Comparable<ComparablePerson> {
      public compareTo(other: ComparablePerson) {
            if(this.age === other.age) return 0;
            else if(this.age > other.age) return 1;
            else return -1;
      }
}
```

Figure 18: Extending functions of Person via inheritance

At the first thought, this might seems acceptable, but if some other client that uses ComparablePerson wish to make the class implements Serializable**,** he/she will need to perform the same steps again, which is:

1. Create a new class with a new name (e.g. SerializableComparablePerson)
2. Extend the existing class
3. Implements the desired interface

The consequences of such phenomena is that the name of the class becomes less and less meaningful, thus causing the original intention of the class to be less obvious. Consider the

word *ComparablePerson*, is the main focus on *Comparable* or is it on *Person*? To some, it is clear that *Person* is the main focus; to others, they might thought that *Comparable* is the important one. Thus, this might reduce the understandability of the code as the intention of each class becomes less obvious when new features are extended from existing class. Moreover, the class hierarchy might grow into an unmanageable pyramid (see Figure 19).

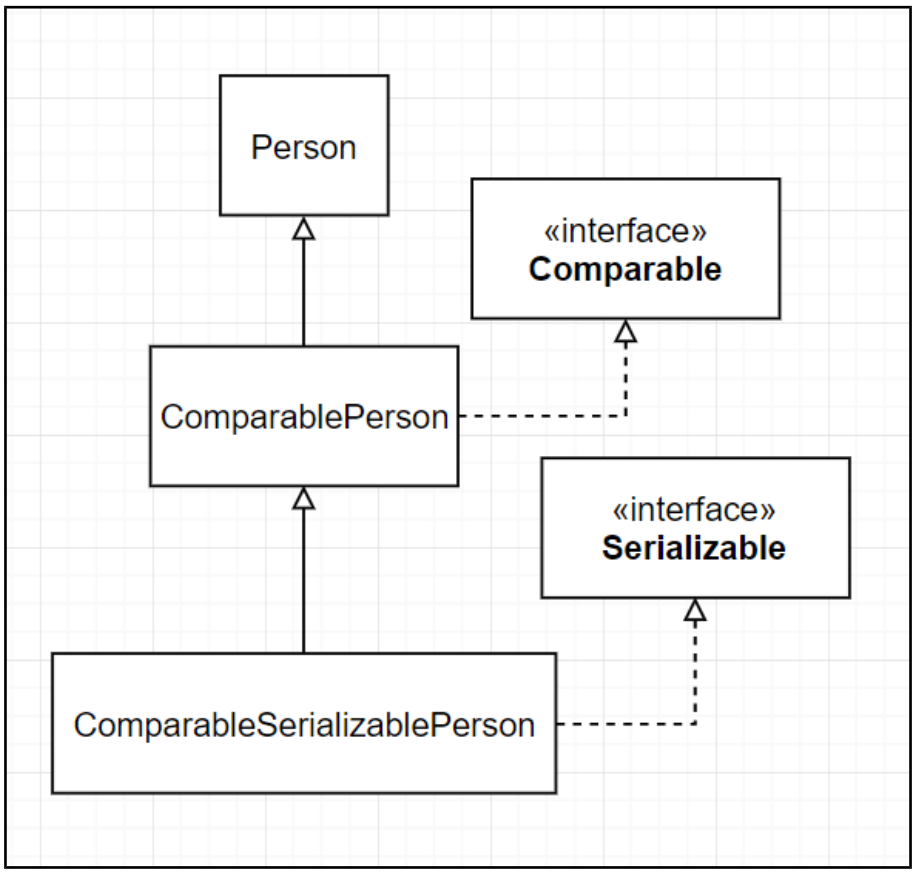

Figure 19: Inheritance's pyramid of doom

1.2.4 Unclear Distinction Between I/O And Non-I/O Operations

Another factor that causes difficulties of writing clean code (DOWCC) is that there are no clear distinction between I/O operations and non-I/O operations. Due to this factor, the debugability of a piece of code can be significantly reduced. To demonstrate, consider the function as shown in Figure 20.

```
function sendRequest() {
      let result1 = performCalculation();
      let result2 = performAnotherCalculation(result1);
      let result3 = checkCorrectness(result2);
      send(result2);
}
```

Figure 20: Unclear distinction between I/O and non-I/O operations

Suppose the function above is found to be erroneous, the programmer who will be fixing it may have hard time finding the actual bug, since it is impossible to tell which function actually involves I/O operations. This is because most of the bug occurs at functions that contains side-effects. In fact, according to Volkmann (2009), functions without side-effects (aka. pure functions) are much easier to understand, debug and tested than those with side-effects. Therefore, if the distinction between I/O functions and non-I/O functions is unclear, it will be hard for a program to be debugged. However, it should be understood that side-effect is necessary in order for a software to be truly useful.

1.2.5 Absence Of Type Annotation

Another factor that contributed to the difficulties of writing clean code is the absence of type annotation. Programming languages that relates to this factor are Python, JavaScript and R. This is because it reduces the understandability of the code.

```
// JavaScript
function draw(shape) {
     // draw some shape
}
```

Figure 21: A function without type annotation

The code presented in Figure 21 might seem innocent, but it is the culprit of confusion, because there are several mental possibilities to call that function (see Figure 22).

```
// Way 1: Specifying a shape name
draw("circle")

// Way 2: Specifying coordinates
draw([[0,0],[1,2],[3,4]])

// Way 3: Specifying an object
draw({shape: "circle", color: "red"})
```

Figure 22: Possible ways to invoke the draw function

Certainly, the possibilities can be expanded ad infinitum. To identify the actual parameter type, one have to look up documentation, and explicitly memorized the type. In case where the same function is to be reused, the programmer have to recall back the parameter type again. Thus, it is evident that weak typing creates confusion and increase cognitive burden. Again, this situation is violating the *Reduce short-term memory load* principle of *Shneiderman's Eight Golden Rules for Interface Design* (Interaction Design Foundation,

2018).

1.2.6 Implicit Mutability Of Variables

Implicit mutability of variables is another that causes DOWCC, as it reduces debuggability.

```
// JavaScript
var a = 3, b = 4, c = 5;
var d = 6;
for(var i = 0; i < d; i += a) {
      for(var j = d; j > b; j -= c) {
            a += 4; b--; a--; c++;
      }
      c += b; a -= -1; b++; c--;
}
alert (d); // What is the value of d?
```

Figure 23: Demonstration of implicit mutability

The gibberish JavaScript code in Figure 23 is used to represent actual production code that might span up to hundreds of lines. It is plain to see that the final value of d is not easily determinable. If more attention is given, one might find that d is actually not mutated at all. Thus, if the code can be rewritten as follow, it might be easier to reason about the value of d (See Figure 24).

```
var a = 3, b = 4, c = 5;
const d = 6; // mark d as constant
for(var i = 0; i < d; i += a) {
      for(var j = d; j > b; j -= c) {
            a += 4; b--; a--; c++;
      }
      c += b; a -= -1; b++; c--;
}
```

Figure 24: Immutability of variable enhances debuggability

Since d is marked as a constant, it shall not be mutated throughout the program, if not the compiler will raise error about it. Thus, one can quickly deduce the value of d without needing to going through the whole program, therefore this proves that immutability will improve debuggability. This is one of the reason why code quality checker such as ESLint

(aka. JavaScript Linter) require const declaration for variables that are never reassigned after declared (ESLint, 2018b). Moreover, Facebook even created a JavaScript library called Immutable.js just to seize the benefits of immutability out of JavaScript (Immutable.js, 2018).

Despite the reasons aforementioned, the other benefits of immutability includes: simple state management; thread-safety; and safe and efficient sharing (Bloch, 2017). However, programmers often prefer mutability, because mutability is implicit, which requires less typing, for example, in JavaScript, one would have to type const instead of var (which is 2 characters lesser); in Java, one would have to type final instead of typing nothing (which is 5 characters lesser) to declare immutability. This statement can be further strengthen by the *Principle of Least Effort*, which postulates that humans will naturally choose the *path of least resistance*, in this case the preference of mutability, as it requires less typing (Zipf, 2016). Due to this reason, the code (written using languages in Table 1) tends to becomes less debuggable.

1.2.7 Implicit Coercion

Another reason that decreases code understandability and debuggability is implicit coercion. Implicit coercion can also be understood as *the conversions of data-type that happens silently.*

```
let result = 9 * "2";  // "2" is implicitly converted to 2
alert(result); // 18
```

Figure 25: Implicit coercion in JavaScript

In Figure 25, the string literal "2" is silently converted to integer 2, causing the variable result to have the value of 18. At first this might seems convenient, however in the long run, this feature will causes difficulties in debugging.

```
let input = window.prompt("Enter a number");
let result = input + 100;
alert(result);
```

Figure 26: Unexpected behavior due to implicit coercion in JavaScript

At first glance, one might thought that the code in Figure 26 will display the number given by a user added by 100. However, if a user entered 99, the answer will be displayed as 99100 instead of 199. The problem that occurred here is due to the implicit coercion of integer 100 into string "100". To some programmers, the solution might be obvious, which is to convert input into integer by perhaps calling the parseInt function. But, if the code above shall span more than 20 lines, even an experienced programmer might end up in a long-lasting debugging loop. This is the reason why ESLint disfavor the usage of implicit coercion, because it is thought the code will be less readable and understandable (ESLint, 2018a).

1.2.8 Context-Sensitive Symbols

Another factor that causes code to be less readable are context-sensitive symbols. Such symbols can causes inconsistency in the language syntax. For example in JavaScript, curly brackets can have several meanings depending on the context (see Figure 27).

```
// As statements wrappers
let myFunction = function () { statement1; statement2; };

// As object wrappers
let myObject = {x:1 , y:2};

// As string interpolation wrappers (ES6 and above only)
let message = `My name is ${myVariable}`;

// As expressions wrapper (in React-JSX only)
let component = <View>{myVariable}</View>
```

Figure 27: Different meaning of curly braces in JavaScript

Some might think it is easy to understand, however, when all of them appears at the same place, it will be hard to explain to others who are not familiar with it (see Figure 28).

```
let Header = function() {
      return (
            <View style={{fontSize: 15, fontWeight: "bold"}}>
                  {`My name is ${myName}`}
            </View>
      );
}
```

Figure 28: Curly braces with different meanings appear altogether at once

Tashtoush, Odat, Alsmadi, and Yatim (2013) stated that one of the characteristics that have a high positive impact on the readability of code is *consistency.* In other words, context-sensitive symbols causes code to have lower readability.

1.3 Project Objectives

The aim of this project is to investigate various programming language features or design that causes such difficulties. The main factors discovered are difficulties in creating understandable functions, uncertainty in creating new functions, difficulties in extending existing classes, implicit coercion, implicit variable mutability etc. The primary solutions for the aforementioned factors are mixfix functions, separation of class definition from interface implementation and functional purity.

The objectives of this project are:

1. To identify the problems of existing programming languages.
2. To design and implement a programming language that will encourage user to write *clean code.*

1.4 Proposed Solutions

As demonstrated in Section 1.2, the current mainstream programming languages have several flaws that does not allow programmer to write clean code easily. Therefore, a new language (which shall be named Keli) shall be introduced as a proposed solution in order to eliminate the difficulties of writing clean code.

1.4.1 Characteristics Of The Proposed Language

A. User can create understandable functions easily

User shall be able to create functions that resemble like natural English, thus minimizing the gulf of execution and evaluation (Hutchins, Hollan, and Norman, 1985). As a consequences, the code will be easier to understand, and easier to memorize.

B. User can create new functions without uncertainty.

User shall be able to create a new function without uncertainty, in other words, the proposed language shall only have a few construct which are obvious to the user for creating functions.

C. User can extends functionalities of existing classes easily

User shall be able to extend functionalities for even built-in data types without the need to use Design Patterns such as Decorator, Adapter or Bridge (Vlissides, Helm, Johnson and Gamma, 1995).

D. User can identify which functions involve I/O and which do not easily

User shall be able to quickly search for bugs as they can reason about the overall code more easily due to the distinction between I/O and non-I/O operations (Software Engineering StackExchange, 2015).

E. User will be prevented from creating impure functions that will access global variables or mutate inputs

The proposed language shall prevent user to create impure functions as impure functions introduces context and reduces the reasonability of the code as a whole (as mentioned in).

F. User will be prevented from passing incorrect type of inputs to functions

The proposed language shall prevent user from passing parameters of incorrect type to desired functions, and those error shall be detected at compile time rather than run time.

G. User will be prevented from wasting time on bugs that are due to implicit coercions

The proposed language shall prevent users to debug a program which is caused by implicit coercion. To achieve this feature, a strong type system must be existential for this language.

H. User can understand code without too much context

In other words, it means that this proposed language will prevent user from writing contextual functions, which are functions that will mutate its input or modify a global variable. Moreover, it shall also prevent user from using context-sensitive symbols as it reduces the reasonability of the code.

1.4.2 General Architecture

The general architecture of this project is illustrated in Figure 29. In a nutshell, the proposed language will go through a lot of process (which is the standard compiler technique) to be translated to another language, and the whole process will be boxed into a single application, which is commonly known as the compiler/ Further explanation shall be discussed in later chapters.

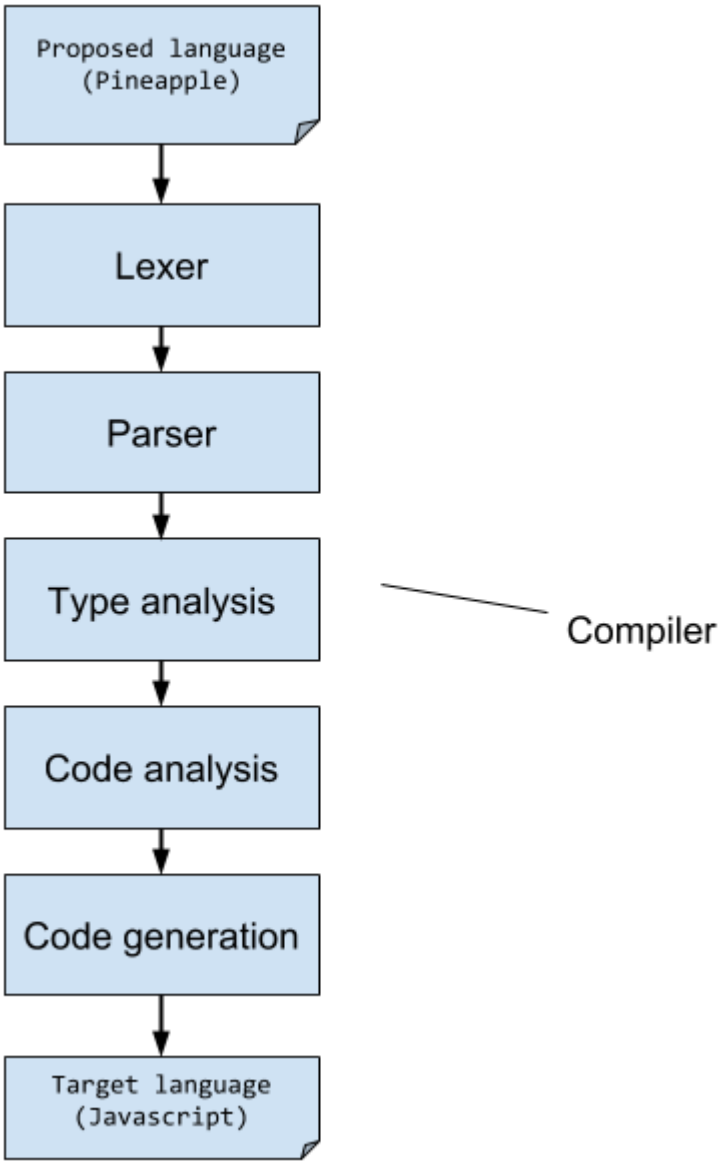

Figure 29: General architecture of this project

1.5 Proposed Approach

The development methodology that will be used is Test-Driven Development (TDD). This process is chosen because it suites the nature of this project, as Keli shall start as a small language and gradually adding more features and development progresses. According to Beck (2003), the TDD process can be summarized as three steps (also illustrated in Figure 30).

1. Red—write a little test that doesn't work, perhaps doesn't even compile at first.
2. Green—make the test work quickly, committing whatever sins necessary in the process.
3. Refactor—eliminate all the duplication created in just getting the test to work.

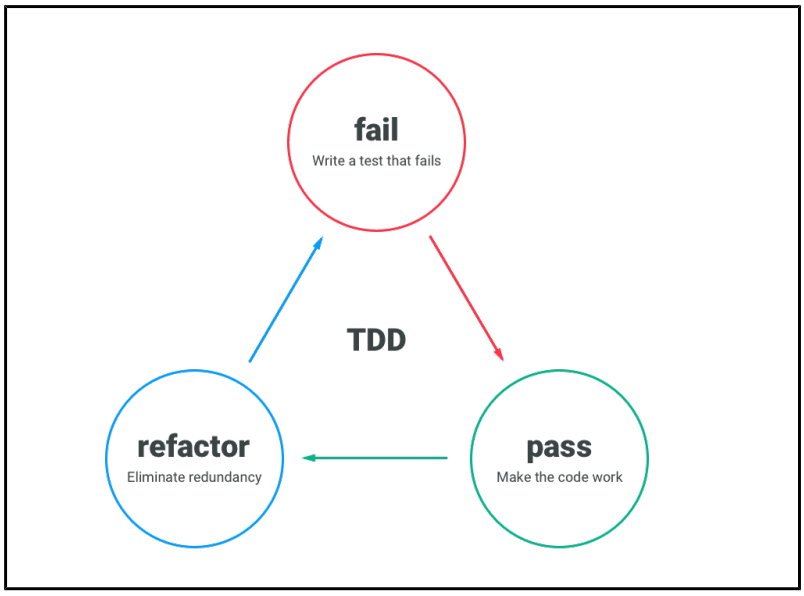

Figure 30: Test-driven development process (PromptWorks, 2018)

Further explanation about TDD shall be discussed in Section 3: Methodology.

1.6 Project Scope

1.6.1 Target Audience

1. Programmers who wish to write maintainable code. (The definition of *maintainability* is as defined in Section 1.2)
2. Non-programmers who wish to learn programming

1.6.2 Language Features

The proposed programming language, Keli shall include the following features:

A. Fundamentals Construct

This section is purposely labelled as 5.2.0 instead of 5.2.1 because the features that will be described here are fundamentals construct for any high level programming language:

1. Variables
2. Logical operations (not, and, or)
3. If-else statement
4. While loop
5. Foreach loop (similar as Python's)
6. Array literals
7. Object literals

Note that basic mathematical expressions are not included here as they are treated as function calls instead.

B. First-class prefix, infix, suffix and mixfix functions declaration

Keli shall allow user to declare prefix, infix, suffix and mixfix functions in a natural manner. In other words, a function in Keli shall be declared as how it is being invoked.

C. Function overloading using type signature

It means that a user can declare two or more functions that carries the same name, as long as there are differences in their argument(s) type signature.

D. Optional type annotation

It means that a variable or arguments of a function can be annotated with a type signature; in other words, it is acceptable if the type annotation is absent.

E. Strong type system

The type system of Keli shall be able to achieve the following features:

1. Type inference based on Hindley-Milner type system (Hindley, 1969; Milner, 1978).

2. Report type error.
3. Parametric polymorphism, which is also known as Generics in language like Java.
4. Enforce explicit nullability, means that a nullable variable shall be declared as so.

F. Generic structure declaration

It is just like struct in C. For example:

```
struct BinaryTree<T>
      .value T
      .left  BinaryTree<T>?
      .right BinaryTree<T>?
```

Figure 31: Example of generic structure

G. Parametric polymorphism / multiple dispatch

It means that a function call depends on the input type. For example, different function definition will be used between plus operator for two integer and plus operator for two list:

```
let x = 1 + 2
let y = [1 2 3] + [4 5 6]
```

Figure 32: Example of parametric polymorphism

H. Generic trait declaration

The term *trait* here means **interface** in Java or C#. For example:

```
trait Equatable<T>
      def this T (==) that T -> Boolean
            pass

      def this T (!=) that T -> Boolean
            return not this == that
```

Figure 33: Example of trait

I. Generic trait implementation

It also means the implementation of interface, which shall be separated from the structure definition. For example:

```
struct Person
      .name String
      .age  Int

implements Equatable<Person>
      def this Person (==) that Person -> Boolean
            return
                  this.name == that.name and
                  this.age  == that.age
```

Figure 34: Example of trait implementation

1.6.3 Deliverables

At the end of this project, the following items shall be delivered:

A. Web-Based Tutorial

A simple web-based tutorial that will teach new user on:

1. Motivation of Keli
2. Basic syntax and concept of Keli
3. How to install Keli
4. How to use Keli

B. Transpiler As A Haskell Binary

A transpiler that will translate the proposed language, Keli into JavaScript. The reason why JavaScript (or more precisely ECMAScript 6) is chosen is because it includes features such as Prototypes, which can simplify the code-generation of Keli. The transpiler shall provide readable error message (such as Python's) and strong type checking.

C. Basic Modules

The following basic modules shall be included in the first release of Keli:

1. Basic mathematical operators
    1.1. Addition
    1.2. Subtraction
    1.3. Multiplication
    1.4. Float/double division
    1.5. Integral division
    1.6. Modulus
    1.7. Exponent
2. Basic string/array operations
    2.1. Concatenation operator
    2.2. Append/prepend function
    2.3. Slicing function
    2.4. Range function
3. Assertion library (for writing unit test)
4. List library (to be implemented as inductive singly-linked list)

D. Executable Interpreter

An interpreter that can be downloaded as a single executable program which can interpret Keli source file and execute the code on the fly. In fact this will be just a simple program that embeds the Keli transpiler and the basic modules defined in Section 5.3.3.

E. Syntax Highlighter

A syntax highlighter, which shall be developed using Visual Studio Code (VSCode) extension model shall be included as part of the deliverables. Also, the syntax highlighter shall be hosted on VSCode official extension repository.

E. Visual Studio Code Extension

This extension shall incorporates Intellisense feature(a.k.a code completion suggestion) for Keli. It should be implemented by building a language serveer.

1.6.4 Non-features

Keli shall not support the following features at all (even in the future):

1. Object-orientation

Keli shall not support the following features at the moment:

1. Low level optimization

1.6.5 Non-deliverables

The following are non-deliverables of this project:

1. Integrated Development Environment (IDE)
2. Complete documentation of the features of Keli, as many features are subjected to changes over time
3. Compiler that will compile Keli to native code

CHAPTER 2

LITERATURE REVIEW

2.0 Introduction

In this section, the following topics shall be discussed. Firstly, brief history of programming languages and its evolution. Secondly, the paradigm of programming languages, followed by criteria of designing programming languages. After that, brief theory of compilers and its construction method will be elaborated. Lastly, existing programming languages that may solve the problem stated in Section 1.2 will be examined.

2.1 Brief History Of Programming Languages

According to Lambert and Louden (2011), programming language is "a notion for communicating to a computer about what we want it to do". Coding Tech (2018) mentioned that the one of the earliest programming languages, Mack was actually hand-compiled, i.e. operators would take the code written by an algorithm designer and translate it into the machine by manually flipping switches, as illustrated in Figure 35.

Figure 35: An operator hand compiling a program (Coding Tech, 2018)

It is notable that operators (aka. coders) and algorithm designer are different back then, however this two entities had been combined into a single entity which is known as *programmers* nowadays ever since the born of high level language.

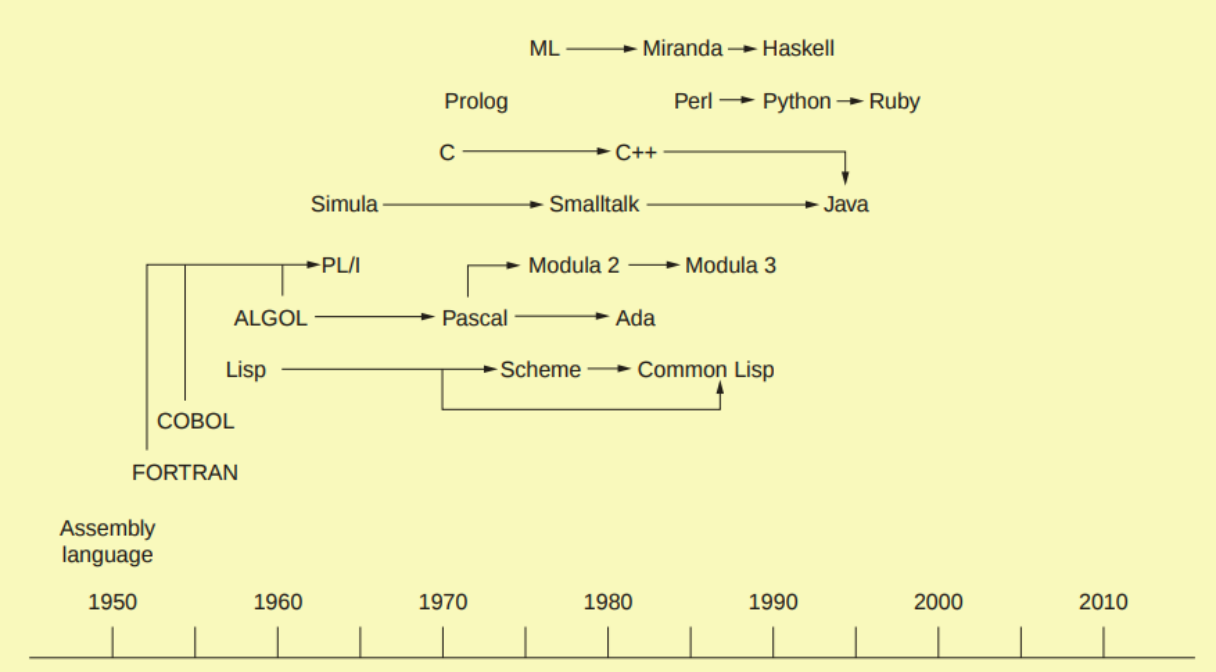

Figure 36: A programming language timeline (Lambert and Louden, 2011)

2.1.1 Machine Language

Schneider and Gersting(2018) expressed that computer design undergone huge advancement in the late 1940s when John von Neumann that computers should have a small set of permanently hardwired general-purpose operations, which give rise to the famous John von Neumann computer architecture, which is still used in modern days. Thus, operator could insert code into a machine by flipping switches (as in Figure 36) and the notion of flipping switches is what known as *machine language* (Lambert and Louden, 2011). For example, 011011 might means move 12 to register A.

```
01001000 01100101 01101100 01101100 01101111
00100000 01010111 01101111 01110010 01101100
01100100
```

Figure 37: Sample of machine language (Computer Hope, 2017)

2.1.2 Assembly Language

Obviously, coding in machine language is too tedious and error-prone as one have to refer to code manuals to type out desired code. Thus, early programmers developed assembly language to use mnemonic symbols to represent instruction codes and memory locations (Lambert and Louden, 2011). Nonetheless, assembly language is semantically equivalent to machine language as most assembly instructions maps directly to a correspondent machine code.

```
LDR R1, X      ; Copy number from memory location X to register R1
LDR R2, Y      ; Copy number from memory location Y to register R2
ADD R3, R2, R1 ; Add the numbers in R1 and R2 and place in register R3
```

Figure 38: Adding two numbers in assembly

Despite the increased usability, assembly language still lacks the power to express abstraction. For example, one cannot easily express the Mathematical notation $x + y$ in assembly language, as depicted in Figure 38. The ability to express abstract notation is significant as the mathematician A.N. Whitehead once said:

"*By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems ...* " (Schoenfeld, 2013).

2.1.3 High Level Language

To realize the importance of expressing abstraction, languages such as FORTRAN is introduced, which allowed programmer to directly use algebraic notation such as addition of 2 numbers or variables (Lambert and Louden, 2011). Then, ALGOL (acronym for ALGOrithmic Language) introduces block structures such as if block, while block and switch block, which allow programmers to express algorithm in a more structured way (Coding Tech, 2018). Since then, a lot of new languages such as C, Pascal and Ada descended from the ALGOL (Lambert and Louden, 2011).

```
function add(int x, int y) {
      return x + y;
}
```

Figure 39: Example of high-level language code

2.2 Paradigms Of Programming Languages

2.2.1 Functional Programming (FP)

A. High-Order Functions

One of the characteristics of functional programming is the support of high-order functions. According to Elliott (2017), a higher order function is a function that takes a function as an argument, or returns a function.

```
function map(f, xs) {
      let result = []
      for(var i = 0; i < xs.length; i++) {
            result.push(f(xs));
      }
      return result;
}

// Usage
let doubles = map(x => x * 2, [1,2,3,4]); // [2,4,6,8]
```

Figure 40: Example of function that takes function as argument in JavaScript

In Figure 40, the map function is the high-order functions, as it takes `f` which is a function as input. Moreover, the usage of anonymous functions is also illustrated in the figure, as seen in `x => x * 2`, which is equivalent to a function that takes an input and returns the double of it. Another characteristics of high-order functions is that they can be composed of each other, as mentioned by Lambert and Louden (2012). For example *(g o f) (x)* means *g (f(x))*.

B. Referential Transparency

According to HaskellWiki (2009), referential transparency and be understood as a phenomenon where each expression will be evaluated equally despite the context. For example, a function is considered referential transparent if it always give the same output when the same input is given.

```
// A referential transparent function
function double(x) { return x * 2; }

// A non-referential transparent function
function getCurrentTime() { return new Date(); }
```

Figure 41: Simple illustration of referential transparency

Although referential transparency can be achieved in imperative languages, they do not exhibit true referential transparency, because Strachey (2000) states that the construct of variable assignment destroys referential transparency. One example is for-loop, as its iterator variable (usually named as `i`) has different value in different point of time.

```
for(int i = 0; i < 10; i++) {
      print(i);
}
```

Figure 42: Example of for-loop

Thus, pure functional languages like Haskell does not support variable assignment at all. One benefit of referential transparency is that one can always reason about a parts of code easily. For example, the double function in Figure 41 can be understood clearly without knowing what is happening outside of it.

C. Declarative

Another characteristics of FP are declarative-ness, it means that in FP, one tells the computer *what to do*, contrast to imperative programming where one tells the computer *how to do*. That is to say, there are no execution sequence in FP, i.e. no loops at all. Thus, the only way to emulate looping is through recursion.

```
// imperative version
function factorial(n) {
      let result = 1;
      for(var i = 1; i <= n; i++)
            result = result * i
      return result
}

// declarative version
function factorial(n) {
      if(n <= 1) return 1;
      else return n * factorial(n - 1);
}
```

Figure 43: Difference between imperative and declarative

In fact, Mathematical notation always preferred the declarative versions to denote formulas, for example the Fibonacci function as seen in Figure 44.

$$F_n = \sum_{k=n-2}^{n-1} F_k$$

Figure 44: Mathematical notation for Fibonacci sequence (Math StackExchange, 2013)

However, it is also possible to emulate imperative-ness in a purely declarative manner, one example is the usage of monads in Haskell. The term monads can also be understood as callback, however callback is not necessarily a monad as argued by Shigh (2016).

Basically, monads are chain of operations according to HaskellWiki (2017), as illustrated in Figure 45. The code in the figure can be understood as, evaluate getline, when it is evaluated successfully, evaluate putStrLn. Note the return in Haskell is not the same as usual imperative programming, it is just an identity function needed to complete the evaluation of monads (HaskellWiki, 2017).

```
main = getLine                              >>= \x ->
       putStrLn ("Your input is " ++ x) >>= \z ->
       return z
```

Figure 45: Example of monads

D. Algebraic Data Type (ADT)

Another important aspect of FP is the first-class support for Algebraic data type (note that this is different from Abstract Data Type). One advantage of ADT is that it allows programmer to concisely define the model of a domain. For example, assume one want to create a model to count the value of card for a simplified Blackjack game. In this version of Blackjack, all pictures (King, Queen, Jack) have value of 10, Ace have value of 1, and other number cards have value based on their number. By using a procedural approach, one might declare a structure as in Figure 46.

```
struct Card {
      type: CardType;
      number: Integer;
}

enum CardType = { Ace, Picture, Number }
```

Figure 46: Modelling blackjack using procedural approach

Although that structure definition is appropriate, it is not entirely safe, because the property number is not necessary when the card type is Ace or Picture. Thus, one may make some mistake as shown in Figure 47, as one might accidentally access the property which should not be used, for example, when the card type is Picture, the programmer should not access the number property at all, however such restriction cannot be really specified by just using

structure type.

```
function valueOf(card: Card): Integer {
      switch(card.type) {
            case Number:
                  return card.number;
            case Ace:
                  return 1;
            case Picture:
                  return 10 + card.number;
                  // ^Mistake, but the compiler cannot tell
}
```

Figure 47: Flaws of using structure type to model the blackjack game

The actual problem with using structure type is because the model is actually disjoint union, it means that different kind of data have different kind of property, but all those kind belongs to the same type. In this case, when the kind of card is Ace or Picture, the data have no property at all; however when the kind of Card is Number, the data have one property which is number.

This problem described just now can be easily solved with ADT, as demonstrated in Figure 48, i.e., when the card type is Ace or Picture, it does not have the property number, thus it is impossible to access the number property no matter how, one can only access it when the card type is Number.

```
data Card = Ace
          | Picture
          | Number Integer

valueOf :: Card -> Integer
valueOf card = case card of
                Ace      -> 1
                Picture  -> 10
                Number x -> x
```

Figure 48: Example of ADT in Haskell

Although ADT did solve the disjoint union problem nicely, the current conventional syntax for describing ADT in functional languages such as Haskell, ML and OCaml still suffers the argument-position problem as described in . For instance, in Figure 49, the definition of binary tree does not tell which node is left or right, thus it might be a cognitive burden if the programmer wish to implement a left-balanced tree or right-balanced tree, as he/she needs to memorize that the first Tree is the left node (or the right node).

```
data Tree = Empty
          | Leaf Int
          | Node Tree Tree // Is the first Tree at left or right?
```

Figure 49: Ambiguous argument position in ADT

Also, it should be noted that OOP can also solve the disjoint union above precisely, however it is definitely harder to define and harder to understand compare to that of ADT (see Figure 50).

```
interface Card {
      int valueOf();
}

class Ace implements Card {
      public int valueOf() { return 1; }
}

class PictureCard implements Card {
      public int valueOf() { return 10; }
}

class NumberCard implements Card {
      private int value;
      public NumberCard(int value) { this.value = value; }
      public int valueOf() { return this.value; }
}
```

Figure 50: Modelling blackjack game using OOP

Moreover, it should be noted that imperative languages that supports discriminated unions can also describe the blackjack model precisely. Consider the Typescript code in Figure 51.

```
  interface AceCard {
      kind: "Ace";
}

interface PictureCard {
      kind: "Picture";
}

interface NumberCard {
      kind: "Number";
      value: integer;
}

type Card = AceCard | PictureCard | NumberCard;

function valueOf(card: Card): integer {
      switch(card.kind) {
            case "Ace":     return 1;
            case "Picture": return 10;
            case "Number":  return card.value;
}
```

Figure 51: Modelling blackjack using discriminated union with struct

In a nutshell, the notation used in FP to describe ADT is terser than its counterparts, thus it is arguable that FP might be easier to understand as one do not need to bother too much about unnecessary details. This situation can also be compared to musical notation. For beginners, the music score in Figure 52 might be unreasonable and hard to read, however for one who is proficient, he/she can definitely interpret faster than compare to the textual version in Figure 53.

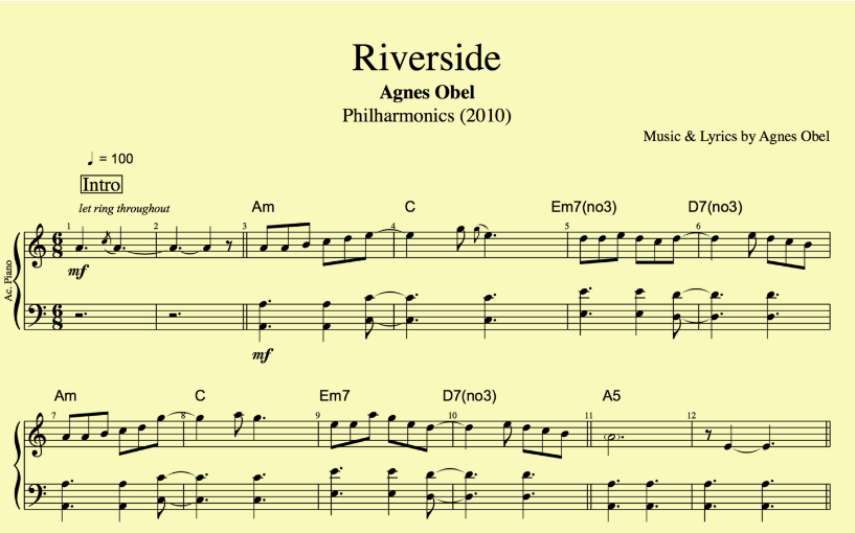

Figure 52: Standard musical notation

```
Treble: A5(6) A5(½) A5(½) B5(½) C6(½) …
Bass: REST(6) A3(1½)+A4(1½) A3(1½)+A4(1½) …
```

Figure 53: Textual musical notation

2.2.2 Logic Programming (LP)

Another programming language paradigm that is declarative is logic programming. LP language is based on first-order predicate calculus (Lambert and Louden, 2012). In short, the main construct of LP is facts (aka. clauses) and queries.

```
// Facts
1) All human are immortals.
2) John is a human.

// Queries
1) Is John immortal?
```

Figure 54: Main constructs of logic programming

The English statements in Figure 54 can be expressed in Prolog (one of the most widely used LP language), as illustrated in Figure 55, and when the code is executed, the Prolog interpreter will return either yes or no for each queries. In this case, the interpreter shall return yes.

```
 immortal(x) :- forall(human(x)).
human(John).

?- immortal(John)
```

Figure 55: Example of Prolog code

2.2.3 Object-Oriented Programming

Since, OOP is widely known due to its heavy usage in software engineering industry and its deep penetration software-related educational courses, the fundamentals of OOP shall not be discussed in this section, however, this section will aim to discuss the drawbacks of OOP, and how features from other language paradigm can solve those problems without those features in OOP but still be able to retain the benefits of OOP.

Before the discussion, one should understand that the main importance of OOP is to satisfy the three significant needs in software design: high reusability, high independence and minimal modification (Lambert and Louden, 2012). Besides that, one should know that there are four pillars in OOP, namely Inheritance, Abstraction, Encapsulation and Polymorphism.

A. Problem With Inheritance

One of the first problem is inheritance, even the computer scientist, Alan Kay who coined the term *object-oriented programming* does not like the concept of inheritance (Kay, 1998). Furthermore, the holy book of OOP, Design Patterns even recommend programmers to favor composition over inheritance (Vlissides, et. al, 1995). One issue of inheritance is that it violates encapsulation. To demonstrate, see Figure 56. In this case, a stack overflow exception will be thrown when the Child object invoked the bar method, due to the recursion of bar calling foo and foo calling bar. However, the programmer who created the Child class would not be able to figure out the problem, unless he/she look at the definition of Parent. Thus inheritance violates encapsulation, because a programmer who wish to extends a class from a parent class need to understand the parent class internal behavior.

```
public class Test {
  public class Parent {
    public void foo() {
      bar();
    }

    public void bar() {}
  }

  public class Child extends Parent {
    @Override
    public void bar() {
      foo();
    }
  }

  public static void main(String[] args) {
    new Child().bar(); // StackOverflowException
  }
}
```

Figure 56: Inheritance breaking encapsulation

B. Problem With Encapsulation Of OOP

Another problem is how OOP handle encapsulation. By definition, encapsulation is to hide unnecessary details from the client by exposing necessary interfaces. But OOP even violates encapsulation at the lowest level by allowing member variables to be manipulated by methods. Consider the code in Figure 57, in this case, the add method manipulates the member variables elements and index, and this construct actually violates encapsulation, because to understand the behavior add, one have to look at variables that is located outside of it, which means that this situation is no different than using global variables! The code will be even less understandable if there are more methods in the class, as all of them can possibly manipulates the member variable. This is the main reason why programmers are ban from using global variables in the first place, because it causes code to be context-sensitive. But OOP encourage programmers to use global variables (within a class). Thus, I suggest that OOP is actually a glorified way to handle a global variables, even the creator

of Smalltalk, Alan Kay mentioned the so called "*object-oriented programming*" is simply old style programming with fancier construct. Although it is true that global variables (a.k.a. registers) are crucial in low level language like assembly, this idea should not be brought up to high-level languages due to its known disadvantages.

```
class MyArrayList {
     private Object[] elements;
     private int index = 0;

     public void add(Object newItem) {
          this.elements[index++] = newItem;
     }
}
```

Figure 57: Encapsulation violated by member methods

The code in Figure 57 can be understood better if written in a procedural manner, as shown in Figure 58, as one do not have to worry about global variables that will be potentially manipulated by other methods. To overcome this shortcoming of OOP, language such as C++ provides construct such as const method to prevent the manipulation of instance variable, also it is a recommended to mark a method as const whenever possible, in order to prevent accidental changes, as shown in (Meyers, 2005). Thus, in my opinion, I thought that the concept of *private instance variable that can be mutated by any instance method* is flawed, to the extent that C++ even provides workaround to tackle its drawback.

```
void add(Object[] array, Object item) {
   // function body
}
```

Figure 58: Procedural version of the add function

```
class Car {
      private:
            double speed;

      public:
            void showSpeed() const {
                  this->speed = 23; // Compile error
            }
}
```

Figure 59: Example of const method in C++

C. Getter And Setter Prevent Reusability

According to StackOverflow (2010), one of the important uses getters and setters is to allow programmer to provide logic around retrieving or writing data, e.g., validation. However, such usage can actually prevent reusability. To demonstrate, consider the Person class presented in Figure 60.

```
class Employee {
     private String name;
      public String getName() {
            return this.name;
      }
     public void setName(String newName) {
          if(newName.isValidName(newName)) {
               this.name = newName;
          } else {
               throw new Exception("Invalid name.");
     }
     private boolean isValidName(newName) {
          // validation logic
     }
}
```

Figure 60: A simple class with setter

Suppose a new class Product is to be created, and coincidentally the validation logic for its name is same as the validation logic for name of Person. Here comes the question, should Product class inherit the Person class or have an instance variable of Person (see Figure 61)? Obviously, both ways are not good as they are semantically illogical, because based on common sense, Product should not be a subclass of Person, and meanwhile, Product should not be composed by Person (unless requirements states that Person is logically related to Product, e.g. ownership). Thus, this proves that setters may prevent reusability (which is one of the main objective of OOP) in certain situation. This problem can be overcome by using function composition and pure data structures, as shown in Figure 62.

```
// should Product inherit Person?
class Product extends Person {
      // this sounds semantically awful even though it works
}

// should Product compose Person?
class Product {
      private Person person;
      public void setName(String newName) {
            this.person.setName(newName);
      }
      public String getName() {
            return this.person.getName();
      }
 }
```

Figure 61: Getter and setter promote reusability that might break semantic consistency

```
struct Product { String name;}

struct Person { String name; }

boolean isValidName(String name) { /* validation logic */ }

void setName(Person|Product p, String newName) {
      if(isValidName(newName)) {
            p.name = newName;
      }
}
```

Figure 62: Function composition can prevent the setter problem

Due to the reason above, the proposed language shall not fully incorporate object-oriented features.

2.3 Criteria Of Designing Programming Language

According to Lambert and Louden (2011), the early machines were extremely slow with limited memory, thus the main criteria for designing a language in the older days are primarily efficiency, and also writability as programs are not meant to be read or maintain by other programmers. They also mentioned that as programs got bigger and bigger, readability and maintainability becomes an issue, thus languages such as COBOL emerges. Since then, the criteria of designing programming languages expanded, which will be discussed in the following sections.

2.3.1 Regularity

Regularity can also be defined as consistency, i.e. a programming language is said to be regular if it adheres to the principle of least astonishment (Lambert and Louden, 2011). To further explain regularity, they subdivided it into three concepts, namely generality, orthogonality and uniformity.

A. Generality

They state that a language with generality shall avoid special cases whenever possible. A common example of violation of generality is the == operator in C, as it cannot be used to compare two arrays or two structures. This restriction no longer exist in languages such as C++ and Haskell as operator overloading is permitted, thus such languages are said to have achieved higher generality (Lambert and Louden, 2011).

B. Orthogonality

According to Lambert and Louden (2011), a language is orthogonal if its constructs do not have different behaviors in different context, thus restrictions that depends on context is viewed as non-orthogonal. They states that one example of lack of orthogonality is shown in C and C++, because a function can return any data type but not array type. Fortunately, such lack of orthogonality is removed in modern languages such as Haskell, Python and JavaScript.

C. Uniformity

Uniformity also means the consistency of appearance and behavior of language constructs, as defined by Lambert and Louden (2011). One example of violation of uniformity is the

context-sensitive symbols in JavaScript, as discussed in .

2.3.2 Extensibility

A language which is extensible allow programmers to add new features to it. Lambert and Louden (2011) states that most languages have certain degrees of extensibility, i.e. to allow programmers to define new data type (such as classes and structures), new functions and even new modules. They further mentioned that some languages like LISP, C and C++ even allow programmer to add new syntax and semantics by using macros.

```
#define IF   if (
#define THEN ) {
#define ELSE } else {
#define ELIF } else if (
#define FI   ; }

int x = 5
IF x > 5 THEN
     printf("xx is more than 5");
ELIF x < 5 THEN
     printf("x is less than 5");
ELSE
     printf("x is equals 5");
FI
```

Figure 63: Using C macros to emulate ALGOL-68 syntax (StackOverflow, 2017)

2.4 Existing Programming Language Features

In this section, existing programming languages features that might solve the problem statements (as per) shall be discuss. Furthermore, programming languages that have similar objectives as the proposed language (which is Keli) shall be explored as well.

2.4.1 First-Class Mixfix Functions

One feature that might resolve the problem is mixfix function. In this subsection, programming languages that support or resembles mixfix functions (as define in) will be discussed. Moreover a deliberation about emulation of mixfix function will be examined as well.

A. Smalltalk

Smalltalk is one of the earliest language that supports mixfix functions. As a matter of fact, a recent research actually claims that Smalltalk is one of the most productive programming languages (Jones, 2017). I personally believe it is due to its first-class support for mixfix-ability. The following is an example of grammatically valid Smalltalk code:

```
myNumber isBetween: 1 and: 10
```

In this case, the function name is `isBetween:and:` and the arguments are `myNumber`, `1`, `10`. Note that the function name portions can be increased indefinitely, i.e. the number of function name portions is equals to the number of arguments minus 1. Thus, if the function is to accept 5 arguments, it should have 4 separated partial name.

However, Smalltalk does not support prefix function, because every function invocation must be started with an object, in the example above, the variable, `myNumber` is treated as the object. One possible drawbacks of such limitation is that some statement might seems incomprehensible to some readers, for example:

```
myVariable := true not
```

In this case, the literal `true` is the object, while not is the function being invoked. In such situation, the Smalltalk grammar directly contradicts with the conventional English, because people would usually say *not true* instead of *true not*. Nevertheless, the code snippet above actually resembles *inversion* or *anastrophe*, which is a form of literary style, for example, an inverted English sentence might look like (Encyclopedia Britannica, 2018):

He is trustworthy not.

Despite the fact that such grammar is unorthodox in English, it is actually natural in SOV (Subject-Object-Verb) language such as Japanese and Korean, where verb should appear last in each statement (The Atlantic, 2015).

One possible reason that Smalltalk enforce such limitation is that the overall consistency of the language can be preserved, if not the grammar will become context-sensitive, for example (note that comments are enclosed within double-quotes):

```
"Statement 1"
sort myList

"Statement 2"
myList sum
```

In Statement 1, the first token is the function, however in Statement 2 the first token is the object. In such case, it is impossible for a context-free grammar to differentiate whether the first token of each statement is an object identifier or a function identifier as the first token and second token are *lexically equal*.

Another reason that such rule is dictated is that function chaining (aka. message chaining in Smalltalk aka. method chaining in Java) can be done more easily. For example, one can apply multiple operation to a list in the following manner:

```
result := myList sort tail sum
```

A similar code in prefix-oriented language would be:

```
result = sum(tail(sort(myList)))
```

One obvious benefit of function chaining is that it resembles much more closer to English, as English sentence are read from left to right, unlike the second example where reader are forced to read from right to left. Certainly, there are some workaround for prefix-oriented functions to emulate function chaining, for instance in F#, one could use the pipe-forward operator (`|>`) as such:

```
let result = myList |> sort |> tail |> sum
```

B. Agda

Besides Smalltalk, another programming language that supports mixfix function out of the box is Agda (a direct descendant of Haskell). Comparing Smalltalk, Agda mixfix support is much more sophisticated as its compiler can parse prefixed-mixfix function and postfixed-mixfix function such as the following (Danielsson and Norell, 2008).

```
_[_]

if_then_else

_?_:_

[[_]]
```

Figure 64: Example of mixfix functions in Agda

Note that underscore means arguments position, thus in the first line, the function names is the open bracket ([) and the close bracket (]). Example of declaring mixfix function in Agda is depicted in Figure 65.

```
_and_ : Bool → Bool → Bool
true and x = x
false and _ = false
```

Figure 65: Declaring mixfix function in Agda

Note that the function body is done using pattern matching instead of list of statements, which is usually seen in imperative languages. On top of that, Agda even supports user-defined associativity and precedence level, for example:

```
infixr 20 _and_
```

The code above means that the function _and_ will have infix precedence of 20 and is right associated (because of the **r** at the back of **infix** keyword).

2.4.2 Emulation Of Mixfix Function

Despite the fact that most programming languages does not support mixfix function, one can actually emulate it using some other language features which shall be discussed later.

A. Object-chaining

In OOP language, one could emulate mixfix using object-chaining. One of the great example is the famous JavaScript TDD - assertion framework: Chai.js

Example code snippet of Chai.js (Chai, 2017):

```
expect(beverages).to.have.property('tea').with.lengthOf(3);
```

The example above demonstrated that this library is actually trying to emulate mixfix function.

However, there is one problem with it, because to achieve such API, the author have to abuse *object-chaining*, which is actually a cognitive burden. The code snippet in Figure 66 (workable in Python 3.5) shall demonstrate how one could emulate the mixfix-style replace function using object-chaining.

```
class Replace:
      # Constructor
      def __init__(self, oldChar):
            self.oldChar = oldChar

      def _with(self, newChar):
            return ReplaceWithChain(self.oldChar, newChar)

class ReplaceWithChain:
      def __init__(self, oldChar, newChar):
            self.oldChar = oldChar
            self.newChar = newChar

      def _in(self, targetString):
            return targetString.replace(self.oldChar, self.newChar)
```

Figure 66: Using object-chaining to emulate mixfix function in Python

Then, to use the library:

```
result = Replace(",")._with(".")._in("hello,world")
```

Figure 67: Usage of the emulated mixfix function

Although it is doable, it is clear that it is mental burden to emulate mixfix function, because *two* different classes are needed to emulate *one* function. Thus, it is rare to see library that will try to emulate mixfix, except for the aforementioned Chai.js library.

B. Named Parameters

Besides object-chaining, one could emulate mixfix functions using named parameters. Programming language that support named parameters are Swift, Python, C# etc. For example in Swift, the **replace** function can be declared as shown in Figure 68.

```
// Declaration
func replace(old:String, with new:String, in target:String) -> String {
      // function body
}

// Invocation
replace("," , with: "." , in: "Hello world")
```

Figure 68: Named parameters in Swift

Although emulating mixfix-ality using named parameters is much easier to construct than object-chaining, it has several drawbacks. First, it requires more typing due to the need to type argument separator (which is usually comma), causing the overall readability to be lower than that of Smalltalk or Agda that support true mixfix functions. Secondly, this method cannot entirely fulfil the definition of mixfix function, because it cannot emulate infix function. Note that mixfix function should include infix function as well (as defined in 3.1.1).

C. Object Destructuring

Another method to emulate mixfix function is through object destructuring. One

programming language that support such feature is JavaScript (since ES6). Destructuring is a convenient way of extracting multiple values from data stored in (possibly nested) objects.

```
const name = { first: 'Jane', last: 'Doe' };
const {first: x, last: y} = name;
alert(x); // 'Jane'
alert(y); // 'Doe'
```

Figure 69: Object destructuring in JavaScript

In the second line, two variables are created, namely x and y; then, the value of name.first is stored into x and name.last is stored into y respectively, this situation is the so-called object destructuring. By using such feature, one could emulate the replace function (see Figure 70).

```
// declaration
function replace(oldStr, {with: newStr, in: input}) {
     return input.replace(oldStr, newStr);
}

// invocation
replace("," , {with: ".", in: "Hello, world"});
```

Figure 70: Emulating mixfix function using object destructuring in JavaScript

Similarly, using object destructuring to emulate mixfix function is not as hard as object-chaining, however it is noticeable that the invocation interface is not consistent, as the first argument does not require any curly braces but the second and third arguments require curly braces.

2.4.3 Summary Regarding Mixfix

Through the review above, it is apparent that none of the techniques to emulate mixfix functions are not as clean and clear as true mixfix functions as in Smalltalk or Agda, thus Keli should support true mixfix function and make it a first-class feature in order to tackle

the problem of difficulties in creating understandable function.

2.4.4 Consistent Function Declaration

A programming language is said to have consistent function declaration if it satisfy the following criteria (as opposed to):

1. Only supports one of the following:
    a. Static method
    b. Instance method
    c. Free functions
2. Does not support mutable function inputs
3. Does not support global variables
4. Does not have the problem of arguments position

The only programming language that supports consistent function declaration is possibly Agda, as it only support free functions and fulfil the rest of the criteria above.

2.4.5 Extension For Existing Classes Without Inheritance

To resolve the issue of difficulties in extending existing classes, a programming language must provide a method for programmer to add more functionalities to a class without modifying the definition of the class or extending it.

A. Haskell

One language that support such feature is Haskell. To demonstrate, suppose there is a record definition (aka. struct in C) for the Person entity (see Figure 71).

```
data Person = Person {
      name   :: String ,
      salary :: Double
}
```

Figure 71: Declaration of Person entity in Haskell

Then, suppose there is a class (aka. interface in Java) named Comparable (see Figure 72).

```
class Comparable a where
      equals    :: a -> a -> Bool
      moreThan  :: a -> a -> Bool
```

Figure 72: Comparable class in Haskell

In order to let the Person record to implement the Comparable interface, the code should be written as in Figure 73.

```
instance Comparable Person where
      equals   x y = (name x)   == (name y)
      moreThan x y = (salary x) >  (salary y)
```

Figure 73: Implementing Comparable class in Haskell

This is where the difference comes in, because the interface implementation is totally separated from the record definition. To contrast, look at the Java code in Figure 74 which bears similar semantics, notice that the interface implementation is within the class definition:

```
class Person implements Comparable<Person> {
    public string name;
    public double salary;

    @override
    public bool equals(Person that) {
         return this.name == that.name;
    }

    @override
    public bool moreThan(Person that) {
         return this.salary > that.salary;
    }
}
```

Figure 74: Implementing Comparable interface in Java

A direct advantage of Haskell's extension system is that one can even let primitive types such as Integer or String to implement a custom-defined interface (aka. class in Haskell). However, it is apparent that Haskell uses keywords that may confused programmers from object-oriented background, as shown in Table 2.

Haskell	Java/C#
data	*class*
class	*interface*
instance	*implements*

Table 2: Comparison of keywords between Haskell and Java/C#

These keywords confusion may be one of the reason why Haskell's learning curve is very steep. Therefore, Keli shall not adopt to keyword that will confuse object-oriented programmers.

Despite the fact that such feature of Haskell allows programmer to extend functionalities of existing data types easily, another advantages which might not seems obvious is that the problem of *case discrimination* which is seemingly hard in OOP can be resolved easily. For example, suppose there is a Shape interface, Triangle class and Circle class (as in Figure 75).

```
public interface Shape {
            public boolean intersect(Shape that);
}

public class Triangle implements Shape {
      public Coordinate v1, v2, v3; // three vertices
      public boolean intersect(Shape that) {
            // Definition
      }
}

public class Circle implements Shape {
      public Coordinate center;
      public double radius;
      public boolean intersect(Shape that) {
            // Definition
      }
}
```

Figure 75: Shapes class in Java

The main problem here is that the intersection algorithm for Triangle-Circle, Circle-Circle and Triangle-Triangle are totally different, so it is almost impossible to define the definition of intersect for each class that implements Shape without using switch-case statement. If switch-case statement is being used, then the whole purpose of polymorphism is defiled, consequently causing the violation of Open-Closed Principle when more Shape class is added in the future. One possible solution that will not violates true polymorphism is using multiple implementation of the same generic interface, as shown in Figure 76.

```
public interface Shape<T extends Shape>{
      public boolean intersect(T that);
}

public class Triangle implements Shape<Triangle>, Shape<Circle> {
      public Coordinate v1, v2, v3; // three vertices
      public boolean intersect(Triangle that) { // Definition }
      public boolean intersect(Circle that)   { // Definition }
}

public class Circle implements Shape<Circle>, Shape<Triangle> {
      public Coordinate center;
      public double radius;
      public boolean intersect(Circle that)   { // Definition }
      public boolean intersect(Triangle that) { // Definition }
}
```

Figure 76: Multiple implementation of the same generic interface with different type parameters

However, at the time of writing, such construct as shown above is not supported in Java (as of JDK 10.0.1), because the compiler reject classes that implements the same interface with different type parameters.

To understand how Haskell solve the *case discrimination* problem, one shall read the paper written by Lämmel and Ostermann (2006), as it is difficult to summarize in this paper.

B. JavaScript

Besides Haskell, another language that supports implementation separation is JavaScript, because of its Prototype model. The code snippet in Figure 77 shall demonstrate how the previous Person class example could be implemented using Prototypes.

```
function Person(name, salary) {
     this.name = name;
     this.salary = salary;
}

Person.prototype.equals = function(that) {
     return this.name == that.name;
}

Person.prototype.moreThan = function(that) {
     return this.salary > that.salary;
}
```

Figure 77: Example of Prototype programming in JavaScript

However, such feature is usually not recommended by most JavaScript developers, as one can accidentally override built-in functions, as JavaScript would not warn for repeated definitions (see Figure 78)

```
// Class declaration
function Person() { /*Empty body*/ }

// Initial declaration of sayHi
Person.prototype.sayHi = function () {
     return "hi";
}

// Accidental re-declaration which is faulty
Person.prototype.sayHi = function () {
     return "bye";
}
```

```
    let john = new Person();
    john.sayHi(); // Result is "bye" instead of "hi"
```

Figure 78: JavaScript's Prototype programming is dangerous as it might override existing functions

The code in Figure 78 will be interpreted without any warnings or error at all, this is because JavaScript is meant to be tolerant, that is to say, although separation of implementation can be achieved, it is not totally safe to be done in JavaScript. Therefore, the proposed language shall have built-in mechanism to prevent implicit function overwrite.

C. C#

Another language that allows methods implementation to be separated from class definition is C#. Consider the example in Figure 79 that add a WordCount method to the built-in String type.

```
public static class MyExtensions {
    public static int WordCount(this String str) {
          return str.Split(new char[] { ' ', '.', '?' },
            StringSplitOptions.RemoveEmptyEntries).Length;
    }
}

public class Program {
    public static void Main() {
          String s = "Hello extension methods";
          int count = s.WordCount(); // Calling the extension method
    }
}
```

Figure 79: Example of C# extension methods (Microsoft, 2015).

Although C# allow extension, it does not allow interface to be implemented via such construct. As a consequences, reusing of generic functions is not achievable for compiled class. For example, the code in Figure 80 will work but the code in Figure 81 will not work.

```
class Program {
      static void Main(string[] args) {
            var people = new List<Person>() {
                  new Person {name= "John",age= 12 },
                  new Person {name= "Lee", age= 31 },
            };
            people.Sort(); // This line works
      }
}

class Person : IComparable<Person> {
      public string name;
      public int age;
      public int CompareTo(Person that) {
            if(this.name == that.name)   return  0;
            else if(this.age < that.age) return -1;
            else                         return  1;
      }
}
```

Figure 80: Invoking extension method in C#

However, if one uses extension method to implement the CompareTo method, the invocation of Sort will result in error:

```
class Program {
      static void Main(string[] args) {
            var people = new List<Person>() {
                  new Person {name= "John",age= 12 },
                  new Person {name= "Lee", age= 31 },
            };
            people.Sort(); // Error: InvalidOperationException
      }
}

class Person {
      public string name;
      public int age;
}

static class PersonExtension {
      public static int CompareTo(this Person x, Person y) {
            if(x.name == y.name)   return  0;
```

```
            else if(x.age < y.age) return -1;
            else                   return  1;
        }
    }
```

Figure 81: Limitation of C# extension methods

Thus, C# does not truly solve the problem separation of interface implementation, because it does not allow generic functions to be reuse when functions are extended to a class via this extension method.

D. Elixir

Elixir allows user to achieve extensibility via protocols instead of inheritance. For example, if one wish to create have a unified method count for some common data structures, it could be done as shown in Figure 82.

```
defprotocol Counter do def count(data) end

defimpl Counter, for: BitString do
  def count(string), do: byte_size(string)
end

defimpl Counter, for: Map do
  def count(map), do: map_size(map)
end

defimpl Counter, for: Tuple do
  def count(tuple), do: tuple_size(tuple)
end
```

Figure 82: Declaration and implementation of interface in Elixir

Example of using the count method is shown in Figure 83. Note that iex> is the prompt of the Elixir REPL.

```
iex> Counter.count("foo")
3
```

```
iex> Counter.count({:ok, "hello"})
2
iex> Counter.count(%{label: "some label"})
1
```

Figure 83: Invocating interface method in Elixir

One drawback of such construct is that the method invocation syntax is a little bloated compared to Haskell or JavaScript, as one have to specify the protocol(a.k.a. interface) name (in this case it is Counter), this is more or like similar to static methods in OOP languages.

2.4.6 Distinction Between I/O And Non-I/O Operation

One language that supports distinction between I/O and non-I/O operation is Haskell. Haskell achieve this by using the I/O Monad. In layman terms, every function that involve I/O operations must be tainted with the IO keyword.

```
readln :: Read a => IO a
```

Figure 84: Signature of readln function in Haskell

From Figure 84, it can be seen that the return type of the readln function must be of IO. Moreover, a function that wish to invoke another function that return IO must start with the do keyword.

```
main = do putStr "What is your name?"
          putStr "How old are you?"
          putStr "Nice day!"
```

Figure 85: Invocating function that returns monadic IO in Haskell

However, this approach differs greatly from the one discussed in Section 3.3.4, the latter is much more simplistic, as it only provide distinction at syntax level, meanwhile Haskell's approach provide distinction at a semantic level.

2.5 Brief Theory Of Compiler

In short, one can think of the compiler as a program that receive a source program as input and output an output program which may be machine code, byte code or even other programming language, as demonstrated in Figure 86 (Aho, 2012).

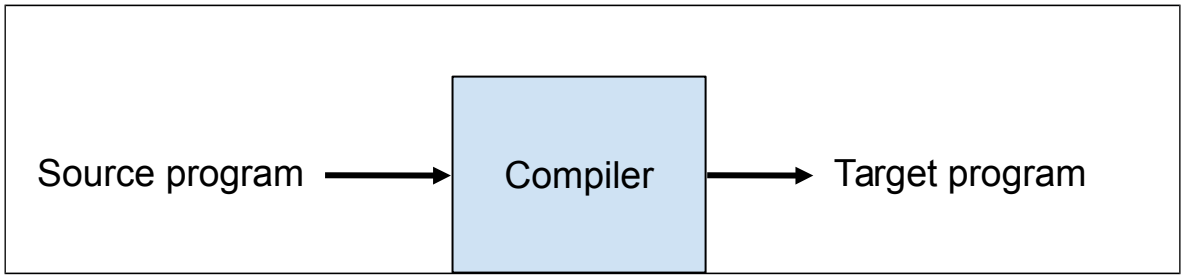

Figure 86: Simplified structure of a compiler (Aho, 2012)

It should be noted that compiler are different from interpreter, because interpreter does not produces a program based on source code, however it executes the code on the fly.

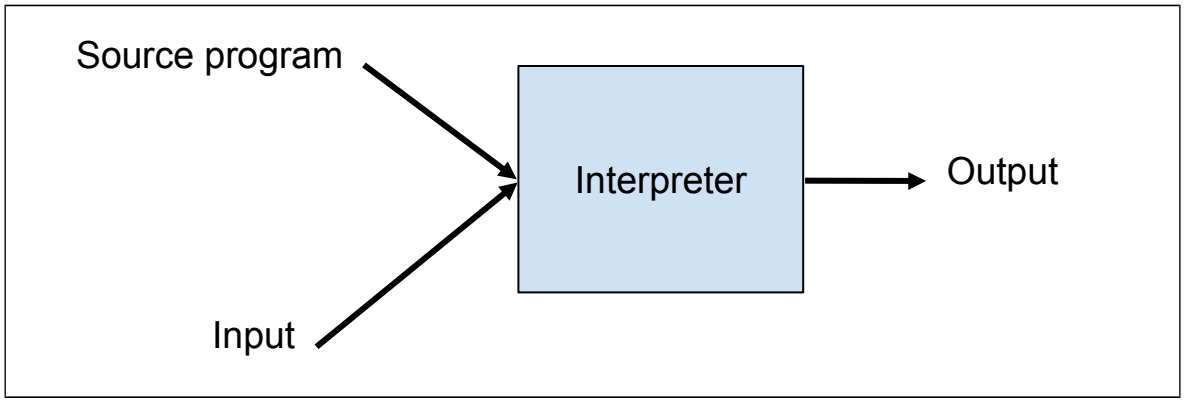

Figure 87: Simplified structure of an interpreter (Aho, 2012)

Example of compiled language are C, Java and C#; meanwhile examples of interpreted language are Python, PHP and JavaScript.

2.5.1 Structure Of A Compiler

Despite the simplistic representation in Figure 86, a compiler is actually a complex program which consists of many phases as in Figure 88. Aho (2012) mentioned that in general, the compiler consist of two process, the analysis process and the synthesis process. The analysis process are lexical analysis, syntax analysis and semantic analysis, meanwhile the synthesis process mainly involves phases of code generation.

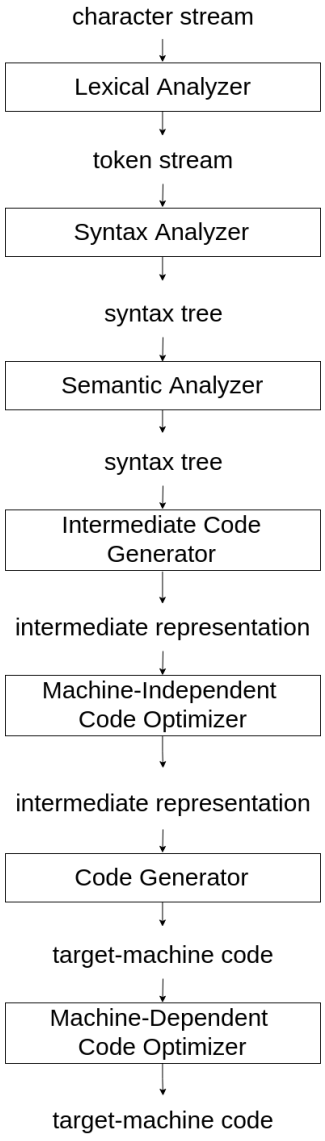

Figure 88: Phases of compiler (Aho, 2012)

A. Lexical Analysis

According to Aho (2012), the very first phase of a compiler is the lexical analysis phase, a process where streams of characters are groups as tokens (aka. lexemes). For example, a stream of character "int x = 5" will produces tokens as in Figure 89 after passing through the lexer.

```
Token 1 : <keyword "int">

Token 2 : <identifier "x">

Token 3 : <operator "=">

Token 4 : <number "5">
```

Figure 89: Tokenized characters

During this phase, only lexical error will be detected, i.e. if the lexer found some characters that it cannot group together. For example, the code in Figure 90 will result in lexical error (assume the language is C).

```
int @x = 5
    ^ Lexical error: Unexpected '@' symbol.
```

Figure 90: Example of lexical error

However, the lexer will not detect syntax error (aka. parsing error) or semantic error. Thus, the code in Figure 91 (which is actually erroneous) will safely pass through the lexer (assume the language is C).

```
// Syntax error: 'int' is declared more than once
int int int myNumber = 5
```

Figure 91: Syntax error cannot be caught by lexer

B. Syntax Analysis

Aho (2012) said that in this phase, the parser (aka. syntax analyzer) will take in the tokens as produced by the lexer can construct a syntax tree.

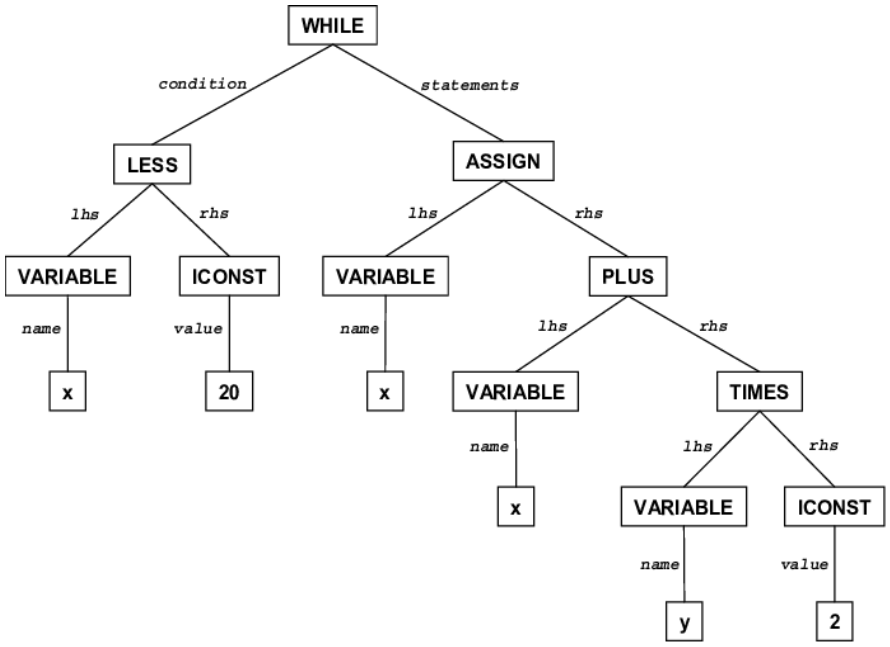

Figure 92: Example of a syntax tree (Fritzson, Privitzer, Sjölund and Pop, 2009)

Besides the diagram representation as seen in Figure 92, a syntax tree can actually be easily represented in JSON (see Figure 93).

```
// repr means representation
// lhs means left hand side
// rhs means right hand side
{
      kind: "Assignment",
      lhs: {
            kind: "Variable",
            dataType: "int",
            repr: "x"
      },
      rhs: {
            kind: "Number",
            repr: "5"
      }
}
```

Figure 93: Examples of representation of syntax tree in JSON

C. Semantic Analysis

During this phase, the compiler will analyze the correctness of the program, such as type checking and null checking.

```
int x = "12345"
        ^ Error: Cannot assign `string` to `int`
```

Figure 94: Example of type checking.

However, semantic analyzer cannot detect high-level bugs such as logical error or architectural bug. For example, the compiler will never know if the behavior of the program is correct, as depicted in Figure 95. The only way to reveal such bug is through testing.

```
if(score > 80) {
      result = "C"; // This should be "A" instead of "C"
}
```

Figure 95: Examples of logical error that cannot be detected by semantic analyzer

D. Intermediate Representation Generation

According to Aho (2012), during the process of converting source code to target program, a compiler may build several intermediate representations (IR), examples are syntax tree and three-address code. He also states that syntax tree is mainly for semantic analysis while three-address code is primarily for code optimization.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Figure 96: Example of three-address code

E. Code Generation

The last phase of a compiler is to map the intermediate representation of the source code and maps into the target language which are usually lower-levelled (Aho, 2012). The target language might differ depending on the requirement. For instance, C/C++ compilers will translates its source code into machine language of the target platform, meanwhile Java compiler will translate its source code into Java Bytecode, which is independent of platforms. Some compiler even compiles into high level language, for example Typescript compiler will translate Typescript into JavaScript. Such compiler are also known as transpiler (Sengstacke, 2016).

2.5 Compiler Development Methodology

In this section the methods to develop the components of a compiler will be discussed. The components are lexer, parser, static checker and code generator. Predominantly, there are only three ways to create a compiler, the first way is to build every component of a compiler from scratch; the second way is to use generators to build a fully functional compiler; the last way is to use a combination of both, meaning to write some component from scratch and use generators for some other components.

2.5.1 Methods To Develop A Lexer

Generally, there are 2 methods to develop a lexer, one way is to create the lexer manually from scratch, the other method will be using lexer generator.

A. Handwritten Lexer

Basically, lexer is to tokenize stream of characters into tokens. Thus the Typescript code in Figure 97 can be considered as a legit lexer, despite its simplicity. However, the lexer can only pass a language that is space-sensitive.

```
function lex(input: string): Lexeme[] {
      const tokens = lex.split(" ");
      const lexemes: Lexeme[] = [];
      for(var i = 0; i < token.length; i++) {
            if(tokens[i].containsDigit())
                  result.push(new Lexeme("Number", tokens[i]);
            else if(tokens[i].containsOperator())
                  result.push(new Lexeme("Operator", tokens[i]);
            else if(tokens[i].containsAlphabet())
                  result.push(new Lexeme("Variable", tokens[i]);
            else
                  throw new Error("Unrecognized lexeme at " + i);
      }
      return lexemes;
}
```

Figure 97: A simple handwritten lexer

B. Generated Lexer

Despite writing the lexer from scratch, one could use lexer generator such as lex to

generate a lexer. This lexer generator is based on regular expressions. A sample code of lex is shown in Figure 98.

```
%%
[a-zA-Z]+  IDENTIFIER
[0-9]+     NUMBER
"="        ASSIGNMENT_OPERATOR
%%
```

Figure 98: Example of lex program

2.5.2 Methods To Develop A Parser

Similarly, there are also two general methods to develop a parser, the hand written one and the generated one.

A. Hand Written Parser

One way to build a parser from scratch is to use the recursive-descent parsing algorithm. Moreover, a transition diagram may be used to visualize the parsing process (Aho, 2012). Although it is widely accepted that handwritten parser is complicated, it has the advantage of providing better error message for syntax error, as one can fine tune how the error should be raised or whatsoever (Software Engineering StackExchange, 2010a).

B. Generated Parser

To generate a parser, one can use tools such as Bison. According to GNU Project (2014), Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR (left-to-right) or generalized LR (GLR) parser employing LALR (1) parser tables. This topic is too broad thus shall not be covered in this paper. A sample of Bison program can be seen below. One advantage of using parser generator is that one do not have to worry about too much complexity and can get things done very quickly, however the error message thrown by the generated parser will be hard to modify. Another drawback of using parser generator(PG) is that if the PG contains bug, it will be very hard to debug, as the user have to determine whether some error occur due to the PG internal error or the grammar error done by user. Besides Bison, there is another popular parser generator ANTLR (ANother Tool for Language Recognition) which is recognized as

more powerful as it can handle LL grammar (Stack Overflow, 2008).

```
Statement
      : AssignmentStatement
      | ReturnStatement
      ;

AssignmentStatement
      : Variable ASSIGN_OPERATOR Expression
      ;

Expression
      : Variable
      | Number
      | String
      ;
```

Figure 99: A sample Bison program

2.5.3 Method To Develop Compiler

As mentioned before, one way to build a compiler is to build every component of it from scratch; meanwhile, the other way is to use compiler generator or framework. One popular compiler generator is YACC (which stands for Yet Another Compiler Compiler), basically, it is just a tool that combine lexer generator with parser generator. Another more sophisticated way is to use compiler infrastructure such as the LLVM Compiler Infrastructure to allow user to build compiler that have code optimization and can generate cross-platform binaries.

2.6 LLVM Compiler Infrastructure

This section is a literature review about the paper authored by Lattner and Adve (2004), *which* is *LLVM: A compilation framework for lifelong program analysis & transformation.* Lattner and Adve (2004) stated that program analysis and transformation should be applied throughout the lifetime of a program so that its efficiency can be maximized. Thus, they presented LLVM (Low-Level Virtual Machine), which is a compiler framework that allow user to achieve the aforementioned goal through two parts: an intermediate representation (IR) that eases code analysis and transformation; and a compiler design that exploits the IR to provide groundbreaking optimization.

Lattner and Adve (2004) describe LLVM IR as an enhanced RISC-like instruction set that allow effective analysis. They further states that the LLVM IR is source-language-independent but it is not intended to be a universal compiler IR, thus LLVM should be treated as complementary to other high-level virtual machines like JVM and Microsoft's CLI, but not alternatives. In fact, LLVM can be used to implement high-level virtual machines. In short, LLVM have the following capabilities which Lattner and Adve (2004) believe that no preceding framework provided all of them at once:

1. Persistent program information
2. Offline code generation
3. User-based profiling and optimization
4. Transparent runtime model
5. Uniform, whole-program compilation

The LLVM instruction sets is a set of operations of ordinary processors which do not include machine-specific construct such as physical registers, low-level calling conventions and pipelines. However, LLVM bestow infinite virtual registers that can store primitive type's information (i.e. Boolean, integer, floating point and pointer), which can be used in the form of Static Single Assignment (SSA). Lattner and Adve(2004) mentioned that there are only 31 opcodes in the LLVM instruction set, which is made possible due to polymorphism or overloading (which is not a feature in most assembly language), for example, the add instruction can operate on both integer and floating point.

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

Figure 100: Example of LLVM IR code (Anderson, 2009)

Lastly, they showed that LLVM IR is in fact space-efficient, as it have almost similar size with X86 native code. Moreover, they proved that LLVM can perform faster optimization than GCC (GNU Compiler Collections).

2.7 Hindley/Milner (HM) Type System

The Hindley/Milner (HM) type system is a classical type system which allow parametric polymorphism. It was first explained by J. Roger Hindley and later rediscovered by Robin Milner, thus the name Hindley/Milner type system. This section is a literature review about the paper authored by Milner (1978), *A theory of type polymorphism in programming*. Milner (1978) states that polymorphic functions are good as they offer flexibility, but the lack of type checking for polymorphic function can cause programmers to fall into a tedious debugging cycle. For example, in JavaScript (JS), the plus operator have at least 2 overloaded semantics: to add numbers, or to concatenate strings. However, since JS is dynamically typed, code such as 99 + "77" can run without error. Although such behavior is good for learning the language, but it is often undesirable as it will breed bug that is hard to be found. Thus, Milner (1978) wanted to present a type system that can offer both: functions that are polymorphic and does not sacrifice type safety. In short, the type system can be summarized as a list of rules, as shown in Figure 101. In addition, those rules are proven to be true by Milner (1978) and Hindley (1969) using discrete mathematics.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Figure 101: The Hindley/Milner Type System rules (Stack Overflow, 2012)

The meaning of each symbols are as shown in Table 3.

Symbol	Meaning
:	Has type of
∈	Is element of
Γ	Given environment
⊢	Proves that
,	And
Horizontal bar	[above] implies [below]

Table 3: Meaning of symbols in Hindley/Milner rules

The meaning of each rule are explained as follows:

[Var] Variable

Given a variable V have type of T where T exist in environment E, then from E, one can conclude that V has type of T.

[App] Function Application

If an environment E ensure that a function F have type of T1→ T2 and an expression X have type of T1, then from E, one can conclude that F(X) has type of T2.

[Abs] Function Abstraction

If an environment E ensure that an expression X have type of T1 and an expression Y have type of T2, then from E, one can conclude that an anonymous function F that takes in X and returning Y have type of T1→ T2.

[Let] Let variable declaration

If an environment E assert that an expression X have type of T1 and if within E, an expression Y that have type of T1 and an expression Z have type of T2, then from E, one can substitute X with Y whenever X appear in Z.

[Inst] Instantiation

If an environment E assert that an expression X have type of T1 and T1 is a subtype of T2, then from E, one can conclude that X have type of T2.

[Gen] Generalization

If an environment E assert that an expression X have type of T, and A is not an element of the free variables of E, then one can conclude that from E, X has type of T for every expression A that returns type of T.

2.7.1 Application of HM Type System

One application of HM type system is that it allow programming language to have high flexibility and compactness of dynamic languages such as Python and Ruby, yet offering type safety of statically-typed language like C++ and Java. One programming languages

that are built on top of HM type system is Haskell. A comparison of Python, Java, and Haskell is shown in Figure 102.

```
# Python: Terse code,
# nut no compile error, only run-time error
x = [1,2,3]
y = x - 20

// Java: Can catch type error at compile time,
// but code is verbose
List<Integer> x = new ArrayList(){1,2,3};
int y = x - 20;

-- Haskell: Terse code, no type annotation
-- yet, type error can be caught at compile time
-- by using type inference
x = [1,2,3]
y = x - 20
```

Figure 102: Type inference vs. dynamic typing vs. static typing

Due to this benefit, the proposed language should use HM type system to allow type inference.

2.8 Programming Language And Human-Computer Interaction (HCI)

This section is a literature review about the paper authored by Grudin and Norman (1991), *Language Evolution and Human-Computer Interaction*. They stated that the four characteristics of a usable language are clear, quick and easy, expressive, and processible.

Clear means that there should be a consistent mapping between the language's surface form and its underlying semantics. However, they further mentioned that there are no natural language that is completely clear, as there are always inconsistencies in every language.

The second criterion is *quick and easy*, this criteria actually conflicts with the first criterion *clear*, because word comprehension speed is crucial for effective communication. Grudin and Norman (1991) disclosed that this criterion can be commonly seen in any natural language, for example in English, all common words are almost monosyllabic, e.g. *"I", "You", "We", "They", "is", "are", "not"* etc. Moreover, new words that are commonly used tend to be abbreviated, for instance, the word "television" is more commonly pronounced as "TV" instead due to its popularity. In the perspective of HCI, this phenomenon is also common, for example in UNIX shell, almost every commonly used command are abbreviated, e.g. ls, cd, mv, cp etc. However, *quick and easy* usually introduce inconsistencies, thus the language might contains ambiguity, which make it violates the *clear* criterion.

The third criterion is *expressive*. Grudin and Norman (1991) argue that in order for a language to be expressive and efficient, it must be compressed, thereby sacrificing consistency, and introduces more complexity to the language. They also states that this can be observed on programming language, when a language becomes more and more expressive, it becomes more and more complicated. For example, C++ is a complex language which offers high expressivity, meanwhile SQL is a more simplistic language, but the expressiveness is limited.

The last criterion is *processible*. Grudin and Norman (1991) defined *processible* as the comparability between the rate which the speaker encode and the rate which the listener decode. Some programming language such as Perl offer very fast encoding rate (i.e. once can comprehend code very fast), but the decoding rate is very slow, to the extent that some Perl's opponent said that Perl is a *write once, read never* language. On the contrary, language such as Java offers very slow encoding rate (i.e. one needs to type a lot to achieve something minor), but its decoding rate is quite fast, where one can read and understand the

code quickly. In the definition of Grudin and Norman (1991), languages such as Java and Perl are *not processible*, because the rate of encoding and rate of decoding is imbalance. Thus, language designer should strike a balance between encoding speed and decoding speed.

In my opinion, these four criteria can be visualized in a matrix such as Johari window, as shown in Figure 103. They can be visualized as such because each of the criterion are contradicting with each other, just like playing tug-of-war.

Clear	Quick and Easy
Processible	Expressive

Figure 103: Language criteria matrix

For instance, if a language must be clear, it will be not be quick and easy, as seen in Figure 104.

Clear	Quick and Easy
Processible	Expressive

Figure 104: Clear-ness suppresses quick-and-easy-ness

On the contrary, if a language is too expressive, it might be less processible (see Figure 105).

Clear	Quick and Easy
Processible	Expressive

Figure 105: Expressivity kills processability

In conclusion, language designers should not be day-dreaming of creating a language that will satisfy all the 4 criteria, as it is impossible. Instead, they should try to balance each of the criterion, so that none of the criterion will tower over the others.

CHAPTER 3

METHODOLOGY

3.1 Development Methodology

3.1.1 Test-driven development

The development methodology that will be used is Test-Driven Development (TDD). This process is chosen because it suites the nature of this project, as Keli shall start as a small language and gradually adding more features and development progresses. According to Beck (2003), the TDD process can be summarized as three steps (also illustrated in Figure 106):

1. Red—write a little test that doesn't work, perhaps doesn't even compile at first.
2. Green—make the test work quickly, committing whatever sins necessary in the process.
3. Refactor—eliminate all the duplication created in just getting the test to work.

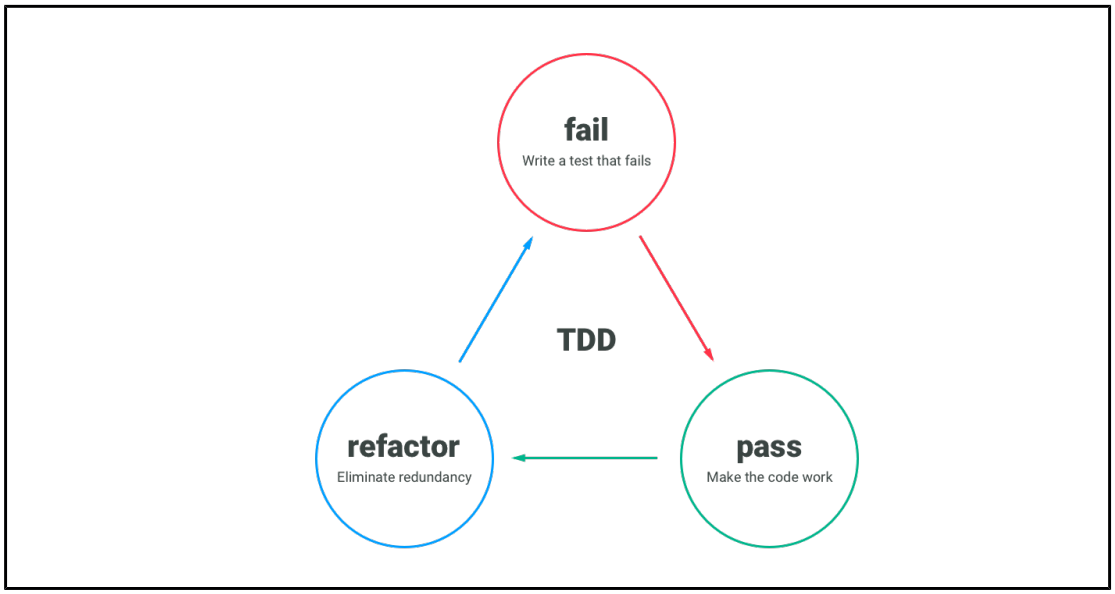

Figure 106: Test-driven development process (PromptWorks, 2018)

```
# Suppose one wants to create a multiply function
# First, write a unit test for it
def test_multiply():
      assert multiply(2,3) == 6

# Secondly, write the code for it, until the unit test passes
def multiply(x, y):
      result = 0
      for i in range(y):
            result += x
      return result

# Thirdly, refactor it
```

Figure 107: Example of TDD in Python

Beck (2003) states that one major advantages of TDD is that fear of writing code can be managed properly. He said that fear is bad as it makes a programmer tentative and grumpy, e.g., if a programmer feels that some part of the code can be refactored , he/she probably would not do that due to fear of breaking other parts of the overall software. However, with TDD, this kind of fear can be managed, as every code refactored will have feedback from the previously written unit tests whether other code are broken or not. Moreover, TDD will promote the habit of writing many small functions instead of big functions, thus improving code reusability. As a consequences, the code base would have increased maintainability.

Furthermore, the research conducted by Maximilien and Williams (2003) revealed several advantages of TDD:

1. Defect rate reduces significantly by 50% compared to ad-hoc unit testing approach
2. Project completed on time with minimal impact on productivity
3. Aided product design that can incorporate late changes
4. Developers have increased morale
5. Code defects can be identified early

Despite that, George and Williams (2004) also pointed out that code produced by TDD programmers exhibit higher code quality than non-TDD programmers, however, TDD programmers spend 16% more time in development. Due to the benefits of aforementioned, TDD will be applied as the development methodology of this project.

3.1.2 Application Of TDD In This Project

TDD will be applied in this project as follows, firstly, create a unit test for a feature-to-be-implemented, then write the code for it until the test passes. Secondly, fix all previous tests that are broken by this new feature implementation. Thirdly, seek out opportunity to refactor the code base, for example, extract common functions, rename variables etc. Finally, repeat from the first step.

Furthermore, the feature-to-be-implemented shall follow the work breakdown structure (WBS) which will be defined in Section 4. For each WBS task, at least one unit test should be written. By doing so, a progress of the project can be tracked properly.

3.1.3 Summary of methodology

The methodology applied in this project can be summarized in Figure 108.

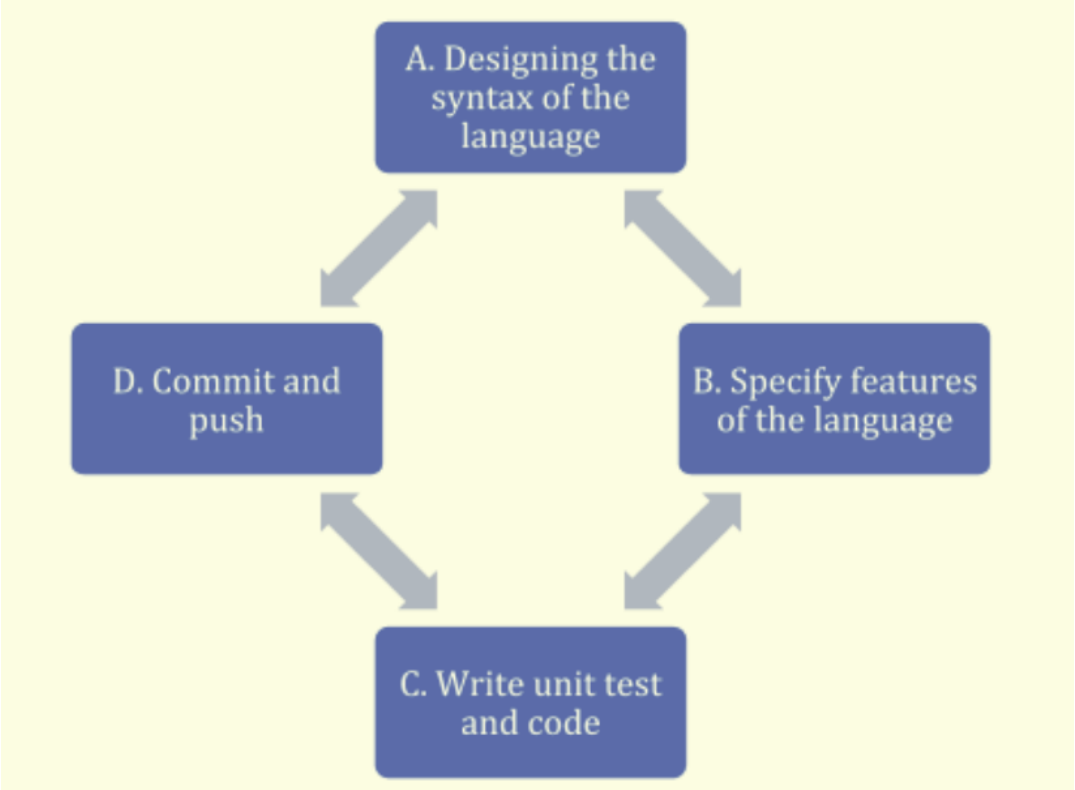

Figure 108: Summary of methodology

A. Designing the syntax of the language

In this step, the syntax of the language will be designed in such a way that it will/may overcome the stated problems in Section 1.2.

B. Specify features of the language

In this step, features that shall be incorporated into the proposed language will be specified, with example given for each features.

C. Write unit test and code

Since TDD is going to be applied, unit test for each specified feature will be written, and for each of them, code will be written to pass corresponding unit test.

D. Commit and push

After each unit test is written and passed, a commit shall be created and the update should

be pushed to a remote repository so that the code can be backup from time to time.

E. Reiterate

As mentioned before, TDD is an iterative cycle, so any aforementioned step may jump to previous step whenever necessary. Thus, one shall not thought that the methodology is a cycle of A →B→C→D, instead it might be A →B→A →B→C→D→B→C. By doing so, the quality of the project can be maintained while a better design will be crafted over time.

3.2 Research Methodology

The research methodology will be surveying developer's websites such as Stack Overflow, Reddit, Quora and Medium etc. Stack Overflow is primarily chosen because according to Anderson et, al. (2012), content published on Stack Overflow tends to have high quality due to the self-regulation mechanism, e.g. duplicated questions, unhelpful answers and unclear questions will be flagged by moderators in a quick manner. Moreover, Anderson et, al. (2012) also stated that there are at least one million questions where 69% of them contains accepted answer. In addition, Vasilescu, Filkov and Serebrenik (2013) also disclosed that there are approximately 25% or 93771 GitHub users are linked to Stack Overflow. Therefore, Stack Overflow is chosen as the site-to-be-surveyed for this project due to its reliability and quality contents.

Besides that, another method employed for researching are observations, that is the features of many programming languages of various paradigm are studied despite their popularity. By observing their features, one can figure out which features are important. Then, based on that, findings can be crafted through logical reasoning.

Lastly, questionnaires are used to obtain some findings. The questionnaire are exclusively given to students of software engineering in UTAR, because they are more familiar to programming languages than those who are not from software engineering, thus they might respond a more valuable feedback.

3.3 Research Findings

The following are the research findings that may resolve the problems stated in Section 1.3.

3.3.1 Solution For 1.3.1

A way to resolve DICUF is to break the prefix-oriented tradition by introducing *mixfix function* to the programming language. To understand what is *mixfix function*, one shall understand the following terms:

- Prefix function
  - Function whose name is in front
  - Example: print (“Hello world”)
- Suffix function
  - Function whose name is at the back
  - Example: (myVariable) toString
- Infix function
  - Function whose name is between parameters/arguments
  - Example: (x) plus (y)

So, *mixfix function* is actually the combination of the affixes aforementioned.

A. Definition Of Mixfix Function

In this paper, the term mixfix function shall means the set of functions which includes prefix functions, suffix functions, infix functions and functions which identifier can be separated by arguments or parameters.

```
// This is a hypothetical language

// Example 1
convert (myString) toInteger

// Example 2
split (myString) by (comma)

// Example 3
replace (oldChar) with (newChar) in (myString)
```

Figure 109: Examples of mixfix functions

From Figure 109, it can be seen that mixfix function directly addresses the problem discussed in , as it will leaves no space for confusion, unless the programmer has very low English literacy.

B. Named Parameters Vs. Mixfix

One who have experienced in language such as Swift, Python or C# might think that named parameters and mixfix functions are equivalent. In short, mixfix functions actually convey clearer semantics but named parameters is a little more verbose, thus the meaning conveyed would be less direct. For example, look at the function will divide two numbers in Figure 110.

```
// mixfix
10 divide 5

// named parameters
divide(dividend=10, divisor=5)
```

Figure 110: Named parameters vs. Mixfix in term of direct-ness

From the example in Figure 110, it is clear that mixfix is not only shorter, it actually conveys clearer meaning although it is terser than the named parameters version. In fact,

human natural language tends to be mixfix rather than named parameters. For example, in English, one would say "*go to school*" instead of *"go to a place which is school"*, because the latter included redundant details, as it is obvious that *school* is a *place*, thus it is not necessary to mentioned the word *place*. Similarly, in the example of Figure 110, when one who understand Mathematics will know that 10 divide 2 means that 10 is the dividend and 2 is the divisor. Furthermore, another advantage of mixfix functions over named parameters is that it allows quick comprehension as the latter is more verbose (Software Engineering StackExchange, 2018b).

Moreover, sometimes mixfix conveys more meaning than named parameters, despite its shorter length. Consider the example in Figure 111, it is clear that the named parameter version is ambiguous but the mixfix version is clear and concise.

```
// Named parameter
tree.insert(child=x, parent=y)
// Does it means insert x as child of y
// OR insert y as parent of x ?

// Mixfix
tree.insert(x asChildOf y)
// No ambiguity here
```

Figure 111: Named parameters vs. Mixfix in term of ambiguity

Thus, mixfix is not only different from named parameters, it is also more effective in terms of improving readability.

3.3.2 Solution For 1.3.2

A. Solution For 1.3.2(A)

Since the existence of several ways to construct a function (static method, instance method, free function, constructor etc.) causes UDFI, the solution is to allow only a single choice. The following paragraph shall explain why *free function* shall be the only choice.

I. Drawbacks Of Static Method

Static method shall be excluded as it introduces meaningless redundancy, as illustrated in Figure 112.

```
function euclideanDistance(x1, x2, y1, y2) {
    return Math.sqrt(Math.pow((x1-x2), 2) + Math.pow((y1-y2), 2));
}
```

Figure 112: Defining Euclidean distance using static method

The word Math is redundant, as it obscure the meaning of the code. The code would be much clearer if the word Math is absent (see Figure 113).

```
function euclideanDistance(x1, x2, y1, y2) {
    return sqrt(pow((x1-x2), 2) + pow((y1-y2), 2));
}
```

Figure 113: Defining Euclidean distance using free functions

Due to the fact that static methods might obscure the meaning of codes, static method shall not be considered as the best way for constructing functions.

II. Drawbacks Of Instance Method

One major drawback of instance method is that it forces the function to be suffixed, and causes some code to read unnaturally as it makes statements to be in Object-Verb structure instead of Verb-Object structure. For example, to sort a list in JavaScript, one is required to write list.sort() instead of sort(list) which is arguably unnatural (as compared to English). On top of that, instance method cannot be defined intuitively as the difference between definition and invocation is immense (see Figure 114).

```
// JavaScript
class List {
      // To define instance method
      public sort() {
            /* method body */
      }
}

// To invoke instance method
let x = (new List()).sort();
```

Figure 114: Definition of instance method is quite different from its invocation

Another drawback of instance method is that sometimes it is not easy to decide where to place the desired method. For example, in Figure 115, one can place the join method either in the haystack class or the needle class.

```
// Typescript
// Variant 1
class StringList {
      join(needle: String) { /*method body*/ }
}

// Variant 2
class String {
      join(haystack: StringList) { /*method body*/ }
}

// Invoking join of Variant 1
["duck", "cat", "dog"].join(" and ")

// Invoking join of Variant 2
" and ".join(["duck", "cat", "dog"])
```

Figure 115: Variants for defining join method

In such situation, the function designer would hesitate where to put the method. Nonetheless, it should be noted none of the variants in Figure 115 is superior over the other. In fact, JavaScript uses Variant 1 and Python uses Variant 2 (MDN, 2018; Python, 2018). This is because both variants are equally ambiguous, as the verb is in the middle of the

statement, instead of being at front. If it is define using mixfix function, it could be much clearer (see Figure 116).

```
// Definition
function join (stringList) by (string) {
      /* body */
}

// Invocation
join (["duck", "cat", "dog"]) by ("and")
```

Figure 116: Definition of join using mixfix function

The reason why instance method does not fit well into such situation is because the join function is non-commutative (which means arguments order is important). That is to say, instance method actually fitted well when the binary function is commutative (meaning the position of argument is unimportant). Example of commutative functions are equals, add, multiply etc. But, such function are uncommon, even in the field of mathematics (as in Figure 117). Hence, instance method shall be excluded as well.

Commutative operations:	
Addition	A + B = B + A
Multiplication	A * B = B * A
Non-commutative operations:	
Division	A / B ≠ B / A
Subtraction	A – B ≠ B – A
Exponentiation	A ^ B ≠ B ^ A
Modulo	A % B ≠ B % A

Figure 117: Examples of commutative and non-commutative operations in Mathematics

III. Advantage Of Free Functions

Firstly, free functions does not have the downside of static method and instance method. Secondly, free functions can be defined intuitively, as there are not much difference between its definition and invocation (see Figure 118).

```
// JavaScript
// To define free function
function sort(xs) {
      /* function body */
}

// To invoke free function
sort([1,2,3,4]);
```

Figure 118: Definition of free function is similar to its invocation

Furthermore, free function can actually emulate instance method, if suffix function is supported (see Figure 119).

```
// Hypothetical language
// Creating a suffix function
function (thisNumber).isEven {
      return thisNumber % 2 == 0
}

// To invoke it
let result = (999).isEven
```

Figure 119: Emulating instance method with suffix function

IV. Conclusion

Due to the aforementioned reasons, option C (free functions) shall be the only choice to be retained. Thus, it implies that object-oriented features shold not be fully supported in the proposed language.

B. Solution For 1.3.2(B)

Disallow mutable inputs, meaning that every function should be pure and will not mutate the input. In other words, every function parameters are passed by value instead of reference.

C. Solution For 1.3.2(C)

Disallow the usage of global variables, i.e., no variable can be declared outside the scope of a function.

D. Solution For 1.3.2(D)

Mixfix function, as it can solve the ambiguous arguments position, as discussed in .

3.3.3 Solution For 1.3.3

Separate the *declaration of interface implementation* from the *declaration of class definition*. For example, see Figure 120 (which is a hypothetical language that looks similar to Typescript).

```
// Declaration of class definition
public class Person {
      public name : string;
      public age  : number;
}

// Separated interface implementation
// which is plausible to be written in another file
Person implements Comparable<Person> {
      public compareTo(other: Person) {
            if(this.age === other.age) return 0;
            else if(this.age > other.age) return 1;
            else return -1;
      }
}
```

Figure 120: Separation of interface implementation from class definition

By doing so, one does not need to inherit from the existing class to implement new interface, thus the name of the original class can be kept as so, leaving the original intention of the class to be clear. Moreover, this allows programmers to truly adhere to the Open-Closed Principle, this is because they do not need to create a new class (which results in confusing class name such as ComparablePerson) whenever they need to extend the functionality of existing class. The meaning of Open-Closed Principle can be read from Martin (1996). The difference between Java's interface and Haskell's typeclass can be found at (Wei, 2011).

3.3.4 Solution For 1.3.4

To make the distinction clear between I/O operation and non-I/O operation, one possible solution is to mandate that each invocation on I/O operation should be annotated with some symbols. Let say the symbol is asterisk (*), then code should look resemble Figure 121.

```
let number1 = *readline()
let number2 = *readline()
let result  = parseInt(number1) + parseInt(number2)
*print(result)
```

Figure 121: Every I/O operation is tainted by asterisk

By using annotation, code such as above would be much more easier to debug, as one just have to focus on those functions calls that are annotated with asterisk, instead of diving into the definition of every function that is invoked. Despite the increased easiness for debugging, another possible benefits is that it allows a programmers to easily identify which part of the code should be redesigned or refactored, by looking for functions which contains a lot of asterisks.

3.3.5 Solution For 1.3.5

Allowing type annotation can solve such problem. According to Pierce and Benjamin (2002), type declaration actually serves as a form of documentation, which is useful when reading programs. For example, the code in Figure 21 can be understood a lot better with type annotation (See Figure 122).

```
def draw(shape: List[Coordinate]):
     # Function body
```

Figure 122: Annotating argument with type information

In fact, Python 3.6 did include an optional type annotation for variables and function arguments (Python Software Foundation, 2016). Therefore, the code in Figure 122 is actually valid since Python 3.6.

3.3.6 Solution For 1.3.6

Mutability should be explicitly defined, i.e. to declare a variable as mutable should require more effort than its counterpart, as shown in Figure 123.

```
let x = 10 // Immutable
let mutable y = 20 // Mutable
```

Figure 123: Variable mutability declared explicitly

As a consequence, programmers will tend to use immutable variables as that require less effort.

3.3.7 Solution For 1.3.7

Disallow implicit coercion and make sure user have to explicitly convert a value to the desired type, if not type error shall be raised. For example, the code in Figure 124 shall raise compile error, rather than runtime error.

```
let myNumber = "123"
let result = myNumber + 2
 // Error: Can't apply (+) to String and Number
```

Figure 124: Implicit coercion raising compilation error

3.3.8 Solution For 1.3.8

To resolve the ambiguity, every symbols in Keli shall have only one meaning that is independent of the context. For example, the colon symbol shall means *type-of* no matter which part of the code it appears. It is important to note that it would be hard to achieve 100% context-free, so a 95% context-free should be acceptable.

3.4 Development Tools

3.4.1 Git And Github

Git is a distributed version control system (VCS), which can help developers to keep track

of their project. Git is used because it reduce the fear of changes. In an environment without Git or any other VCS, developers tend to not modify their code when the application is working already, due to fear that any changes might break the whole application. Such behaviors is defective in software development, as it will causes the code base to be more and more messy, since developers do not want to refactor their code, consequently the codebase will be untestable and the project would not able to be delivered with expected quality. Git prevent such issue as developers can revert changes to their last committed changes easily.

Github is used as a backup of source code, so that even if the developers' laptops are lost or broken due to some unforeseen reason, the code will still conserved online.

3.4.2 Visual Studio Code

Visual Studio Code (VSCode) is an enhanced code editor developed by Microsoft which include a lot of features like built-in Git support, Intellisense, powerful extension mechanism etc. VSCode is different from Visual Studio in a few perspective. Firstly, VSCode is open source and 100% free while Visual Studio requires licensing fees; secondly, VSCode is cross-platform while Visual Studio can only be used on Windows platform. VSCode is used in this project as it will improve developer productivity through Intellisense. Moreover, since Typescript is also a Microsoft's project, VSCode actually have first-class support for Typescript, thus it will be a perfect match to use Typescript alongside with VSCode. Not only that, VSCode have powerful Git-diff feature, which allow developer to clearly identify which line of code have been changed since the last commit (as in Figure 128).

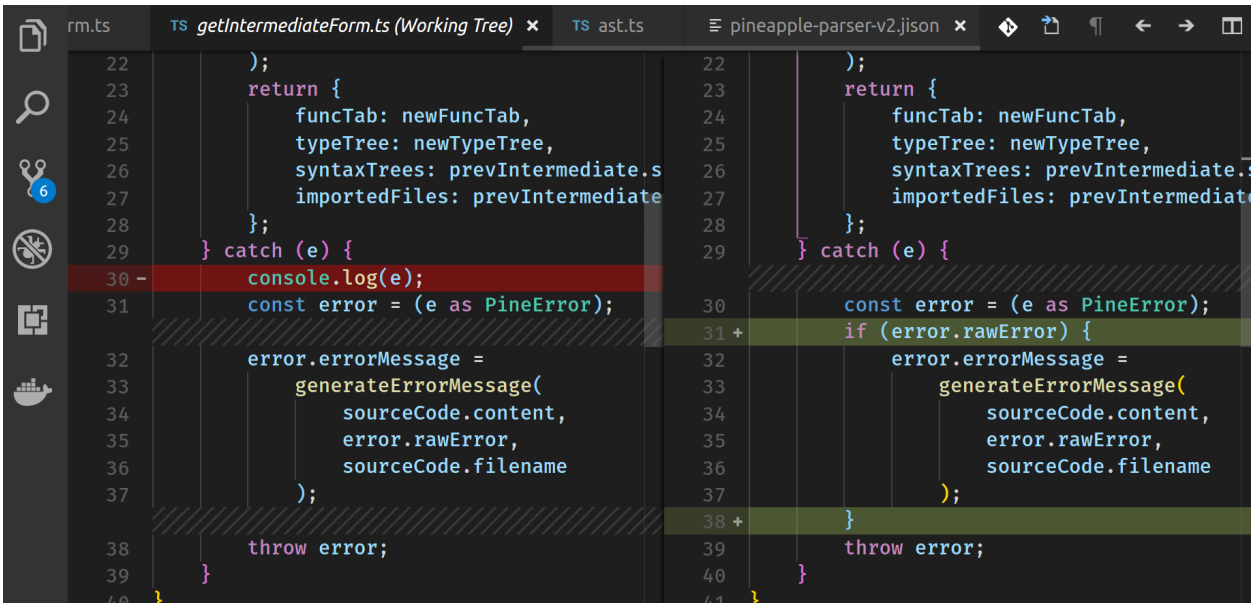

Figure 128: Git-diff feature of VSCode

3.4.3 Haskell

Haskell is a pure functional programming languages. Haskell is chosen to be the primary tools for developing the transpiler of Keli because firstly, it supports applicative parsing, which easifies the parsing process. Secondly, Haskell's type system is sound, which means once the Haskell code can be compiled without error, the resulting program with not have any unexpected runtime error, unless the programmer defines so.

3.4.4 Parser combinators

There is a very powerful library in the Haskell packages repository known as parsec, which is the short form for parser combinators. Unlike parser generators, which do not incorporate type safety at compile time (i.e., you might be building an invalid abstract syntax tree from the source code without you knowing until you run the actual parser), parser combinators can prevent one from building an ill-formed parser by leveraging the power of type checking of the compiler. Parsec is very reliable, as it is not written by any random developers, it is in fact implemented based on a thesis related to parser combinators authored by several programming language researcher. Thus, parser combinators is used in this project to build the parser for Keli.

3.5 Architecture Pattern

The architecture pattern that would be adopted in this project is the data transformation pattern. In this pattern, input data will go through a lot of transformers, where the output of one transformer is the input of the next transformer, and lastly reaching the desired transformed state. The advantage of this pattern is that each transformer can be built and tester separately, thus Separation of Concern (SOC) can be achieved easily.

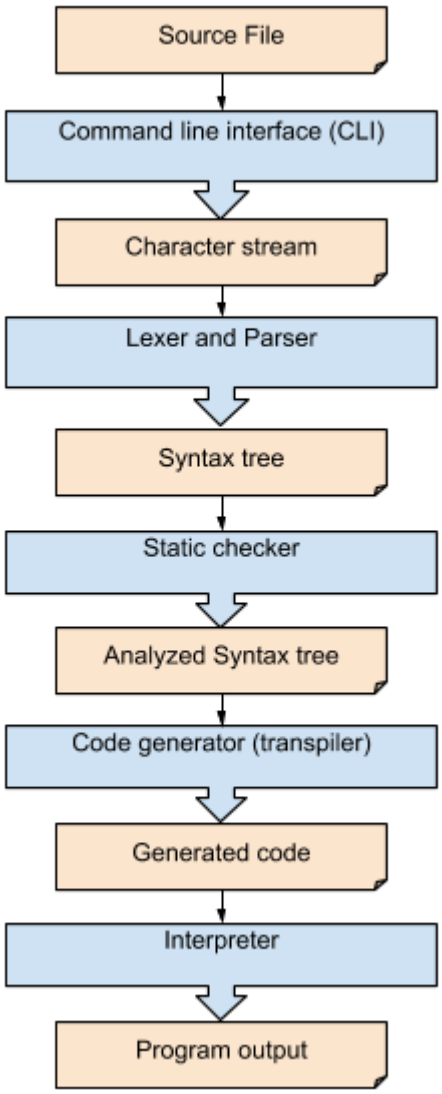

Figure 129: The data transformation architectural pattern

3.6 Project Plan

The plan for the development of this project will be presented using a simplified Gantt Chart in Figure 130.

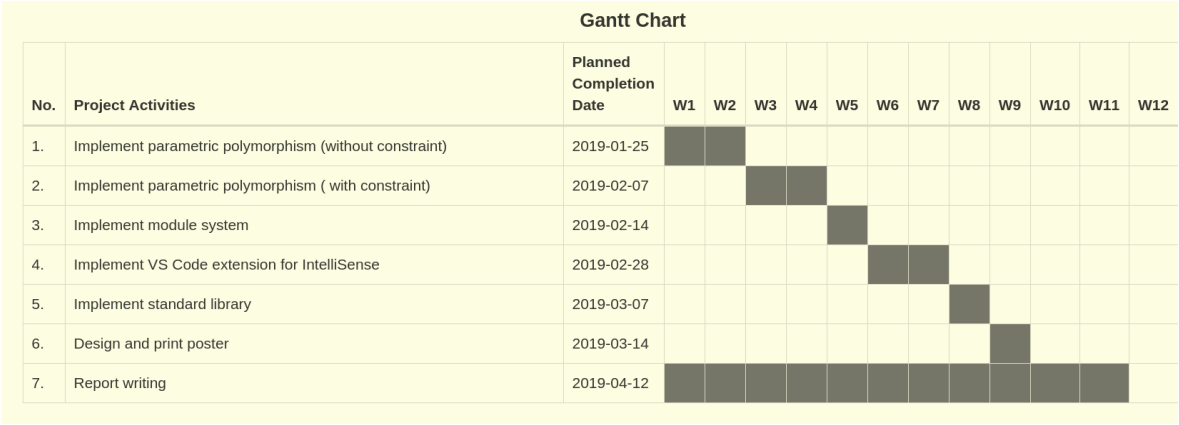

Figure 130: Gantt Chart

Please note that the Gantt Chart above assumes that the following components are already completed beforehand:

1. Lexer
2. Parser
3. Type checker (without parametric polymorphism)
4. Transpiler (for generating JavaScript code from abstract syntax tree)
5. Custom testing framework

This is possible because I begin my development before the start of this trimester (January 2019).

Despite using a Gantt Chart, I also use Github Issues (refer Appendix J) to track the progress of this project. Github Issues is used so that I can link each source code commit to a particular issues, to enhance the documentation of the workflow.

CHAPTER 4

PROJECT SPECIFICATIONS

4.1 Initial Project Specification

4.1.1 Grammar Specification

The grammar of Keli shall be described using EBNF (Extended Backus–Naur form). The notation used shall be based on ISO/IEC 14977:1996(E). Note that the actual implementation will not use the presented grammar, because Jison only takes BNF which is a subset of EBNF.

Notation	Meaning
=	definition
,	concatenation
;	termination
|	alternation
[...]	optional
{...}	repetition
(...)	grouping
"..."	terminal string
'...'	terminal string
(*...*)	comment

Table 4: Notation of EBNF based on ISO/IEC 14977:1996(E)

The initial grammar specification is defined as below (Figure 131).

```
(* Abbreviations:
    Decl = Declaration
    Stmt = Statement
    Id   = Identifier
    Expr = Expression
    Lit  = Literal
    Arg  = Argument
    Op   = Operator
    Var  = Variable
    Func = Function
*)

EntryPoint = {Decl};

Decl
    = EnumDecl
    | ImportDecl
    | StructDecl
    | FuncDecl
    ;

EnumDecl
    = DEF TypeId NEWLINE INDENT {Enum} DEDENT
    ;

ImportDecl
    = IMPORT StringLit NEWLINE
    ;

StructDecl
    = DEF TypeId NEWLINE INDENT {MemberDefinition} DEDENT
    | DEF TypeId NEWLINE INDENT PASS NEWLINE DEDENT
    | DEF TypeId '{' {GenericId} '}' NEWLINE INDENT {MemberIdTypeList}
NEWLINE DEDENT
    | DEF TypeId '{' {GenericId} '}' NEWLINE INDENT PASS NEWLINE DEDENT
    ;

MemberDefinition
    = MemberId TypeExpr
    ;

FuncDecl
    = DEF NulliFuncDecl
    | DEF MonoFuncDecl
```

```
    | DEF BiFuncDecl
    | DEF TriFuncDecl
    ;

(* 0 Arg Func *)
NulliFuncDecl
    = FuncId '->' TypeExpr Block
    | FuncId Block
    ;

(* 1 Arg Func *)
MonoFuncDecl
    = Arg FuncId '->' TypeExpr Block
    | Arg FuncId Block
    ;

(* 2 Arg Func *)
BiFuncDecl
    = Arg FuncId Arg '->' TypeExpr Block
    | Arg OpId Arg '->' TypeExpr Block
    ;

(* 3 or more Arg Func *)
TriFuncDecl
    = Arg FuncId '(' Arg {SubFuncId Arg}')' '->' TypeExpr Block
    ;

Block
    = NEWLINE INDENT {Stmt} DEDENT
    | NEWLINE INDENT JavascriptCodeSnippet NEWLINE DEDENT
    | NEWLINE INDENT PASS NEWLINE DEDENT
    ;

Stmt
    = AssignmentStmt
    | ReturnStmt
    | FuncCall    NEWLINE
    | OpFuncCall  NEWLINE
    | IfStmt
    | ForStmt
    | WhileStmt
    ;

ReturnStmt
```

```
    = RETURN MultilineExpr
    ;

ForStmt
    = FOR VarId IN SinglelineExpr Block
    ;

WhileStmt
    = WHILE Test Block
    ;

IfStmt
    = IF Test Block [{ELIF Test Block}] [ELSE Block]
    ;

Test
    = SinglelineExpr
    | SinglelineExpr LogicOp Test
    | NOT SinglelineExpr
    | NOT SinglelineExpr LogicOp Test
    ;

AssignmentStmt
    = LET VarDecl '=' MultilineExpr
    | VarId '=' MultilineExpr
    ;

VarDecl
    = VarId
    | VarId MUTABLE
    | VarId TypeExpr
    | VarId TypeExpr MUTABLE
    | '(' VarId ')'
    | '(' VarId TypeExpr ')'
    ;

TypeExpr
    = AtomicTypeExpr '|' TypeExpr
    | AtomicTypeExpr '&' TypeExpr
    | AtomicTypeExpr
    ;

AtomicTypeExpr
    = TypeId
```

```
    | TypeId '?'
    | GenericAtom
    | GenericAtom '?'
    | TypeId '{' {TypeExprList} '}'
    | TypeId '{' {TypeExprList} '}' '?'
    ;

MultilineExpr
    = Object
    | Dictionary
    | MultilineList
    | SinglelineExpr NEWLINE
    ;

SinglelineExpr
    = AtomicExpr
    | OpFuncCall
    ;

OpFuncCall
    = OpFuncCall OpId AtomicExpr
    | AtomicExpr OpId AtomicExpr
    ;

FuncCall
    = NulliFuncCall
    | MonoFuncCall
    | BiFuncCall
    | TriFuncCall
    ;

NulliFuncCall
    = FuncId
    ;

MonoFuncCall
    = AtomicExpr FuncId
    ;

BiFuncCall
    = AtomicExpr FuncId '(' SinglelineExpr ')'
    ;

TriFuncCall
```

```
    = AtomicExpr FuncId '(' SinglelineExpr VarId SinglelineExpr ')'
    ;

AtomicExpr
    = '(' SinglelineExpr ')'
    | ObjectAccessExpr
    | SinglelineList
    | FuncCall
    | BooleanLit
    | StringLit
    | NumberLit
    | EnumLit
    | VarId
    | 'nil'
    ;

ObjectAccessExpr
    = AtomicExpr MemberId
    ;

Object
    = TypeId NEWLINE
    | TypeId NEWLINE INDENT {ObjectKeyValue} DEDENT
    ;

ObjectKeyValue
    = MemberId '=' MultilineExpr
    ;

Dictionary
    = '{' '}'
    | NEWLINE INDENT {DictionaryKeyValue} DEDENT
    ;

DictionaryKeyValue
    = StringLit '=' MultilineExpr
    ;

SinglelineList
    = '[' [Element [{',' Element}]] ']'
    ;

MultilineList
    = NEWLINE INDENT MultilineElements DEDENT
```

```
    ;

MultilineElements
    = BULLET SinglelineExpr NEWLINE MultilineElements
    | BULLET SinglelineExpr NEWLINE
    ;

BooleanLit = 'true' | 'false'
```

Figure 131: Initial grammar specification of Keli

4.1.2 Lexical Specification

Note: The matching rule is based on JavaScript's Regular Expression format.

Name	Matching rule	Examples	
		1	2
VarId	`[a-z][a-zA-Z]*`	`myVar`	`x`
FuncId	`[.]([a-z][a-zA-Z0-9]*)?`	`.sum`	`.`
SubFuncId	`[a-z][a-zA-Z]*`	`with`	`to`
OpId	`[~!@#$%^&*-+:;<>=]+`	`+`	`<=`
TypeId	`[A-Z][a-zA-Z0-9]*`	`T`	`BinaryTree`
MemberId	`['][a-z][a-zA-Z0-9]*`	`'age`	`'name`
StringLit	`["].*?["]`	`"Hello world"`	`""`
NumberLit	`\d+([.]\d+)?((e|E)[+-]?\d+)?`	`123`	`123.4e+10`
EnumLit	`[`][a-zA-Z0-9]+`	`` `nil ``	`` `red ``

Table 5: Initial lexical specification of Keli

4.1.3 Structure Chart

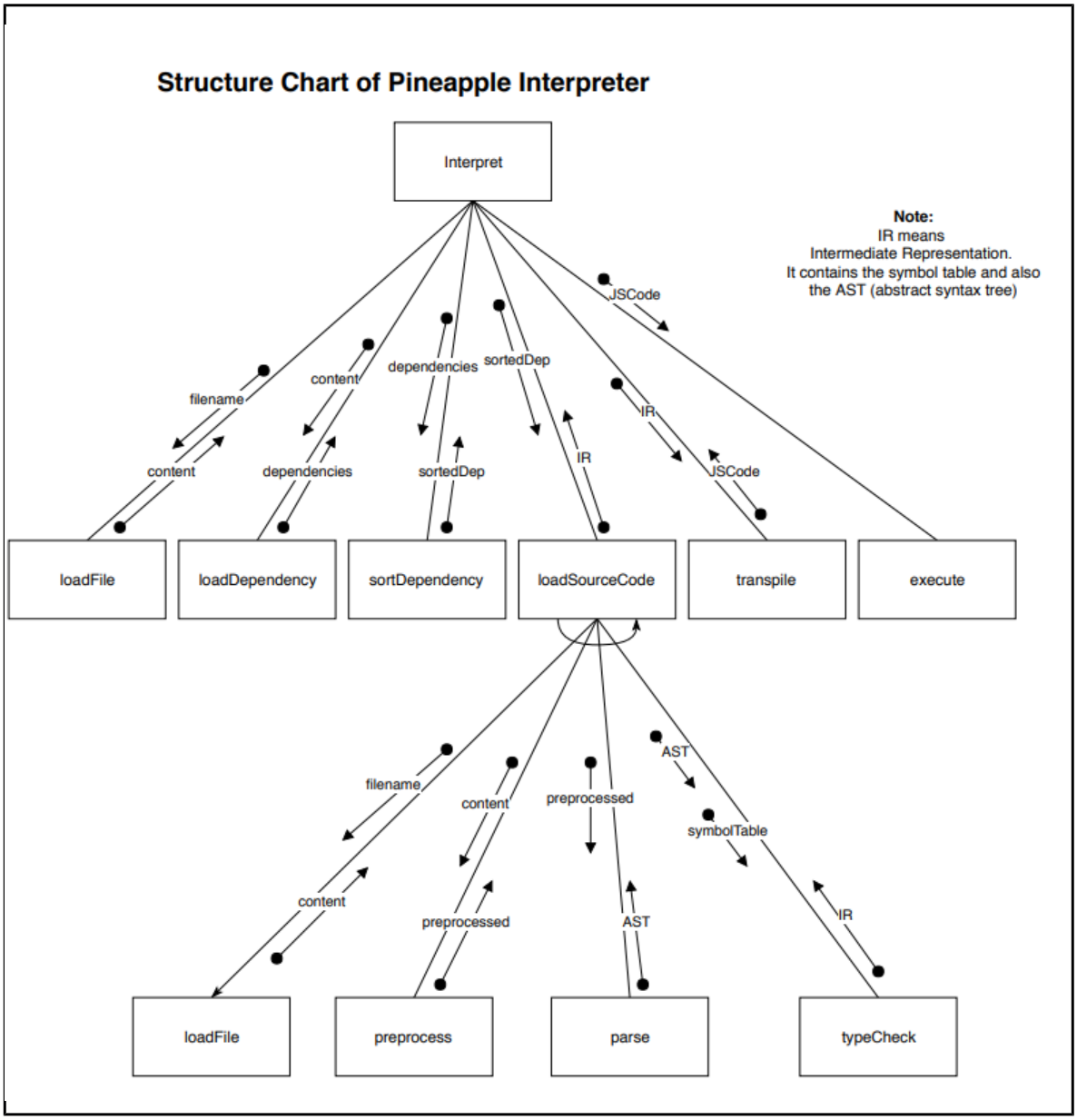

Figure 132: Structure of Keli interpreter

4.2 Revised Project Specification

4.2.1 Keli Language Specification

Refer Appendix E(1).

4.2.2 Abstract Sytax Tree Model

Refer Appendix E(2).

4.2.3 Keli Compiler Architecture

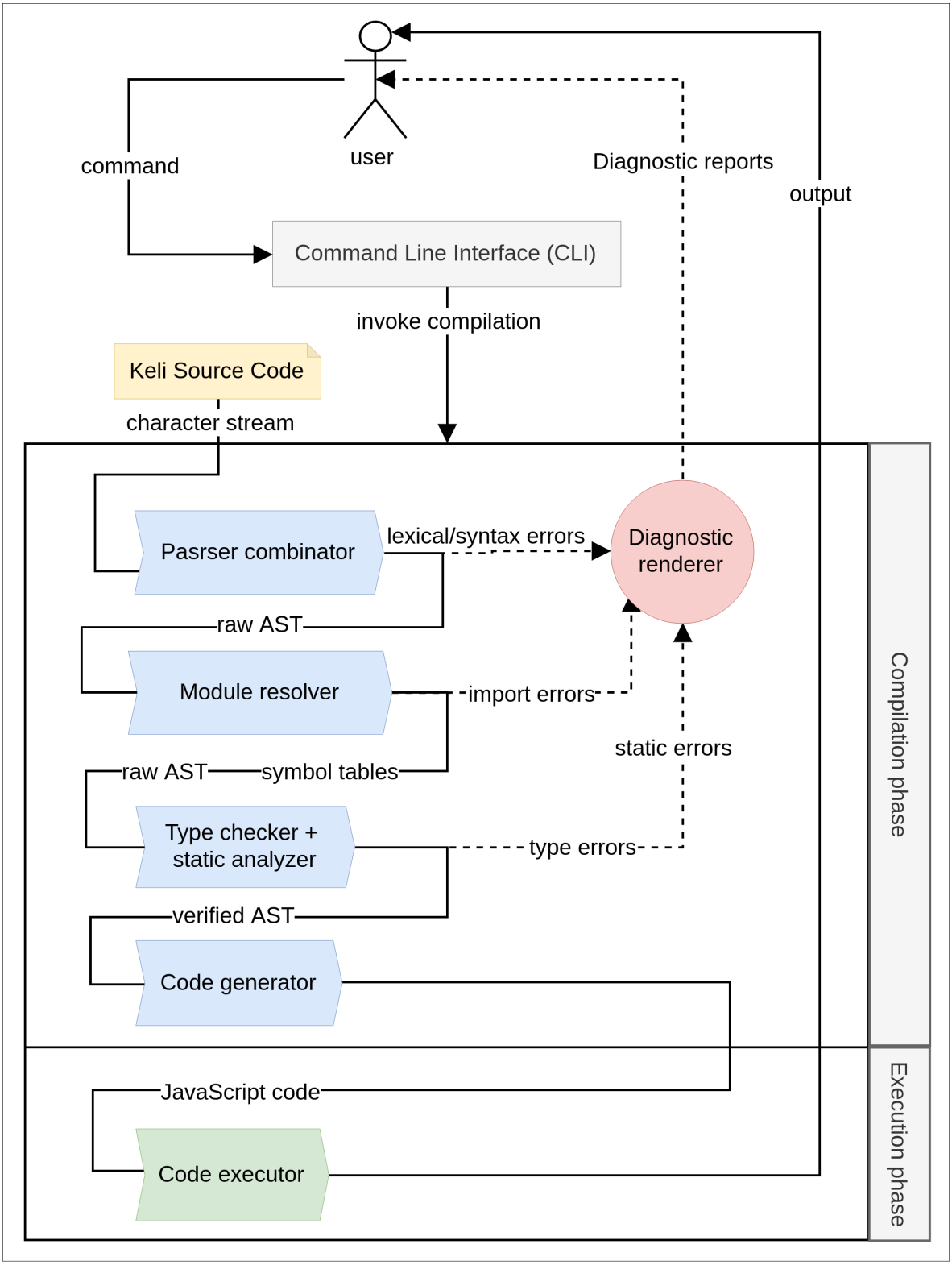

Figure 133: Keli compiler architecture

4.2.4 Keli Visual Studio Code extension architecture

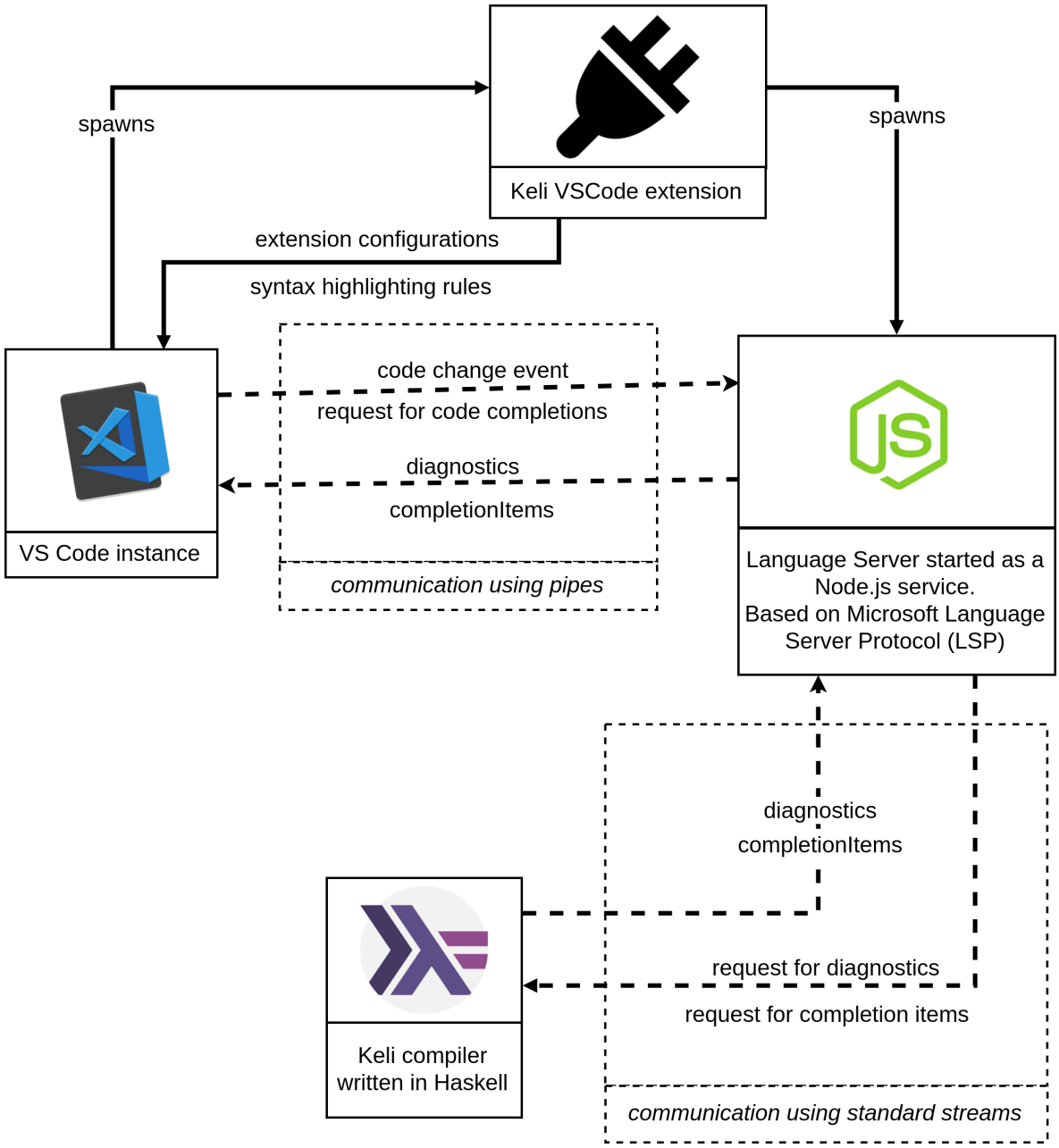

Figure 134: Keli VSCode Extension Architecture

CHAPTER 5

IMPLEMENTATION

In this chapter, the project implementation, which consists of several parts, namely the Keli Language syntax, compiler and development tools will be unveiled as screenshots. Note that the following screenshots only exhibits a small portions of this project, as most of the efforts actually went into implementing static type checking, which consists about at least 100 types of errors.

```
// Single parameter function
// Definition
(this Int).square = this.^(2)

// Invocation
  81  = 9.square

// Double parameter function
// Definition
(this Int).plus(that Int) = this.+(that)

// Invocation
  468  = 123.plus(345)

// Triple parameter function
// Definition
(x Int).<=(y Int) <=(z Int) = x.<=(y).and(y.<=(z))

// Invocation
  "Boolean.True"  = 12.<=(45) <=(56)
```

Figure 135: Keli function definitions and invocations

```
= module.import("../_src/Math.keli")
= module.import("../_src/List.keli")
= module.import("../_src/String.keli")
= module.import("../_src/Box.keli")
= module.import("../_src/Boolean.keli")
```

Figure 136: Module imports

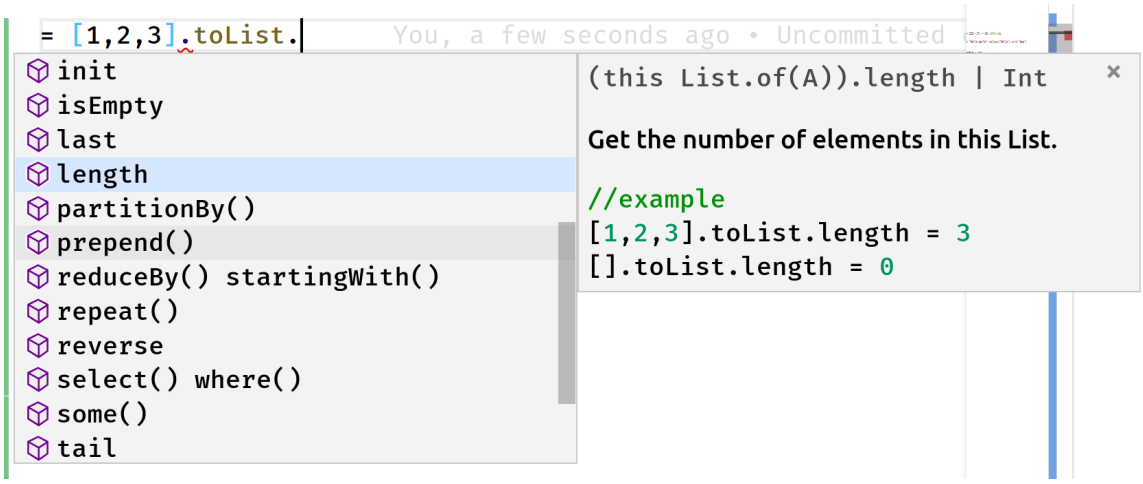

Figure 137: Function Intellisense

```
Shape = choice
    .Circle($.radius(Float))
    .Square($.side(Float))
    .Rectange($.width(Float) height(Float))

(this Shape).area = this.
    if(.Circle(c)) then
        (3.14.*(c.radius.^(2.0)))

    if(.Square(s)) then
        (s.side.^(2.0))

    if(.Rectange(r)) then
        (r.width.*(r.height))

144 = Shape.Rectange($.width(12.0) height(12.0)).area
```

Figure 138: Tagged union definition and usage

```
westernFood = [
    $.name("Burger") price(12.9),
    $.name("Spaggethi") price(99.9)
]
```

Figure 139: Object literal and array literal

```
People = $.name(String) age(Int)

= People.name("Wong") age(99)
```

Figure 140: Record type definition and usage

```
List.of(A Type) = choice
    .Nil
    .Cons($.value(A) next(List.of(A)))
```

Figure 141: Generic inductive type

```
{A Type}
(this List.of(A)).length
    | Int
    = this.
        if(.Nil) then
            (0)

        if(.Cons(c)) then
            (1.+(c.next.length))
```

Figure 142: Generic recursive functions

```
6 | = 123.+("Hello")
```

Demo.keli 1 of 1 problem

```
Expected `Int` but got `String` [keli]
```

Figure 143: Type checking parameter type for function invocation

```
= People.name("wong")
```

Demo.keli 1 of 1 problem

```
Missing properties: age [keli]
```

Figure 144: Static checking – missing record property

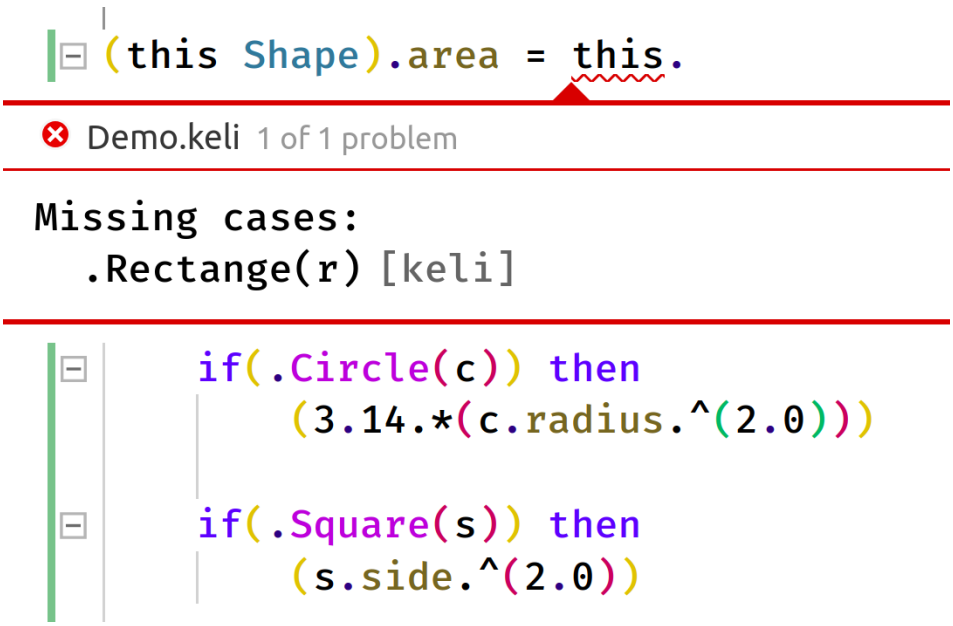

Figure 145: Static checking – missing cases in tagged union pattern matching

CHAPTER 6

EVALUATION

6.1 Evaluation Of Solutions

Two methods are employed to evaluate Keli language. First is the functional specification test, which is used to check if the specification defined is thoroughly implemented, along with its correctness. Second is the usability test plan, which is used to verify if Keli language had met its objective (as defined in Chapter 1).

6.2 Functional Specification Test

6.2.1 Functional Specification Test Plan

6.2.1.1 Introduction

This test plan is structured according to the IEEE 829 Test Plan Template. The objective of this test plan is to describe the required action and non-required action for implementing a test suite for the Keli compiler. This test plan do not strive to be exhaustive (i.e., covering all the features) due to time and budget constraint. Thus, only high priority features and their corresponding negative cases will be included in this test plan. The features to be tested shall be based on the functional specification of Keli.

6.2.1.2 Features To Be Tested

Compiler

1. Packages
    1. Creating new package
    2. Adding dependency to package
    3. Install dependency based on manifest file
2. Modules
    1. Relative imports
    2. Import conflict detection
    3. Encapsulation
3. Declarations
    1. Constants

2. Functions
    1. Single parameter function
    2. Multiple parameters function
    3. Generic functions
    4. Docstring
3. Object type alias
4. Tagged Union
    1. Carryless tag
    2. carryful tag
    3. Non-generic
    4. Generic

4. Expressions
    1. Literals
        1. Number
            1. Integer
            2. Float
        2. String
        3. Constant
        4. Array
            1. Empty array
            2. Filled array
    2. Function invocation
        1. Single parameter function invocation
        2. Multiple parameters function invocation
        3. Chained invocation
        4. Multiple dispatch
        5. Generic function invocation
    3. Lambda
        1. Normal lambda

        2. Shorthand lambda
    4. Object
        1. Construction
        2. Property getter
        3. Property setter
            1. Direct value update
            2. Referential value update
    5. Tag constructors
        1. Carryless tag constructor
        2. Carryful tag constructor
    6. Tag matchers
        1. Exhaustive matching
        2. Non-exhaustive matching
        3. Branch homogeneity
        4. Static analysis
            1. Duplicated tags
            2. Tag's exhaustiveness
        5. Optional bindings
    7. Foreign function interface
        1. Native JavaScript function invocation
        2. Referring constants defined in Keli
        3. Referring tags defined in Keli
        4. Referring functions defined in Keli
        5. Referring symbols imported in Keli
5. Type checking and type inference
    1. Type check function invocation
    2. Type check object construction
    3. Type check tag construction
    4. Type check generic function invocation

5. Type check generic tag construction

6. Static analysis

    1. Syntax checking

    2. Duplicated-identifiers checking

2. Core library

    1. Math functions

    2. Boolean functions

    3. String functions

    4. List functions

3. Visual Studio Code extension

    1. Error diagnostics reporting

    2. Intellisense (i.e. code completion)

6.2.1.3 Features Not To Be Tested

Modules

1. Cyclic import detection

2. Conflict resolution

2. Generic objects

    1. Declaration

    2. Usage

3. Interface

    1. Declaration

    2. Usage

6.2.1.4 Approach

The approach that should be employed is Test-Driven Development (TDD), which is a methodology where test should be written before actual code. In short, TDD is a repetitive 3-steps procedure (Beck, 2003):

1. Write a new test case (based on feature intended to be implemented)
2. Write new code until the all the test cases pass (i.e., new test case(s) and previous test cases), in this step, code quality is not a concern, thus as summarized by the

author of TDD, Kent Beck, *commit any necessary sins*.

3. Refactor the newly amended code

Therefore, test suites and code are developed parallelly, unlike the traditional approach where test suites are only being written after the coding part is completed. The advantages of TDD is that developer will have less fear to modify previous code, because the modification breaks the application, they can be signalled via running previous test case.

6.2.1.5 Test case structure

Since Keli is a compiler, rather than conventional application that relies on databases and user interface, constructing test suites is quite straightforward. Basically, each test cases will be structured as follows, which is inspired by the Glasgow Haskell Compiler (GHC) regression test suites structure (The Glasgow Haskell Compiler, 2019) :

1. The input file which is written in Keli
2. The expected output (i.e., what should be seen after performing the respective action on the Keli source file)
3. Metafiles (i.e., extra files needed based on different types of test cases).

To ease the process of adding, modifying and deleting test cases, a custom testing framework should be implemented based on the following specification:

1. Each test cases should resides in a folder named *specs*.
2. Each subfolders within *specs* represent the type of test cases, the necessary types are(note that extra types of test cases can be added if needed):
    1. *execute*(to execute a Keli source file and observe the output)
    2. *suggest*(to look search for suggestion on a Keli source file based on cursor position).
3. Within each subfolders, the following files should be existent (note that extra files can be added if required):
    1. *entry.keli* (which represent the input)
    2. *output* (which represent the expected output)

The internal implementation for running the test cases specified above is up to the implementor, as long as the structure aforementioned is observed.

6.2.1.6 Item Pass/Fail Criteria

A test case is considered to be passed if the actual output (after performing the respective actions) perfectly, and lexically tally with the expected output. Conversely, a test case is considered to be failed even there is only one character difference between the actual output and the expected output.

6.2.1.7 Suspension Criteria And Resumption Requirements

Since this test plan is meant for testing the functional specification of Keli, and at the same time TDD should be applied, there is no suspension criteria, i.e., testing should not be stopped at all throughout the development of Keli. Although there are no suspension criteria for running tests, there is one suspension criterion for amending new test cases, that is if any of the previous test cases are still failing, new test cases should be suspended from being added into the regression test suite; the resumption requirement for amending new test cases are: all the test cases had already passed.

6.2.1.8 Test Deliverables

The deliverables of this testing plan are listed as follows:

1. A test plan document (i.e., this document)
2. Test cases (at least one for each features to be tested)
3. Testing framework (according to Section 6)
4. Test reports (which should be generated by the framework)
5. Code coverage report

6.2.1.9 Environmental Needs

To run the test cases, the following components should be installed:

1. Stack Version 1.9.3 (a build tool for Haskell), which can be downloaded at https://docs.haskellstack.org/en/stable/README/

2. HSpec (a testing framework for Haskell) https://hspec.github.io/
3. UNIX-like operating system such as MacOS, Linux, etc. (not a must, but preferable due to their builtin Shell utilities)

6.2.1.10 Responsibilities

Since this is a one-man project at the moment, all responsibilities belongs to the author of this document (Wong Jia Hau).

6.2.1.11 Staffing And Training Needs

Since TDD is employed, meaning that the developers are also testers, the only required training is to learn how to use HSpec and the code coverage tools provided by Stack.

6.2.1.12 Schedule

The schedule of testing should be parallel with the development schedule, i.e., if 30% of the project is completed, 30% of the test suites should be completed at the same time.

6.2.1.13 Risks And Contingencies

The risk for executing this test plans are listed as follows:

1. Development delays, because the time required to fix individual test cases will grow exponentially as the test suites grow bigger, thus development progress maybe stunted.
    1. If this risk materializes, the suspension criteria and resumption criteria as defined in Section 6.2.1.7 should be revised.
2. Sudden change of requirements due to late detection of design errors.
    1. If this risk materializes, test cases that correlates with removed features should be removed from the test suites, however test cases that are associated with modified features should remain and be modified to suits the new specification.

6.2.1.14 Approvals

The personnels who can approve and evaluate the execution of this test plan are the project manager and the quality assurance manager. In the current stage of development, the author of this document bears all the aforesaid roles, thus he should be the one responsible for approvals.

6.2.1.15 References

Beck, K., 2003. *Test-driven development: by example*. Addison-Wesley Professional.

The Glasgow Haskell Compiler, 2019. *Adding new test cases*. [online] Available at: https://ghc.haskell.org/trac/ghc/wiki/Building/RunningTests/Adding [Accessed 14 March 2019].

6.2.2 Functional Specification Test Result

6.2.2.1 Test output

The following is the test output that is displayed on the terminal.

```
~
suggest-object-alias-1
~
suggest-for-lambda-shorthand-2
~
suggest-for-function-chaining-1
~
suggest-tagged-union-1
~
suggest-for-function-chaining-2
~
suggest-imported-symbol-1
~
suggest-for-lambda-shorthand-1
```

~
suggest-case-expr-1

Finished in 0.0099 seconds
8 examples, 0 failures

~
record-as-type-annot-2
~
record-value-setter-1
~
ffi-javascript-1
~
identity-func-1
~
generic-tagged-union-5
~
insensitive-decl-order-1
~
generic-tagged-union-3
~
tostring-1
~
carryless-tag-1
~
lambda-shorthand-1
~
fix#33
~
complete-tag-matcher-1
~
@generic-tagged-union-type-mismatch-1
~
mutual-recursion-1
~
@type-mismatch-1
~

module-import-4

~

@tag-matcher-duplicate-tags-1

~

@incorrect-carry-type-1

~

module-import-3

~

record-getter-1

~

@tag-matcher-hetero-branch-1

~

func-chaining-2

~

@importing-unexisting-file-1

~

docstring-1

~

@record-prop-type-mismatch-1

~

recursive-tagged-union-1

~

func-without-return-type-annot-1

~

generic-tagged-union-1

~

recursive-func-1

~

multiple-dispatch-1

~

ffi-javascript-2

~

@tag-matcher-missing-branch-1

~

func-chaining-1

~

access-tag-carry-1

~
bifunc-1
~
module-import-2
~
@duplicated-func-1
~
multiline-string-1
~
record-type-alias-1
~
tagged-union-as-type-annot-1
~
@tag-invalid-carry-type-1
~
record-as-type-annot-1
~
@record-missing-prop-1
~
array-literal-1
~
fix#41
~
access-tag-carry-2
~
comment-1
~
record-creation-1
~
@incorrect-lambda-param-type-1
~
carryful-tag-1
~
trifunc-1
~
@duplicated-const-id-1
~

record-lambda-setter-1
~
module-import-1
~
generic-tagged-union-4
~
generic-tagged-union-2
~
multiple-dispatch-2
~
else-tag-matcher-1
~
@tag-matcher-using-unknown-tag-1

Finished in 2.6415 seconds
59 examples, 0 failures

match
got duplicate
zero intersection
got excessive
missing
perfect match
eli parser
identifiers
string expr
lambda expr
multiple decl
const decl
monofunc decl
polyfunc decl
monofunc call
polyfunc call

Finished in 0.0024 seconds
14 examples, 0 failures

6.2.2.2 Code Coverage Report

From the following figure, we can see that 48% code coverage is achieved, it means that almost half of the codebase is covered by automated test cases.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Analyzer	100%	9/9	48%	38/78	60%	333/554
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Ast.Raw	9%	2/21	20%	2/10	8%	2/23
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Ast.Verified	20%	8/40	16%	11/65	26%	52/195
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Compiler	50%	2/4	70%	7/10	70%	78/110
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/CompletionItems	60%	12/20	63%	33/52	65%	358/545
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Diagnostics	25%	9/36	3%	3/100	8%	58/658
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Env	66%	4/6	-	0/0	66%	47/71
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Interpreter	50%	1/2	50%	1/2	51%	20/39
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Lexer	73%	11/15	-	0/0	85%	51/60
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Module	22%	2/9	-	0/0	18%	2/11
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Parser	94%	33/35	57%	8/14	87%	488/559
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/PreludeJSCode	0%	0/1	-	0/0	0%	0/1
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/StaticError	0%	0/2	-	0/0	0%	0/7
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Transpiler	0%	0/15	0%	0/44	0%	0/476
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/TypeCheck	76%	20/26	44%	87/194	51%	783/1523
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Unify	100%	9/9	36%	17/46	45%	111/244
module keli-compiler-0.1.0.0-IKN4USh2d2YLgjcT2airmw/Util	20%	1/5	18%	3/16	27%	26/96
Program Coverage Total	48%	123/255	33%	210/631	46%	2409/5172

Figure 143: Code coverage report

6.3 Usability Test

6.3.1 Usability Test Plan

6.3.1.1 Introduction

The document is structured according to the IEEE 829 Test Plan Template. The objective of this test plan is to describe the required action for carrying out a simplistic usability test to evaluate the usability of the Keli language. This document does not aims to be a professional usability test plan, but a conventional one which will allow the implementer to update minor features of Keli based on the feedback acquired from this test.

6.3.1.2 Features To Be Tested

1. The Keli Extension
    1. Intellisense
        1. Function invocations
        2. Tag matcher expressions
    2. Error reporting
    3. Direct code evaluation
2. Language features
    1. Function declaration
    2. Function invocation
    3. Object alias declaration
    4. Object construction
    5. Tagged union declaration
    6. Tag construction

6.3.1.3 Features Not To Be Tested

The following features should not be tested, unless the test subject display a strong technicality in programming.

1. Module system
2. Generic functions

3. Generic tagged union
4. Function recursion

6.3.1.4 Approach

A. Approach summary

The testing should be done by conducting a one-to-one review session with a few participants.

B. Participants requirements

Each participant must fulfils all of the following criteria:

1. Must be a student of software engineering
2. Must have at least one year of programming experience

C. Minimum number of participants required.

At least 5 participants should participate in this testing session.

D. Testing procedure

1. A one-to-one testing session that will last for 30 to 45 minutes should be scheduled with the desired participant.
2. During the testing session, the required IDE to type and run Keli program (i.e., Visual Studio Code) and an empty Keli source files with the necessary imports must be opened.
3. Then. the participant should be guided on how to create simple values such as strings and number, and how to evaluate the Keli program
4. After that, the following Keli features should be explained and demonstrated to the participants (note that the order should not be changed):
    1. Function invocation (both single-parametered and multi-parametered)
    2. Object literal
    3. Object type alias

4. Array literal
5. List functions
6. Tagged union creation
7. Tag construction
8. Function declaration (both single-parametered and multi-parametered)

5. For each feature explained above, the following tasks should be completed by the participant with minimum help from the host (variation is allowed):
    1. Invoke basic arithmetic functions on numbers (e.g. addition, subtraction, interval check etc.)
    2. Create a simple object such as a person, which has a name property and age property
    3. Create an object type alias for person
    4. Create a simple array with a few numbers
    5. Invoke list functions such as *select*, *where* and etc.
    6. Create a tagged union to represent Shape, which consists of Square, Circle and Rectangle
    7. Construct a new Shape value
    8. Create a function to calculate the area of a Shape
6. During the time where the participants is trying to complete the given task, the difficulties faced by them should be noted down, and appropriate questions should be raised at the same time to obtain relevant reasons.
7. At the end of the session, a feedback form must be completed by the participant.

6.3.1.5 Test Deliverables

An evaluation report should be compiled based on the feedback acquired from participants. The report should include which features are considered difficult by participants, and a list of corrective design that can be applied to enhance the syntax of the Keli language, and the usability of the Intellisense.

6.3.2 Usability Test Result

6.3.2.1 Preface

This section is compiled based on Appendix C and Appendix D.

6.3.2.2 Results & Discussions

A. Quantitative Questions

Question	Average Score	Max Score
Do you think Keli code is easy to read?	8.33	10
Do you think Keli code is easy to write?	7.33	10
How user-friendly do you think Keli is?	7.17	10
If you're given a choice, would you like to use Keli in your future project?	7.5	10
Are you satisfied with the IntelliSense?	8.17	10

Based on the table above, we can see that most respondents feel that Keli code is easy to read and easy to write, however, it also seems that Keli is easier to be read than it is to be written. Moreover, it seems like most of them would also like to use Keli for their future software projects. Also, most of them also thinks that the IntelliSense for Keli is quite good.

B. Qualitative Questions

For qualitative questions, not much insights can be harnessed as the number of respondents are too few. However, there are a few common responses that might be useful for the future development of Keli:

1. Lacking of packages
2. Ecosystem compatibility with currently popular languages such as the Java ecosystem
3. The differences of paradigm with other conventional languages needs to be clarified upfront

CHAPTER 7

CONCLUSION & RECOMMENDATION

I had design and implemented a programming language named Keli to address the problem statements aforementioned in Section 1.2. Although Keli is not a full-fledge language at the moment, and is lacking library support, but from the usability test evaluation, it can be concluded that Keli is better in terms of maintainability, as it is highly readable compared to other mainstream programming languages.

At the moment, the design space explored is only about the syntax and conceptual design of a programming language, in the future I hope to explore more in the space of tools design, i.e., how can I design a development tools that can incorporates with Keli to provide an unprecedented user experience for programmers.

For future readers who also wants to design and implement a programming language, I would strongly recommend you to read the Dragon Book, *Compilers: Principles, Techniques, and Tools*. Not only that, you should also join the programming language Reddit community at https://www.reddit.com/r/ProgrammingLanguages. Lastly, for development tools, I strongly recommend using Haskell, as Haskell itself is a favourite playground for programming language researchers to try out their idea, that's why Haskell actually have a lot of fairly advance features, as most of them are rooted from type theory. In short, I would say that Haskell only have advance features that is useful, which is a very good thing, because in conventional programming languages, you have too many ways to achieve a single objective, and none of them leads to a good endeavour anyhow.

In a nutshell, I want to say that designing and implementing a programming language is one of the most challenging and fun project that can be done in the realm of software engineering. Also, I believe that developers who had written their own compiler is also highly respected in the programming field. Let me finish with a quote from Thomas Edison:

"Genius is one percent inspiration and 99 percent perspiration.".

REFERENCES

Aho, A.V., 2012. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India.

Anderson, A., Huttenlocher, D., Kleinberg, J. and Leskovec, J., 2012, August. Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 850-858). ACM.

Anderson, O. 2009. *LLVM Tutorial 1: A First Function*. [ONLINE] Available at: . [Accessed 13 August 2018]

Beck, K., 2003. *Test-driven development: by example*. Addison-Wesley Professional.

Bloch, J., 2017. *Effective java*. Addison-Wesley Professional..

Chai. 2017. *Chai Assertion Library*. [ONLINE] Available at: http://www.chaijs.com/. [Accessed 28 June 2018].

Coding Tech, 2018. *Procedural Programming: It's Back? It Never Went Away*. [Online Video]. 13 January 2018. Available from: . [Accessed: 11 July 2018].

Computer Hope, 2017. *What is Machine Language?*. [ONLINE] Available at: . [Accessed 11 July 2018].

Danielsson, N.A. and Norell, U., 2008. September. Parsing mixfix operators. In *Symposium on Implementation and Application of Functional Languages* (pp. 80-99). Springer, Berlin, Heidelberg.

Elixir. 2018. *Protocols - Elixir*. [ONLINE] Available at: . [Accessed 17 July 2018].

Elliot, E., 2017. *Higher Order Functions (Composing Software) – JavaScript Scene – Medium*. [ONLINE] Available at: . [Accessed 12 July 2018].

Encyclopedia Britannica. 2018. *Inversion | literature | Britannica.com*. [ONLINE] Available at: https://www.britannica.com/art/inversion-literature. [Accessed 28 June 2018].

ESLint - Pluggable JavaScript linter. 2018a. *no-implicit-coercion - Rules* [ONLINE] Available at: . [Accessed 03 July 2018].

ESLint - Pluggable JavaScript linter. 2018b. *prefer-const - Rules* [ONLINE] Available at: . [Accessed 03 July 2018].

Fritzson, P., Privitzer, P., Sjölund, M. and Pop, A., 2009, December. Towards a text generation template language for Modelica. In *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009* (No. 043, pp. 193-207). Linköping University Electronic Press.

Fowler, M. 2018. *Refactoring: Improving the design of existing code*. S.l.: Addison-Wesley.

Galorath. 2017. Accurately Estimate Your Software Maintenance Costs. [ONLINE] Available at: http://galorath.com/software_maintenance_cost. [Accessed 23 June 2018].

George, B. and Williams, L., 2004. A structured experiment of test-driven development. *Information and software Technology*, *46*(5), pp.337-342.

Glass, R.L., 2001. Frequently forgotten fundamental facts about software engineering. *IEEE software*, *18*(3), pp.112-111.

GNU Project - Free Software Foundation. 2014. *Bison - GNU Project - Free Software Foundation*. [ONLINE] Available at: . [Accessed 23 July 2018].

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M.K.C. and Cappos, J., 2017, August. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*(pp. 129-139). ACM.

Grudin, J. and Norman, D.A., 1991, August. Language evolution and human-computer interaction. In *Proceedings of the thirteenth annual conference of the Cognitive Science Society* (Vol. 611, p. 616).

HaskellWiki. 2017. *Monad - HaskellWiki*. [ONLINE] Available at: . [Accessed 12 July 2018].

HaskellWiki. 2009. *Referential transparency - HaskellWiki*. [ONLINE] Available at: . [Accessed 12 July 2018].

Hindley, R., 1969. The principle type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, *146*, pp.29-60.

Hutchins, E.L., Hollan, J.D. and Norman, D.A., 1985. Direct manipulation interfaces. *Human-computer interaction*, *1*(4), pp.311-338.

Immutable.js. 2018. *Immutable.js*. [ONLINE] Available at: . [Accessed 15 August 2018].

Interaction Design Foundation. 2018. *Shneiderman's Eight Golden Rules Will Help You Design Better Interfaces*. [ONLINE] Available at: https://www.interaction-design.org/literature/article/shneiderman-s-eight-golden-rules-will-help-you-design-better-interfaces. [Accessed 15 July 2018].

Kay, A., 1998. *Introducing more meaning into inheritance* . [ONLINE] Available at: .

[Accessed 17 July 2018].

Lambert, K.A. and Louden, K.C., 2011. Programming Languages: Principles and Practices.

Lämmel, R. and Ostermann, K., 2006, October. Software extension and integration with type classes. In *Proceedings of the 5th international conference on Generative programming and component engineering* (pp. 161-170). ACM.

Lattner, C. and Adve, V., 2004, March. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (p. 75). IEEE Computer Society.

London Haskell. (2012). *Why Do Monads Matter?*. [Online Video]. 2 November 2012. Available from: . [Accessed: 13 July 2018].

Wei, Z., 2011, August. J*ava's Interface and Haskell's type class: differences and similarities?* [ONLINE] Available at: [Accessed 16 June 2018].

Jarzabek, S., 2007. *Effective software maintenance and evolution: A reuse-based approach*. CRC Press. p.1

Jones, C. 2017. *Software Economics and Function Point Metrics: Thirty years of IFPUG Progress.* Available at: [Accessed 14 June 2018].

Lientz, B.P., Swanson, E.B. and Tompkins, G.E., 1978. Characteristics of application software maintenance. *Communications of the ACM*, *21*(6), pp.466-471.

Math StackExchange. 2013. *What is the summation notation for the Fibonacci numbers? - Mathematics Stack Exchange*. [ONLINE] Available at: . [Accessed 12 July 2018].

Martin, R.C., 1996. The open-closed principle. *More C++ gems*, *19*(96), p.9.

Martin, R. C., 2009. *Clean code: A handbook of agile software craftsmanship.* Upper Saddle River, NJ: Prentice-Hall.

Maximilien, E.M. and Williams, L., 2003, May. Assessing test-driven development at IBM. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 564-569). IEEE.

Meyers, S., 2005. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education.

MDN Web Docs. 2018. *Array.prototype.join() - JavaScript | MDN*. [ONLINE] Available at: . [Accessed 04 July 2018].

Microsoft, 2015. Extension Methods (C# Programming Guide) | Microsoft Docs. [ONLINE] Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods. [Accessed 02 July 2018].

Milner, R., 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences*, *17*(3), pp.348-375.

Pane, J.F., Myers, B.A. and Miller, L.B., 2002. Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on* (pp. 198-206). IEEE.

Pearse, T., and Oman, P., 1995, October. Maintainability measurements on industrial source code maintenance activities. In *Software Maintenance, 1995. Proceedings., International Conference on* (pp. 295-303). IEEE.

Perrin, C., 2010, October. *Is tmux the GNU Screen killer?* [ONLINE] Available at:

[Accessed 9 June 2018].

Pierce, B.C. and Benjamin, C., 2002. *Types and programming languages*. MIT press.

Pradhan, D., 2008. *Top 10 reasons why there are Bugs/Defects in Software!* [ONLINE] Available at: [Accessed 9 June 2018].

PromptWorks, 2018. *Leading Software Developer PromptWorks Interviewed by WeDoTDD.com*. [ONLINE] Available at: . [Accessed 07 July 2018].

Python Software Foundation, 2016. *What's New In Python 3.6*. [ONLINE] Available at: [Accessed 14 June 2018].

Python. 2018. *7.1. string — Common string operations — Python 2.7.15 documentation*. [ONLINE] Available at: . [Accessed 04 July 2018].

Scalabrino, S., Linares-Vásquez, M., Poshyvanyk, D. and Oliveto, R., 2016, May. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on* (pp. 1-10). IEEE.

Schneider, G.M. and Gersting, J., 2018. *Invitation to computer science*. Cengage Learning.

Sengstacke, P., 2016. *JavaScript Transpilers: What They Are & Why We Need Them — Scotch*. [ONLINE] Available at: . [Accessed 11 July 2018].

Shih, R., 2016. *Functional Computational Thinking — What is a monad?*. [ONLINE] Available at: . [Accessed 12 July 2018].

Software Engineering Stack Exchange. 2018. *functional programming - What Are The Uses of Algebraic Data Types?* [ONLINE] Available at: . [Accessed 31 July 2018].

Software Engineering StackExchange, 2013. *Do you actually write 'clean code'?* [ONLINE] Available at: [Accessed June 18, 2018]

Software Engineering Stack Exchange. 2018b. *Why is verbosity bad for a programming language?* [ONLINE] Available at: . [Accessed 31 July 2018].

Software Engineering Stack Exchange. 2010a. *language design - Should I use a parser generator or should I roll my own custom lexer and parser code?*. [ONLINE] Available at: . [Accessed 24 July 2018].

Software Engineering StackExchange, 2010b. *Why are PHP function signatures so inconsistent?* [ONLINE] Available at: [Accessed 16 June 2018].

Stack Overflow. 2018. *c++ - Advantages of Antlr (versus say, lex/yacc/bison)* [ONLINE] Available at: . [Accessed 24 July 2018].

Stack Overflow. 2009. *Are global variables bad?* [ONLINE] Available at: . [Accessed 10 July 2018].

Stack Overflow. 2010. *Explain to me what is a setter and getter* [ONLINE] Available at: https://stackoverflow.com/questions/2649096/explain-to-me-what-is-a-setter-and-getter. [Accessed 22 July 2018].

Stack Overflow. 2012. *Which part of Hindley-Milner do you not understand?* [ONLINE] Available at: [Accessed 13 August 2018]

Stack Overflow. 2017. *What is the worst real-world macros/pre-processor abuse you've ever come across?*. [ONLINE] Available at: . [Accessed 12 July 2018].

Strachey, C., 2000. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, *13*(1-2), pp.11-49.

Tashtoush, Y., Odat, Z., Alsmadi, I., and Yatim, M., 2013. Impact of programming features on code readability.

The Atlantic. 2015. *Star Wars: Linguists Explain the Way Yoda Speaks* . [ONLINE] Available at: https://www.theatlantic.com/entertainment/archive/2015/12/hmmmmm/420798/. [Accessed 28 June 2018].

TIOBE, 2018. *TIOBE Index*. [ONLINE] Available at: . [Accessed 24 June 2018].

University of Nottingham. (2015). *Computer Science - Brian Kernighan on successful language design*. [Online Video]. 17 November 2017. Available from: . [Accessed: 16 July 2018].

Vasilescu, B., Filkov, V. and Serebrenik, A., 2013, September. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Social computing (SocialCom), 2013 international conference on* (pp. 188-195). IEEE.

Vlissides, J., Helm, R., Johnson, R. and Gamma, E., 1995. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, *49*(120), p.11.

Volkmann, R. M., 2009. *Clojure - Functional Programming for the JVM :: OCI.* [ONLINE] Available at: https://objectcomputing.com/resources/publications/sett/march-2009-clojure-functional-programming-for-the-jvm/. [Accessed 30 June 2018].

Zipf, G. K., 2016. *Human behavior and the principle of least effort: An introduction to human ecology.* Ravenio Books.

APPENDICES

Appendix A: Questionnaire 1 Questions

Appendix B: Questionnaire 1 results

Appendix C: Questionnaire 2 questions

Appendix D: Questionnaire 2 results

Appendix E(1): Keli Language Specification

Appendix E(2): Abstract Syntax Tree Model

Appendix F: Github Issues