The Pennsylvania State University
The Graduate School
College of Information Sciences and Technology

PROGRAMMING IN ELIZA

A Thesis in
Information Sciences and Technology
by
Xiao Liu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

April 2016

I grant The Pennsylvania State University the non-exclusive right to use this work for the University's own purposes and to make single copies of the work available to the public on a not-for-profit basis if copies are not otherwise available.

Xiao Liu

The thesis of Xiao Liu was reviewed and approved* by the following:

Dinghao Wu, Ph.D.
College of Information Sciences and Technology
Thesis Advisor, Chair of Committee

Mary Beth Rosson, Ph.D.
College of Information Sciences and Technology
Committee Member

John Yen, Ph.D.
College of Information Sciences and Technology
Committee Member

*Signatures are on file in the Graduate School.

Abstract

According to the technical report by Microsoft 2012, there will be a shortage of graduates to fill available job positions in computer-related occupations till 2020. One of the possible reasons accounts for the shortage is the challenge of mastering the skill to code; the rigid and abstruse syntax of programming languages is the first barrier. Out of creating an easier environment to program, this thesis proposes a system that allows users to program in natural language, which enhances the coding experience for novices as well as experienced programmers.

Our prototype system, called PiE (Programming in Eliza), is based on Eliza, an early primitive AI prototype. The original Eliza was designed to be a psychotherapist. We make it a programming robot. Using a rule-based method, PiE interprets the natural commands into a universal intermediate language (PiE script) and programs in the target syntax will be synthesized according to the semantics we extract with the rules. To demonstrate the PiE system, we create PiE-LOGO, which synthesizes programs in the LOGO programming language that helps people to draw graphs. Our experimental results show that, on average, the success ratio is 88.4% for synthesizing LOGO programs from simple conversations with Eliza. PiE also enables end-users with no experience to program in LOGO with a smoother learning curve. We also build it with a voice module, with which children can interact with PiE by voice before learning to type, and users can play with it on mobile devices more easily. We also provide an online adaptation with explanatory step-by-step tutorial through which feedbacks for further improvement are collected.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my adviser Dr. Dinghao Wu for his help and invaluable advice for my research and this thesis. I also want to thank my committee for their assistance and my fellow graduate students for all of their help and support. Finally, I am grateful to my family; their forbearance made the whole process possible.

Chapter 1 Introduction

1.1 Programming in a Natural Way

For a long time, programming remains a dark art, especially for beginners. With great enthusiasm, students start to learn some basic languages, e.g. JAVA or Python, in their first CS course, but easily get disappointed by the rigid syntax and semantics. Mastering a programming language is far more beyond their expectation which is one of the reasons that a great number of students wandering around this area. According to the annual technical report from Microsoft 2012, there will be a shortage of graduates to fill all of the available job opening in computer-related occupations in 2020 with additional 1.2 million openings in computing professions that require at least a bachelor's degree (Microsoft, 2012). Under this circumstance, initiatives such "Teach the Hour of Code" are advocated for every student to try coding for one hour (Code.org, 2014). It is of critical importance to ensure beginners with a positive learning experience in order to attract them into and retain them in the computing and information field.

Out of creating a tool that alleviating the students anxiety and enhancing their experience to program, we want to build a system that synthesizes programs for users automatically based on their natural language descriptions. In earlier studies, researchers propose the automatic programming based on the mechanical theorem-proving techniques (Waldinger & Lee, 1969). However, this research problem is much broader than expected as it is difficult to prove the theorems involving existential quantifiers. In this case, some constraints have been appended to the automatic programming, which is gradually considered as program synthesis (Manna & Waldinger, 1980). Program synthesis is more likely a concept that defines the automation of programs in some specific do-

mains, such as robotics (Kress-Gazit, Wongpiromsarn, & Topcu, 2011; Maly, Lahijanian, Kavraki, Kress-Gazit, & Vardi, 2013), and spreadsheet programming (Gulwani, 2011).

Because of the versatility and applicability, natural language programming has been widely discussed as an easy way for novices to learn how to program (Ballard & Biermann, 1979; Dijkstra, 1979; Biermann, Ballard, & Sigmon, 1983). However, the realization of natural language programming is based on "AI Complete" which means that the machine is required to understand every natural language description. As expected, this goal has not been achieved yet, but with advanced developments in natural language processing, it is currently feasible for machines to partially understand people (Lieberman & Liu, 2006), especially in a specific domain. Thus, in a specific domain, programming in natural language is viable.

Additionally, smart devices perform better with natural language as the input. Voice input has indeed advanced gradually in recent years. However, the accuracy of recognition for programming languages is still low (Begel, 2005). For example, "for int i equals zero i less than ten i plus plus" which should be translated into "for (int $i = 0$; $i < 10$; $i++$)" is sometimes recognized as "4 int eye equals 0 aye less then ten i plus plus". In this case, when we target at coding with smart devices, the accuracy of voice recognition of programming language cannot meet the requirements. But things may change when you say "Let's start a loop with integer i from 0 to 9, add 1 in each turn". The recognition accuracy can be much higher.

To lower the entrance bar for fresh students to learn programming, we propose a domain-specific program synthesis system called Programming in Eliza (PiE). PiE interactively takes natural language conversations from users, and synthesizes programs in target syntax. PiE-LOGO is our first implementation, with which programs in the LOGO syntax are generated.

1.2 Eliza: a Chatbot

Eliza, a primitive prototype of natural language processing, plays the role of a psychotherapist to communicate with patients (Weizenbaum, 1966). The input sentences are processed with a pre-defined script, where there are two basic types of rules: the decomposition rules and reassemble rules. Decomposition rules consist of different combinations of keywords, and for each decomposition rule there are a couple of reassemble rules corresponding to it. When a sentence is typed in, it will be decomposed into pieces

English	LOGO Commands
Draw with a Black Pen	COLOR BLACK
DO this 4 Times:	REPEAT 4 [
Move Forward 10 Paces, Drawing	FORWARD 10
Turn Right 90 Degrees	RIGHT 90]

Table 1.1. Web turtle tutorial example: draw a square

according to the decomposition rules and then based on one of the reassemble rules, a response in natural language will be generated automatically. Following is an example of how Eliza works:

Input	It seems that you hate me.
Decomposition Rule	(Any Words) (you) (Any Words) (me).
Decomposition	(1)It seems that (2)you (3)hate (4)me.
Reassemble Rule	(What makes you think I) (3) (you).
Output	What makes you think I hate you?

Although Eliza belongs to the first-generation natural language processing techniques using a rule-based method to understand users, it works quite well in specific domains. In this paper, we extend Eliza for a novel application: program synthesis in the LOGO programming language.

1.3 LOGO: a Graphic-oriented Programming Language

LOGO is a graphic-oriented educational programming language (Feurzeig & Papert, 1967). The well-known application of LOGO is the Turtle Graphics (Feurzeig & Papert, 1967), in which there is a turtle on the screen and commands from users will move the turtle in various ways and the trace left will be a specially designed graph. For example, if a child wants the turtle to move forward 10 steps, she may use the command *FORWARD 10*. Table 1.1 shows the program with four lines of commands that can draw a square with the Turtle.

In this paper, we aim at building a system to automatically generate programs in the LOGO programming language. To demonstrate the idea, we have implemented a prototype called PiE (Programming in Eliza) with Python Turtle to synthesize LOGO programs from natural language conversations with Eliza.

1.4 System Overview

Figure 1.1 illustrates the working flow of the PiE system. The system consists of three parts: Eliza, PiE script and LOGO. The core lies in the PiE script which can be seen as a connector between the other two. This script processes the natural language descriptions from users and synthesizes programs in the LOGO programming language which will be executed by the LOGO module. Meanwhile, it provides a feedback in natural language to users via the Eliza module.

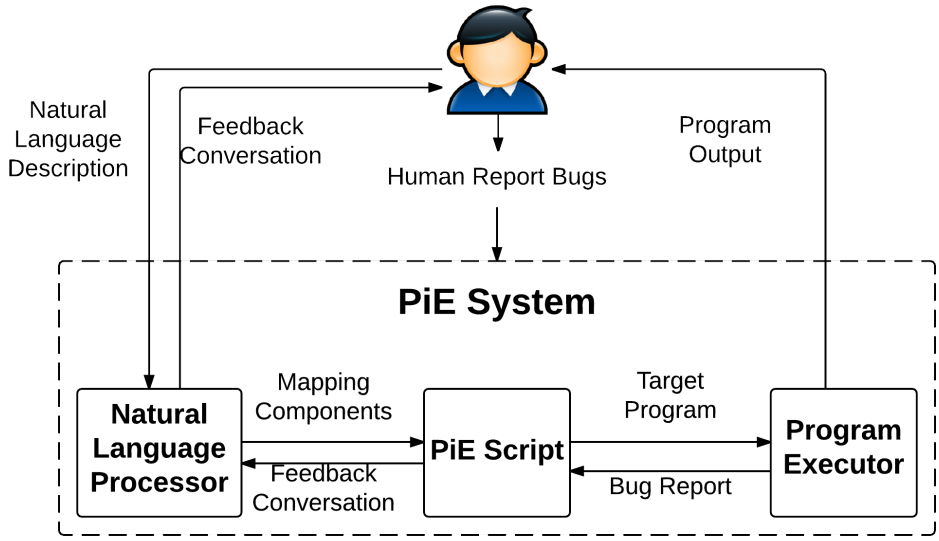

Figure 1.1. Flowchart of PiE

1.5 Contributions

With the proposed system, we make the following contributions:

- We propose a novel way for domain-specific program synthesis based on Eliza and recall the importance of natural language programming with gradually mature techniques in NLP and easy access to programming devices for end-users.
- We have realized program synthesis in the LOGO programming language from English conversations between users and computers. Programs can be synthesized with few constraints on the input natural language commands.

- We have achieved a preliminary step in natural language programming for education use. Program logic can be learned by end-users with no experience during the interaction with the PiE system, regardless of using complicated programming languages like C++ or Java.

The preliminary result has been published as a paper "PiE: Programming in Eliza" (Liu & Wu, 2014) in the *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*.

1.6 Outline of Thesis

The rest of this thesis is structured as follows. In Chapter 2, we first summarize the background of the PiE system with related research results. In Chapter 3, we use an example to illustrate a complete working flow of the PiE system. Meanwhile, we introduce the PiE script used in this specific domain, and elaborate the key techniques for program synthesis in our system. In Chapter 4, we extend the system with voice module and explain the online adaptation of our original system. Chapter 5 shows the experiments and results for system evaluation and we conclude and suggest for future research in Chapter 6.

Chapter 2
Related Works

2.1 End-user Programming

Stemmed from a seed in early 1990s, researchers from a few institutes began to think out how to make it easier for people to program and learn programming. Allen Cypher et al. (Cypher & Halbert, 1993) first proposed the programming by demonstraten, taking advantage of which simple programs can be generated according to users' demonstratens. Myers (Myers, 1998) wrote a proposal called "Natural Programming". In his proposal, the idea of creating a better programming interface for non-professional users thrives. Later on, the field has been expanded to end-user development environments and the very popular ones included spreadsheet development, web authoring tools, and graphical languages, etc. The strongest motivations come from education and enabling end-user to develop software.

2.1.1 Education Use

One of the most popular applications for end-user software engineering is for educational use. Researchers have tried a variety of methods to make programming concepts more accessible for novices. They either make the programming language itself more natural or make the interaction between the system and people more fun. Alice (Conway, Audia, Burnette, Cosgrove, & Christiansen, 2000), a 3-D Interactive Graphics Programming Environment was brought about by Cooper et. al to make it easier for novices to develop interesting 3-D environments and to explore the new medium of interactive 3-D graphics. Taking advantage of this environment, students are able to see how their animated programs run. The highly visual feedback allows them to relate the program "piece"

to the animation action. However, the language syntax it adopts is not as natural but still strict; that is to say, students should still follow the manual book very carefully in case of any warnings or errors. Scratch (Resnick et al., 2009), a visual programming environment, is another representation for software engineering education. It is a platform that allows users to learn through exploraten and peer sharing. However, Scratch does not achieve all the goals that initially set: to introduce programming to those with no previous programming experience. To be more specific, it requires students some pre-knowledge of programming concepts, like condition, loop, before creating a Scratch project, but does not teach them along the practice. We believe a better end-user software engineering tool for education is expected to be natural in syntax, and interactive in the process. Here, "natural" means that syntax should be free enough that allows any sentences without ambiguity and the "interactivity" is supposed to be more heuristic for debugging, but not as plain as its current look. To have a clearer sense of existing methods for people to learn how to program, we summarized them in Table 2.1.

Tool/Platform	Interaction	Output
Alice[1] (Cooper, Dann, & Pausch, 2000)	Drag and Drop	3-D Story
Blockly[2] (Marron, Weiss, & Wiener, 2012)	Drag and Drop	Graphics
Cubiverse[3]	Javascript	Graphics
CodeMonkey[4] (Etemadi, Kharma, & Grogono, 2013)	CoffeeScript	Graphics
Looking Glass[5] (Powers, Ecott, & Hirshfield, 2007)	Drag and Drop	3-D Story
Logo[6] (Feurzeig & Papert, 1967)	LOGO syntax	Turtle Graphics
RoboMind[7] (Yuana, Faisal, Pangestu, & Putri, 2015)	Scripting	Robot simulation
KidSim[8] (Smith, Cypher, & Spohrer, 1994)	Natural Langugage	Graphics
Scratch[9] (Resnick et al., 2009)	Drag and Drop	Graphics

Table 2.1. Summary of educational programming tools/platforms

[1] http://www.alice.org/index.php
[2] https://code.google.com/p/blockly/
[3] http://wiki.cubiverse.net/?title=Main_Page
[4] https://www.playcodemonkey.com/
[5] http://lookingglass.wustl.edu/
[6] https://turtleacademy.com/
[7] http://www.robomind.net/en/index.html
[8] http://www.robomind.net/en/index.html
[9] https://scratch.mit.edu/

2.1.2 Enabling Tool

Other than educational use, researchers are seeking for tools that enable non-professionals to program in a few common development environments. Spreadsheet is one of the most popular end-user development environments. Gulwani et al. developed Flashfill that synthesizes string manipulation programs from input-output examples. From the end-user developers' perspective, programs for specific use is generated with a few input and output pairs that they provide and it is quite convenient when the operatens are complicated. Similar research was conducted on the synthesis of different programming languages, including string manipulation (Gulwani, 2011), smart phone automation scripts (Le, Gulwani, & Su, 2013), geometry constructions (Gulwani, Korthikanti, & Tiwari, 2011), web data extraction queries (Polozov & Gulwani, 2014), and programming assignment feedback scripts (Singh, Gulwani, & Solar-Lezama, 2013). The core techniques behind all the research are the SAT/SMT constraint solver which is borrowed from formal method community (Gulwani, 2012) and some global search strategies. The name for the approach is called *Programming by Example* (PbE) and it is a branch of program synthesis. With tools like these, end-users are able to synthesize code for target platforms, but the capability in most cases is limited. The operatens synthesized in the program are restricted to a domain-specific set that was defined by the developer and the synthesis method is not wise enough to deal with ambiguities that exist in the examples. In addition, the challenge in end-user software engineering still remains with this mechanism: it does not provide people the chance to debug if the synthesized code has bugs. In their studies, what is the unprofessional user supposed to do when they find the synthesized code does not provide the expected results turns out to be the biggest issue.

2.2 End-user Software Engineering

We are introducing a new framework that not only assists people to program, but also makes the developing environment easy to learn, more efficient, and less error-prone in the software development. From a broader perspective, our research lies in the field of end-user software engineering, which not only concerns on how end-user development goes, but also focuses on the whole software maintenance cycle, including testing, debugging, etc.

One of the main tasks in software engineering is software debugging. Ko and Myers

created a tool called "Whyline" (Ko & Myers, 2008) to assist people to find bugs by asking why and why-not questions about program behaviors. The idea is simple: rather than requiring people to translate their questions to code queries, Whyline allows developers to choose a why-did or why-didn't question about program outputs and then Whyline generates an answer to the question using program analysis. This tool makes it possible for people who possess limited programming skills to learn how to find bugs even sooner than professionals with traditional programming environment. We can adopt a similar mechanism in our framework and make it consistent with the program synthesis module as stated.

The other track in software engineering is software testing. Groce et al. (Groce, Kulesza, Zhang, Shamasunder, & Burnett, 2014) brought forward an interactive testing framework for end-users to test the machine learning systems. They proposed and formally defined three test selection methods for machine learning domain which provide very good failure rates even for small test suites. With assistance of this tool, end-users with no expertise in software testing can perform such tasks which helps to lower the cost for testing engineer training. The method is heuristic and its application scope can be broadened if we can abstract a higher level methodology that can be applied to other domains.

2.3 Rule-based Artificial Intelligence

Since we are proposing a tool that can synthesize programs from natural language descriptions, the first task of the system is to understand users' descriptions. In recent studies, researchers are prone to adopt statistic-based methods when there are adequate data to train and test. However, because of the lack of raw data, we adopted the rule-based natural language processing method in our study. Rule-based systems are alternative to statistic-based ones, to store and manipulate knowledge. The rules are usually based on the linguistic theories. Rule-based method is more efficient in specific domains since it does not require many computational resources; and error analysis is easier to perform.

Many researchers utilized the rule-based methods for their study into the natural language. Bill adopted the rules for automatic tag speeches and it works as well as the stochastic tagger as proved by the author (Brill, 1992). Vlas developed a rule-based natural language technique for classification of open-source software according to the informal documents (Vlas & Robinson, 2011). Le et al. proposed a synthesis

method using the rule-based relation detection algorithm to map the natural language descriptions to the relationship between incidents for windows phone commands (Le et al., 2013). Compared with the statistical-based methods, the data “training" in the rule-based methods are often more efficient. In this thesis, the rule-based method as Eliza does to synthesize programs performs quite well in the specific domain of interest.

Chapter 3
PiE: Programming in Eliza

3.1 Example

In this section, we motivate our system with the same example from a web turtle tutorial (Kendrick, 1997) as described in Table 1.1. The objective is to generate LOGO programs when the input are sentences in natural language with few constraints.

As described in Table 1.1, the user would like to draw a square in black with the turtle, but it is her first time to program with LOGO. However, she knows that if she wants to draw a square in black, she needs to pick up a black pen to draw four straight lines and make a 90 degree turn after each line. Thus, if she is allowed to manipulate the turtle in a natural language, one possible description is:

1.Draw with a Black Pen
2.Do this 4 times:
>>1.Move Forward 10 Paces, Drawing
>>2.Turn Right 90 Degrees

Every time a command in natural language is received by PiE, it will be decomposed word by word into text chunks, denoted $w_1, w_2, ..., w_n$ [Step 1]. In the next step [Step 2], the lexical analysis will be conducted with the assistance of the PiE script. Each of these text chunks will be tagged with a pre-defined *Token* in the PiE script, for example, the *Predicate Token* or the *Number Token*, by adopting *regular expression matching*. Then, the analyzed sentence will be easily parsed into a structure. For instance, if all the descriptive sentences in the example are analyzed, the structure of each sentence will be like this:

Draw with a Black Pen
(PredicateTok)+(RedundantTok)+(NumberTok)
Do this 4 times:
(PredicateTok)+(RedundantTok)+(NumberTok)+(KeywordTok)
Move Forward 10 Paces, Drawing
(RedundantTok)+(PredicateTok)+(NumberTok)+(RedundantTok)
Turn Right 90 Degrees
(PredicateTok)+(KeywordTok)+(NumberTok)+(RedundantTok)

In the following step [Step 3], the LOGO program will be synthesized based on the structure. It is a many to one mapping from the natural language structures to a particular predicate in LOGO. For each *Predicate Token* , which can be treated as the key function words, we can select the class of mapping rules. Mapping rules in one class contain the same predicate.

However, there are more than one LOGO command for each predicate. To synthesize the correct LOGO command for the input natural language instruction, the structure of the sentence then plays a part. The last step [Step 4] for the LOGO program synthesis is the substitution of parameters in the incomplete programs from [Step 3]. In this step, incomplete programs will be parsed by rule sequence, and meanwhile, the parameters in these commands will be substitute with the tokens from the input sentences.

3.2 PiE Script

We have designed the PiE Script from an extensive study of the descriptions in natural language for each command in the Web Turtle from various online LOGO language tutorials. Note a script is a set of decomposition and reassemble rules defined in Eliza.

3.2.1 PiE-LOGO

PiE-LOGO is a domain-specific language that can be ported to other platforms via syntax-directed translation. PiE-LOGO maintains the context-free feature of the LOGO programming language. The syntax of PiE-LOGO is shown in Figure 3.1. In PiE-LOGO, we use S to denote a complete statement which consists of two parts: Predicate π and Parameter I. The Parameter can be numbers, directors or color names extracted from the natural language. It can be “Omit" when users forget to include some parameters in their descriptions. Such as “Draw a line" with no mention of the length or “Make a right turn"

Statement S	::=	$(\pi\|T)(I\|T)$
Parameter I	::=	number|direction|color
	|	Ω
Omission Ω	::=	number|direction|color
Predicate π	::=	Predicate (a_1, a_2,..., a_n)
Repeat R	::=	S
	|	foreach $x \in a$,
		do S_1; S_2;...; S_n; od
Transform T	::=	t_1; t_2; ...; t_n
t	::=	transform(a)|transform(I)
Argument a	::=	input(a_1, a_2,..., a_n)

Figure 3.1. The syntax of PiE-LOGO

without a concrete degree. The Transformer T, a part of the Eliza mechanism, transforms predicates or parameters with the same meaning into a regular expression. The Repeat R denotes loops in PiE-LOGO and a denotes input arguments.

The following examples describe the natural language commands and their corresponding synthesized scripts, according to which we can achieve a better understanding of the PiE script that we defined.

3.2.1.1 Statement

[Color of the Pen]
>Use a blue pen!
S := pencolor $blue$
transform($blue$) := [0,0,255]

3.2.1.2 Function

[Define Functions]
>Go forward 200 steps, turtle!
>Then turn left.
>Let STEP1 include the last two commands!
S_1 := forward $number$
$number$:= 200
S_2 := turn $direction$
$direction$:= left
$STEP$:= last[$number$]

$number$:= 2
$last[2]$:= foreach n in $number$, $STEP$ = $STEP1$ + $last[n]$

3.2.1.3 Repeat

[Start Loops]
>repeat STEP1 four times
S := foreach n in $number$, do $STEP1$
$number$:= 4

3.2.2 Key Components

Normally, in each statement, there are three key components: Predicate, Parameter, and Transform.

Predicate: The predicate is one of the most important components in the script and it represents the predicates in the LOGO programming language. These predicates correspond with verbs in natural language commands, for instance,"Move ahead" will be translated to "FORWARD"; "Let the color of the pen be" will be translated to "PENCOLOR". These predicates in the PiE script are later mapped into the operators in the target languages using pre-defined rules.

Parameter: The parameter represents the object of the related predicate or the status of the object. For example, "100" in "FORWARD 100"; or "blue" in "PENCOLOR blue". Parameters are identified using regular expression matching.

Transform: This component comes from the Eliza mechanism as it categorizes predicates or parameters with the same meaning into a dedicated one. For example, "Move ahead" "Go forward" "Move on" will all be translated to "FORWARD". This component plays an important role in the semantic analysis of natural language commands.

3.3 Synthesis Tehniques

Our goal is to map commands in natural language into PiE-LOGO, which means (1) after the decomposition of each sentence in natural language, every chunk can be mapped into a token in our script; (2) when we parse the sequence of tokens, which is the result from the syntax analysis, PiE script is capable of handling all the possible syntax structures.

Token	Regular Expression
Number Token	(\d)*
Color Token	((\d)* \s){3}
Direction Token	(left|right)
Predicate Token	(forward|backward|repeat|...)

Table 3.1. Regular expression matching

Both tasks are quite complicated to be tackled. This section elaborates on the techniques adopted in the PiE system and the design of the PiE script.

3.3.1 Regular Expression Matching

To perform lexical analysis, we adopt *Regular Expression Matching*. Taking advantage of regular expressions, we are able to extract tokens like *Predicate Token* or *Parameter Token* in each command in English. A set of regular expressions are designed manually to match all the structures consist of tokens. In Eliza, the system recognize a sentence by using keywords, but our system adopts regular expressions. The regular expression matching in PiE assists the system to analyze the syntax of the natural language and then each chunk of the natural commands is mapped into a token in the PiE-LOGO. To some extent, keywords matching can be seen as a special regular expression matching; however, Eliza does not parse the input natural sentences into structures. Another advantage of regular expression matching in PiE is that, without a strictly designed ranking of the rules, the matching process can be completed in linear time with respect to the size of the input sentence. Some examples of regular expressions and their corresponding tokens are shown in Table 3.1.

3.3.2 Mapping Rules

We have designed a set of rules to map natural language commands into PiE-LOGO. As we described in the example, programs in PiE-LOGO can be synthesized based on the parsed structure.

We generate a group of natural language commands by ourselves for each predicate. Table 3.2 shows some specific examples of the natural commands that can be mapped into the “FORWARD” function class. As far as the contents in this table are considered, it indicates that there are many expressions in natural language which have the same semantic meaning because of synonyms. Different verbs in natural language are used in

Move forward 100 steps!
Please go forward 100 steps!
Can you go straight on for 100 steps!
Move ahead 100 steps little turtle!
Turtle, go ahead for 100 steps!
Move forth 100 steps!
Let's go forth 100 steps!
Move to the front 100 steps!
Go up for 100 steps!

Table 3.2. Natural command mapped to predicate "FORWARD"

these sentences but the structures are similar. Under this circumstance, we adopt transform T which is borrowed from the Eliza system to categorize predicates or parameters with the same meaning into a dedicated one. Take the "FORWARD" class as an example, *Move forward*, *Go forward*, *Move ahead*, *Go ahead*, *Go forth*, *Move to the front*, and *Go up* are of the same semantic meanings. Thus, they can be categorized into the same "FORWARD" class and they can be replaced by the predicate"forward".

Based on the 1987 Penguin edition of Rogers Thesaurus of English Words and Phrases (Roget, 1982), we maintain a dictionary of words that can be transformed to tackle the challenge from synonyms. The design enables PiE to process flexible natural language syntax.

3.3.2.1 Rule Ranking

Since the number of the rules for the mapping function is large, it is desirable to apply some ranking in the rule sequence to improve the effectiveness of the PiE system. We rank the rules in two steps: (1) by Predicate Order; (2) by Complex Order.

Predicate Order: Rules with the same predicate will be categorized into a class and then, we sort the classes according to the frequency of the predicate occurrence. We collected 484 real LOGO commands from 20 different LOGO programs among the most popular ones from Web-Turtle and count the frequencies of each predicate. For example, by frequency order, "forward" stays ahead of "pencolor", and thus, the rule "Move forward ($\backslash d$)* ($\backslash .$)*" stays before the rule "($\backslash .$)* color of the pen ($\backslash .$)*($\backslash .$)*".

Complexity Order: Rules in the same class are sorted by the complexity order. The rule with more tokens or Redundant Tokens are of more Complexity. For example, for the "FORWARD" class, "Move forward ($\backslash d$)* ($\backslash .$)*" stays before "(($\backslash .$)* forward ($\backslash d$)*

(\.*) forward a line of (\d*) (\.*) in length
(\.*) forward a line of (\d*) (\.*) long
(\.*) (\d*) (\.*) forward(\.*)
(\.*)forward(\.*) (\d*) (\.*)
(\.*) (\d*) forward(\.*)
(\.*) (\d*) (\.*) forward
(\.*)forward (\.*) (\d*)
(\.*)forward (\d*) (\.*)

Table 3.3. Mapping rules to the "FORWARD" class

(\.)*)". This ranking algorithm not only helps shorten the time, but also improves the accuracy of mapping.

3.3.2.2 Rule Prune

At the very beginning, we used the descriptions of online LOGO tutorials as the data set, together with our own experience, according to which we designed the first set of rules. However, there are many redundant rules in this set. For example, "Move forward (\d)* (\.)*" and "Go forward (\d)* (\.)*" are two rules at the very beginning. After the Rule Prune, these two can be merged into one: "(\.)* forward (\d)* (\.)*".

To prune the rules, we firstly sort the rules in a predicate class by *Complexity*. Here, the rules with more Tokens or Redundant Tokens are of more complexity. In this case, the "FORWARD" class is arranged as shown in Table 3.3.

Then, we test if there exists any rule of less complexity that can replace the one of more complexity. For example: any sentence that can be matched with the rule "(\.*) forward a line of (\d*) (\.*) in length" and "(\.*) forward a line of (\d*) (\.*) long" can also be matched with "(\.*)forward(\.*) (\d*) (\.*)". Since this meets the prune requirement, the former two can be pruned.

3.3.3 Rule Adaptation

There are in total 87 rules in the original library which are self- generated. However, to handle all the possible descriptions to manipulate the turtle, the library should be made adaptable. We realize this function by making the Transform Table adjustable. The system will collect the natural language descriptions that cannot be matched up with any rule. Based on these left-behind descriptions, new words will be appended to the Transform Table which possess equal meaning as the words in the original table. Thus,

we can make new rules as an extension to improve the success rate. Meanwhile, in addition to maintaining the rule library, users can add words in any predicate class via communicating with PiE's function definition. Thus, the rules in PiE are extensible.

In the future, with the increasing number of people that play with PiE, we would like to use crowd sourcing to improve the rule set. Whenever the system cannot understand the user, it will ask the user to say in another way. Typically, the user would change some words but not the whole sentence structure. Under this premise, we may find out the transformable tokens in our rules and adaptively complete the library using this crowd sourcing method. However, there is still a chance when the sentence structure is changed. To solve these problems, PiE will pop up with several words belong to different predicate classes. The user will be asked whether or not adding a new word to one of these classes.

3.3.4 Dialog Interaction

To make the interface more interesting, we also present a rule-based natural language response as the feedback to each input sentence. For each synthesized PiE-LOGO command, there is one and only natural language feedback corresponding to it. This feedback serves as the confirmation of program synthesized as well as the reminder of the mistakes, if any. In some cases, when PiE fails to respond to the user's command, the interaction system will request an alternative description.The interaction between users and PiE system makes this interface more user–friendly.

Chapter 4
Extensions

We extend the original PiE-LOGO system with voice input so that users can use our tool in a more natural way. Start with any requests, PiE responds to users in English which make the system accessible to even children who have not learnt typing or spelling. In addition, we believe that with the assistance of the voice module, the system can be implemented on mobile devices where typing is not as convenient as voice input. Thus, we build an adaptation of the original system with voice module. Meanwhile, we also develop an online version with tutorials and examples. In this section, we introduce the newly included voice module and the online version.

4.1 Voice Module

To add the voice function, two parts are built into the previous PiE system: the voice recognition and the text to speech as shown in Figure 4.1 . Originally, PiE takes in natural language in plain text; however, with the voice recognition function, voice can be first translated to plain text and then be processed in the same way. With the text to speech function, after the synthesis of natural language in plain text, voice is generated and output to users. By keeping the modulated core of the entire system, we can make the system more robust and scalable.

4.1.1 Voice Recognition

The voice recognition is performed with the Google speech recognition API which is widely used as it supports developments using various programming languages and it enjoys high reputation for its accuracy. Nevertheless, a problem with the recognition is

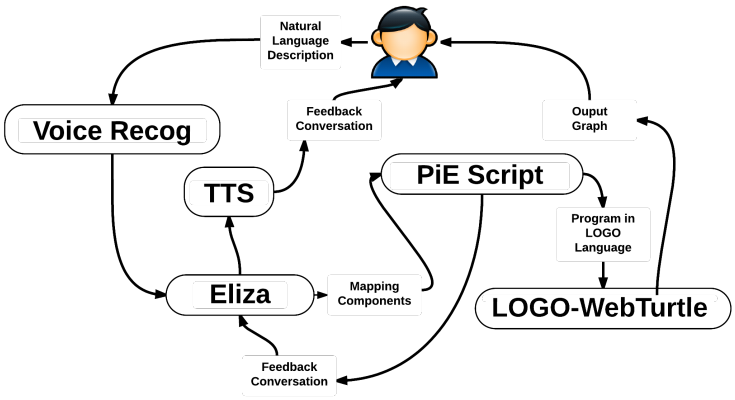

Figure 4.1. PiE with voice module

about the number generation. The return value of number recognitions are in the text format, e.g. 150 will be “one hundred and fifty", which brings difficulty to our original system design.

To overcome the challenge from number recognition, that is to translate the literal saying of each number into the Arabic numerals. To parse the text format of numbers, regular expression is utilized for token matching. We enumerate the basic unit tokens including one to nineteen, tens tokens including twenty to ninety and scale tokens including hundred to trillion, and build syntax tree for each number. The function interprets the literal number into digit.

4.1.2 Text to Speech

Original PiE outputs the plain text sentences to communicate with users. To enhance the experience when playing with PiE, we import the text to speech package to generate the speech and communicate with users in English from sentence in text format. With the new appended function, PiE is more like a programming assistant that communicates with users rather than a tool that just synthesizes programs with instructions.

4.1.3 Advantages and Defects

We incorporate the voice recognition and text to speech functions into our original PiE system to make it accessible for children use and mobile use. The extended part has brought a few benefits to the original system:

1. PiE demonstrate the idea that we can build a tool that translates humans'commands into machine instructions and Voice Module makes it possible that people can use it naturally and smoothly;

2. Voice module enlarge the audience of PiE, including the children who do not learn spelling and typing who are part of the main target people;

3. With the voice module, we can transport the PiE system to mobile devices where voice commands will occupy a dominant than typing input thanks to the efficient and convenient.

However, compared with the previous PiE, the voice module also brings some defects. Since the voice module is internet-based, the users must have access to the internet and the overall experience of using the system highly depends on the quality of the internet service.

4.2 Online Version

We also build an online version for attracting more audience. Figure 4.2 shows the interface of the online adaptation of PiE. To introduce PiE to users step by step without face-to-face demonstration, we write a tutorial which is included in the appendix. In addition, we also provide some examples to demonstrate PiE and one of which is shown in Figure 4.2. To further get users attracted and also to test the effectiveness of PiE, some challenges are set up after the examples to test how well people understand the basic concepts in programming. In this section, we show the detailed design of the online version.

4.2.1 User Interface

The user interface is separated into two main parts: left part consists of the main canvas and the dialog boxes; right part consists of the navigator and the instructions. Users can first select the tutorial from the navigator to go through the basics in PiE: to interact with PiE using the dialog boxes and see how the turtle moves on the canvas. After learning how to use PiE, users will go the examples to get some real practice and try some challenges that we designed.

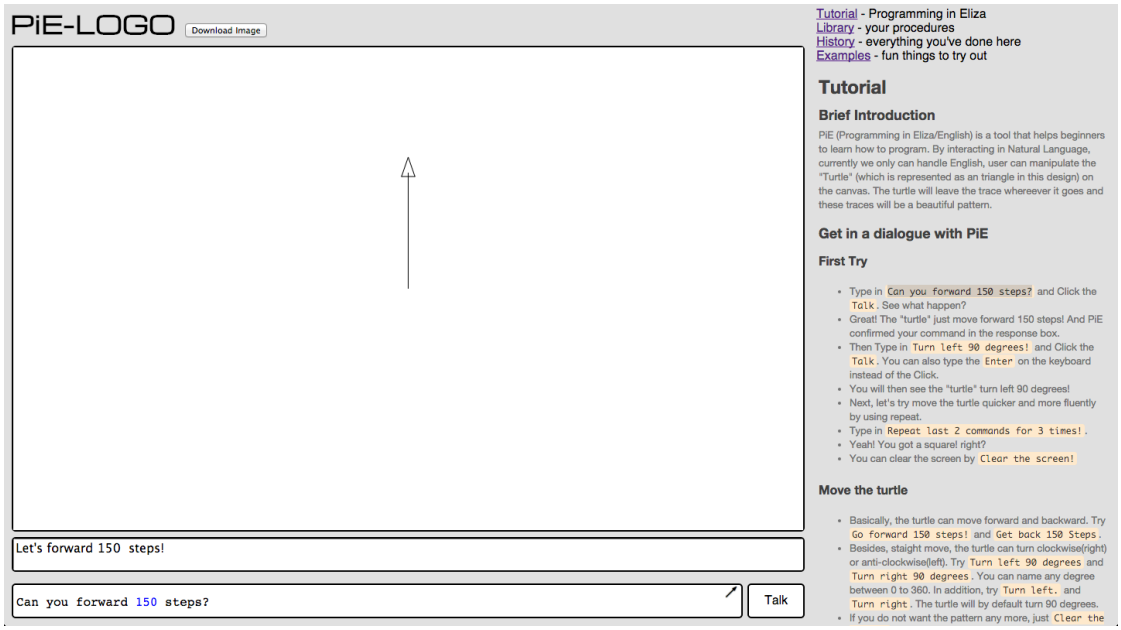

Figure 4.2. Online version of PiE

We keep the interface simple for users to play with and it highlights the main function of the turtle graphics, that is to draw graphs and meanwhile, it makes LOGO an interactive programming language. The concept of “What you see is what you get" is adopted in our design as it is intuitionistic for new programmers.

4.2.2 Tutorial and Examples

There are four sections in the tutorial. We begin with a brief introduction and let users have a first trial via a simple example: to draw a square. This is a common example in LOGO tutorials, but which is more intuitive when playing with PiE. Then we detailed the attributes of the trace that can be adjusted including the width, color, etc., after which users will know what can be done with PiE step by step. Although, users can draw with PiE easily with the current knowledge, they are not learning “real" programming. In the second section, we introduce the basic concept in programming, including, basic statements, looping, function and write scripts. With these advanced knowledge, users can draw more complex graphs with a few very efficient and powerful commands. The last section is to try some examples to review the knowledge that they just learned.

We show some very classic examples like to draw polygons and draw repeat shapes. After each set of example, users are required to take some challenge by revising the

code. Our previous design was to let users draw some shapes themselves, however, it is time-consuming for newbies and not efficient to test their understanding of the programming concept we introduced. Then, we changed the task to code revising, e.g., tell users how to draw a triangle, a square, a hexagon and ask them to draw an octagon. These tasks are designed for testing their understanding of loop and how to change the loop controls and the values in the loop body. We believe that with these examples and challenges, users will get a deeper understanding of PiE and the programming concepts.

Chapter 5
Evaluation

In this section, we first provide an demonstraten to show the process and result when playing with PiE. We then present evaluation in terms of (1) *Learning Efficiency* for non-programmers, novices and experienced users in learning to program with PiE-LOGO and (2) *Success rate* for synthesized programs. We have implemented our system using the Python Turtle, which is a standard library embedded in Python 2.7. Each command in PiE-LOGO is implemented in Python to move the turtle.

5.1 Demonstration

We choose a popular example: the Koch Curve among many latest drawings from users who draw with Turtle on the website, papertlogo in your browser (http://logo.twentygototen.org/). The input natural language is not case-sensitive and by using regular expression matching, the system can tolerate some spelling mistakes as well. This demonstraten is designed to show how complex tasks that PiE can handle. The program goes a little bit further than a beginner can understand, but with a designed algorithm in hand, she could make simple conversations with PiE and draw a Koch Curve without much difficulty.

The interaction between the user and the PiE system is shown in Figure 5.1 and the output graph is Figure 5.1. This demonstraten shows a normal pattern of using recursion in drawing. Consider the education use of the PiE system, when using this system to learn how to program, the non-experienced learners, especially for children, may experience a better interaction than coding directly on their own. The users enjoy the flexibility of natural languages without being required to memorize the strict and rigid syntax of programming languages.

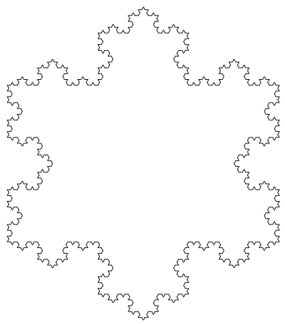

Figure 5.1. Output graph of demo

5.2 Learning Efficiency

Back to the original intention of the PiE system: in addition to the purpose to simplify the process of learning to code, we are aiming at describing a novel way for program synthesis. Nevertheless, there is no benchmark for evaluation of the easiness to master a programming language. We develop a notion called *Learning Efficiency* for learning how to program. Here, we define *Learning Efficiency* as, the extent to which time is well used for the intended programming task using a new programming language. We provide users equal time to learn both the LOGO programming language and PiE-LOGO and test the time they spent on a certain task. The less time they spent on programming, the higher *Learning Efficiency* of a specific language.

Six volunteers participated in our experiment. Among them, two are non-programmers (graduate students who have learned logic in the previous math studies), two are novices (first-year learners of the Java programming language) and the remaining two are experienced programmers (with four and five years of programming experiences, respectively). All of them are not involved in the system design and they have not learned the LOGO programming language before. Before the experiment, we start with six examples as the tutorial, which include simply drawing an “L-shape", defining a function, and drawing repeated patterns. It takes five minutes on average for them to go through the tutorial. Then, equal time is provided to go through the tutorial of the LOGO programming language provided by the online tutorial as introduced in Chapter 3.

After reviewing both the tutorials of our PiE system and the LOGO programming language, the six participants are asked to draw a target shape as shown in Figure 5.2. They need to draw a base unit recursively and the base unit is a diamond as shown in Figure 5.3. The edge length of the diamond is 100 steps and the two different angles are

User Descriptions	**PiE Responses**	**LOGO Program**
[PiE Start]	Thanks for trying PiE. How can I help you?	
> Let's first draw a line	Okay, what is the length of that line?	
> Oh, I mean, to define a function called line	I see, what are the arguments of the function Line?	
> They are count and n	Okay, I have designed a function called Line with input arguments called count and n. Then?	**to** line :**count** :**n**
> let the turtle move ahead n steps when the count equals to one	Okay, if the Count is 1, turtle will forward N steps	**ifelse** :**count** = 1 [**fd** :**n**]
> In other cases, do the following things:	I see. do what else?	
> let count decrease by one	Okay, Count equals to Count -1	**make** "**count** :**count** - 1
> repeat the line function with new count	I see. Function Line is called with the new argument count	**line** :**count** :**n**
> turn left 60 degrees	Okay, the turtle turns 60 degrees to the left	**lt** 60
> repeat the function line	Okay, Function Line is called	**line** :**count** :**n**
> turn right 120 degree and repeat function line	I see, the turtle turns 120 degrees to the right. Function Line is called.	**rt** 120 **line** :**count** :**n**
> Again turn but left 60 degrees and repeat the function line	Okay, the turtle turns 60 degrees to the left. Function Line is called	**lt** 60 **line** :**count** :**n**
> That is all for the function	I see, Function Line is end.	**end**
> Start another function called Koch	I see, what are the arguments of the function Koch?	
> still count and n	Okay, I have designed a function called Koch with input arguments called count and n. Then?	**to** koch :**count** :**n**
> first turn right 30 degrees and execute the function line	I see. The turtle turns 30 degrees to the right. Function Line is called.	**rt** 30 **line** :**count** :**n**
> turn right 120 degrees and execute the function line	I see. The turtle turns 120 degrees to the right. Function Line is called.	**rt** 120 **line** :**count** :**n**
> repeat the last command	I see. The turtle turns 120 degrees to the right. Function Line is called.	**rt** 30 **line** :**count** :**n**
> Let's call it Koch	I see, Function Koch is end.	**end**
> Clear everything	Okay, the canvas is clean.	clear
> let the turtle start from point (45,370)	I see, turtle is moved to point (45,370).	**setxy** 45 370
> Begin the function Koch	Okay, but what are the arguments?	
> count is 5 and n is 5	Okay, function Koch is called with the argument Count=5 and n=5.	**koch** 5 5
> Quit	Okay, Done!	Done

Table 5.1. Interaction with PiE for demo

of 60 and 120 degrees, respectively. The angle between the two dash lines is 10 degrees.

Given these indications on how to draw a graph in Figure 5.2, the testers are asked to use both PiE-LOGO and LOGO to complete the task. Testers are separated into two groups, one does PiE-LOGO first and the other does LOGO first. The average time consumed is shown in Table 5.2. PiE saves 43.8%, 21.8% and 22.4% of time, respectively, for non-programmers, novices and experienced programmers to draw the same graph.

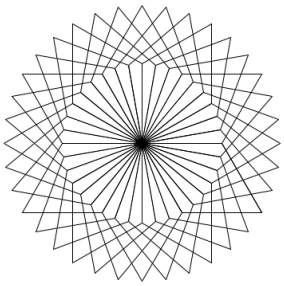

Figure 5.2. Learning efficiency test

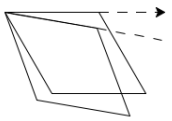

Figure 5.3. Base unit

	Experienced	**Novices**	**Non-programmer**
Original LOGO	67s	69s	130s
PiE	52s	54s	73s
Time Saving	22.39%	21.74%	43.85%

Table 5.2. Time used in the programming with LOGO and PiE

The preliminary results indicate that PiE creates a platform for them to program in a new language with a smoother learning curve. In addition, we may find that, PiE saves the most time for the non-programmers, in which case, we could say that PiE performs well as an introduction tutorial for those who want to learn basic programming skills.

Another interesting fact in the experiment is that, every experienced programmer checked the tutorial of the original LOGO during the programming task but that does not happen when using PiE. As a result, PiE relieves the burden of memorizing strict syntax and checking language references and tutorials to some extent.

5.3 Success Rate

We further evaluate the PiE system in terms of *Success rate* for the synthesized program. *Success Rate* is defined as the ratio of the descriptions that successfully accepted by our system and the corresponding LOGO commands are correctly generated. We collected 877 descriptions when the six participants use our system to go through the example-oriented tasks and the test as presented in §5.2, and to just play with the turtle. The collection includes 19 types of commands as shown in Table 5.3 and we set these 19 types as the benchmark. We ask the users to describe these types of commands in their own way and collect another 1,000 pieces of descriptions. Thus in total, we have 1,877 natural command descriptions, about 100 for each type.

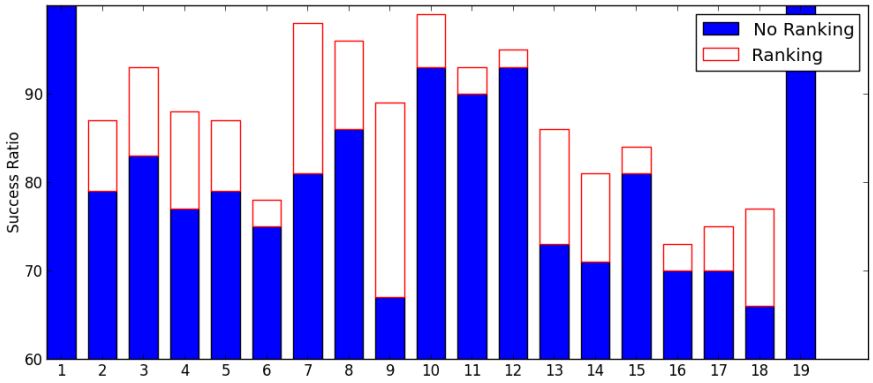

Figure 5.4. Success rates for synthesis

In total, we have constructed 96 rules in the PiE script, including 87 from the original library and 9 adapted, to understand the natural language descriptions. Typically, we make rules for the system without ranking. To test the effectiveness of the ranking algorithm described in §3.3.2.1, we first test our system with unranked rules to get the *Success rate*. Then, after applying our ranking algorithm with the rules, we test the system again to get another set of *Success rate* and compare with the previous unranked results.

In Figure 5.4, we show the *Success rate* in two cases when the rules are ranked or not, to synthesize the benchmark commands from the natural expressions. From the figure we can see that with ranked rules, higher *Success rates* are achieved. Our system performs well as it achieves average success rate of 88.4% in most of the commands that supported by the LOGO language. PiE works in most cases but there are still some exceptions. For example, descriptions which are too oral, such as “how about 40 steps" or “move! turtle!", cannot be understood by our system. The former requires a context which we will address in the future; the latter is a partial command which needs a length parameter. It is our goal to improve further in the future.

	Benchmarks Natural Descriptions
1	Clear the screen
2	The turtle move forward a few steps and a line is drawn
3	The turtle move backward a few steps and a line is drawn
4	The turtle rotate a degree clockwise
5	The turtle rotate a degree anti-clockwise
6	The turtle move directly to a pointed place
7	The turtle's color is changed and the line is in the decided color
8	Put up the turtle and no trace will be left when move the turtle
9	Put the turtle back to the paper
10	Undo several previous commands
11	Change the width of the pen to a decided value
12	Let the turtle face a decided direction
13	Define a function with a name that includes several commands
14	Repeat several previous commands or a function
15	Draw a circle with a certain radius
16	Draw a triangle with certain lengths of the edges
17	Draw a square with certain lengths of the edges
18	Draw a diamond with a certain length of the edge
19	Quit the system

Table 5.3. Benchmarks for synthesis

Chapter 6
Conclusion and Future Work

In this thesis, we developed a system called Programming in Eliza (PiE) to synthesize LOGO programs from natural language conversations between users and computer. We adopted Eliza for a novel application on program synthesis. Our preliminary experience showed that PiE can assist and enhance programming experience of novices as well as experiences programmers. We also developed the voice module and made it an online adaptation. With the new extension, the programming tool is accessible to children before they learn to type and spell. We also got more feedbacks from users from different countries and backgrounds since the release of the online version.

We will focus on making PiE some other adaptations in the future to realize our original objective, that is to use one syntax to program for various platforms. This applies to the Spreadsheet engineering, Shell programming, etc. Further, we can build multi-lingual models such as the Chinese version of PiE that children from other countries can learn programming when play with PiE.

Appendix A
Tutorial for Online PiE

A.1 Brief Introduction

PiE (Programming in Eliza/English) is a tool that helps beginners to learn how to program. By interacting in natural language, currently we only can handle English, user can manipulate the “Turtle" (which is represented as an triangle in this design) on the canvas. The turtle will leave the trace wherever it goes and these traces will be a beautiful pattern.

A.2 Get in a Dialog with PiE

A.2.1 First Try

- Type in “Can you forward 150 steps?” and Click the “Talk”. See what happen?
- Great! The turtle just moved forward 150 steps! And PiE confirmed your command in the response box.
- Then Type in “Turn left 90 degrees!” and Click the “Talk”. You can also type the “Enter” on the keyboard instead of the Click.
- You will then see the turtle turn left 90 degrees!
- Next, let’s try move the turtle quicker and more fluently by using repeat.
- Type in “Repeat last 2 commands for 3 times!”
- Yeah! You got a square! Right?

- You can clear the screen by “Clear the screen!”

A.2.2 Move the Turtle

- Basically, the turtle can move forward and backward. Try “Go forward 150 steps!” and “Get back 150 Steps”.
- Besides, straight move, the turtle can turn clockwise(right) or anti- clockwise(left). Try “Turn left 90 degrees and Turn right 90 degrees”. You can name any degree between 0 to 360. In addition, try “Turn left” and “Turn right”. The turtle will by default turn 90 degrees.
- If you do not want the pattern any more, just “Clear the screen!”

A.2.3 Attributes can be Modified

- The turtle can draw with its movements. If you want to change how the trace looks like, please go on with this section.
- First, let’s try to make the line bolder. Type in “Can you use a bolder pen?” and then try “Go forward 100 steps!” How it look?
- Then, let’s make the line lighter. Type in Use a lighter pen and try again “Go forward 100 steps!” See the difference?
- You can also change the color of the line by “Can you change to color red?” You can change to any other colors like: grey, purple, green, yellow, black, blue, red.
- Test with “Forward 100 steps.” Your turtle might hit the wall if you continued three 100 step forward. After it hit the wall, it will appear from the other side.
- Sometimes, you just want to move the turtle but not leave any traces. In this case, let’s suspend the turtle.
- Suspend the turtle by “Get up!” and the turtle will be off the canvas. There will be no trace left when the turtle is off the paper. You can try “Can you forward 50 steps?” to double check.
- Put down the turtle by “Get down!”

- Test with “Forward 100 steps”.

A.3 Basics in Programming

In this section, we will tell you some basics in programming.

A.3.1 Loop

- Loop is a faster way to get things done if you want to repeat several commands over and over again. Let’s see how to use it.
- First, let’s try these commands one by one. “Forward 150 steps, Turn left 90 degrees, Forward 150 steps, Turn left 90 degrees, Forward 150 steps, Turn left 90 degrees, Forward 150 steps, Turn left 90 degrees,” What do you get? A square! But how to use less commands to realize the same thing? Try these commands one by one: “Go forward 100 steps.” “Turn left.” “can you repeat last 2 commands for 3 times? ”
- You see. We use 3 commands to draw a square instead of 8 commands.

A.3.2 Function

You can read this part after the next section “Write in Scripts". Function helps to reuse some encapsulated commands. Try the following commands:

```
Let’s define a function called Square
  Repeat the following commands for 4 times
    Go forward 100 Steps
    Turn left 90 degrees
  End the Repeat
End the function
Then Clear the screen
Call the function square
```

To experience the benefit from Function, you can go on with

```
Turn right 60 degrees
Again, Call the function square
```

A.3.3 Write in Scripts

You can write down a few sentences to manipulate the turtle at a time instead of being involved in the tedious dialog. Take the square as an example. Try copy-paste the following commands in the Multi-line Mode by clicking on the arrow beside the Talk button.

```
Use a red pen
Let the pen be bolder
Repeat the following commands for 4 times
  Go forward 100 Steps
  Turn left 90 degrees
End the Repeat
```

After the copy-paste, click on the Talk button. What do you get? You can revise the script in the Multi-line mode by adding or deleting in the box. Try add Repeat the following commands for 36 times after Let the pen be bolder and Turn left 10 degrees and End the repeat after the original End the Repeat. What do you get?

```
Use a red pen
Let the pen be bolder
Repeat the following commands for 36 times
  Repeat the following commands for 4 times
    Go forward 100 Steps
    Turn left 90 degrees
    End the first Repeat
  Turn left 10 degrees
End the second repeat
```

What? Did not see it clearly. Let's try it step by step. In the third command, change "36" into 1, 2, 4, 8, 12, 16 respectively. Do you understand now?

A.4 Try More

Till Now, you are good with basics in programming and You can handle most cases in LOGO to draw any graphs. Don't believe it? Try the Examples!

Appendix B
Examples for Online PiE

B.1 Draw Polygon!

Try the following commands first: You can click on the block of commands!

```
clear the screen
Repeat the following for 4 times:
  Go forward 150 steps
  turn right
end the repeat
clear the screen
Repeat the following for 5 times:
  Go forward 150 steps
  turn right 72 degrees
end the repeat
clear the screen
Repeat the following for 6 times:
  Go forward 150 steps
  turn right 60 degrees
end the repeat
```

Task1: (1) Now, can you draw an octagon (8 sides)? Try modifying the code! Save the figure by Clicking on “Download Image". How about (2) decagon(10 sides)?

B.2 Draw Stars!

Try the following commands first: You can click on the block of commands!

```
clear the screen
Use a red pen!
repeat the following commands for 5 times:
  Can you forward 150 steps?
  turn right 144 degrees please!
end the repeat
clear the screen
Use a red pen!
repeat the following commands for 7 times:
  Can you forward 150 steps?
  turn right 154 degrees please!
end the repeat
```

Task2: Now, can you draw a (4) Nine-pointed star? How about (5) Twelve-pointed star? Save the Image. Hint: The angle turned in each round has relationship with how many points and 180.

B.3 Draw Spiral LOGO!

Try the following commands first: You can click on the block of commands!

```
clear the screen
repeat the following for 90 times
  let the label height be the same as the repeat count
  put the pen off the paper
  move forward repcount * repcount / 30 steps
  put the label Logo here
  move back repcount * repcount / 30 steps
  put the pen down
  turn right 10 degrees
end the repeat
```

Task3: Now, (6) can you use color green and draw these LOGOs again? Save the Image. Then, (7) if I have already implemented a function called set randomcolor, can you call the function and make the LOGO's color changed in each round?

References

Ballard, B. W., & Biermann, A. W. (1979). Programming in natural language: "NLC" as a prototype. In *Proceedings of the 1979 Annual Conference (ACM '79).*

Begel, A. (2005). Programming by voice: A domain-specific application of speech recognition. In *Proceedings of the AVIOS Speech Tech. Symposium—SpeechTek West.*

Biermann, A. W., Ballard, B. W., & Sigmon, A. H. (1983). An experimental study of natural language programming. *International Journal of Man-machine Studies*, *18*(1), 71–87.

Brill, E. (1992). A simple rule-based part of speech tagger. In *Proceedings of the Workshop on Speech and Natural Language.*

Code.org. (2014). *The hour of code.* Retrieved from `http://code.org/`

Conway, M., Audia, S., Burnette, T., Cosgrove, D., & Christiansen, K. (2000). Alice: Lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 486–493).

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: A 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, *15*(5), 107–116.

Cypher, A., & Halbert, D. C. (1993). *Watch What I do: Programming by Demonstration*. MIT press.

Dijkstra, E. W. (1979). On the foolishness of "natural language programming". In *Program Construction: International Summer School* (pp. 51–53). Berlin, Heidelberg: Springer Berlin Heidelberg.

Etemadi, R., Kharma, N., & Grogono, P. (2013). CodeMonkey: a GUI Driven Platform for Swift Synthesis of Evolutionary Algorithms in Java. In *Proceedings of 16th European Conference on Applications of Evolutionary Computation* (pp. 439–448).

Feurzeig, W., & Papert, S. (1967). The LOGO programming language. *ODP-Open Directory Project*.

Groce, A., Kulesza, T., Zhang, C., Shamasunder, S., & Burnett, M. (2014). You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Transactions on Software Engineering*, *40*(3), 307–323.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 317–330). New York, NY, USA: ACM.

Gulwani, S. (2012). Synthesis from examples: Interaction models and algorithms. In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (pp. 8–14).

Gulwani, S., Korthikanti, V. A., & Tiwari, A. (2011). Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming*

Language Design and Implementation (pp. 50–61). New York, NY, USA: ACM.

Kendrick, B. (1997). *Web turtle.* (sonic.net/˜nbs/webturtle)

Ko, A. J., & Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 301–310). New York, NY, USA: ACM.

Kress-Gazit, H., Wongpiromsarn, T., & Topcu, U. (2011). Correct, reactive, high-level robot control. *Robotics & Automation Magazine, IEEE*, *18*(3), 65-74.

Le, V., Gulwani, S., & Su, Z. (2013). Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (pp. 193–206). New York, NY, USA: ACM.

Lieberman, H., & Liu, H. (2006). Feasibility studies for programming in natural language. In *End User Development* (pp. 459–473). Springer Netherlands.

Liu, X., & Wu, D. (2014). PiE: Programming in Eliza. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (pp. 695–700). New York, NY, USA: ACM.

Maly, M. R., Lahijanian, M., Kavraki, L. E., Kress-Gazit, H., & Vardi, M. Y. (2013). Iterative temporal motion planning for hybrid systems in partially unknown environments. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control* (pp. 353–362). New York, NY, USA: ACM.

Manna, Z., & Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Language System*, *2*(1), 90–121.

Marron, A., Weiss, G., & Wiener, G. (2012). A decentralized approach for programming interactive applications with JavaScript and Blockly. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions* (pp. 59–70). New York, NY, USA: ACM.

Microsoft. (2012). *A National Talent Strategy.* `https://news.microsoft.com/download/presskits/citizenship/msnts.pdf`.

Myers, B. A. (1998). *Natural programming: Project overview and proposal* (Tech. Rep. No. CMU-CS-98-101). Pittsburgh, PA, USA: Carnegie-Mellon University.

Polozov, O., & Gulwani, S. (2014). Laseweb: Automating search strategies over semi-structured web data. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 741–750).

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin*, *39*(1), 213–217.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., & Brennan. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67.

Roget. (1982). *Roget's Thesaurus*. Longman Group.

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated feedback generation for

introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 15–26). New York, NY, USA: ACM.

Smith, D. C., Cypher, A., & Spohrer, J. (1994, July). KidSim: Programming agents without a programming language. *Communications of the ACM*, *37*(7), 54–67.

Vlas, R. E., & Robinson, W. N. (2011). A rule-based natural language technique for requirements discovery and classification in open-source software development projects. In *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS)* (pp. 1–10). IEEE Computer Society.

Waldinger, R. J., & Lee, R. C. T. (1969). PROW: a step toward automatic program writing. In D. E. Walker & L. M. Norton (Eds.), *Proceedings of the 1st International Joint Conference on Artificial Intelligence, IJCAI* (pp. 241–252). Morgan Kaufmann.

Weizenbaum, J. (1966). ELIZA—A computer program for the study of natural language communication between man and machine. *Communications of the ACM*.

Yuana, R. A., Faisal, M., Pangestu, D., & Putri, Y. R. L. (2015). Math thematic learning through the introduction of basic science-based programming games virtual robot for high school students. *Advanced Science Letters*, *21*(7), 2235–2238.