PiE: Programming in Eliza

Xiao Liu
College of Information Sciences and Technology
The Pennsylvania State University
xvl5190@ist.psu.edu

Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
dwu@ist.psu.edu

ABSTRACT

Eliza, a primitive example of natural language processing, adopts a rule-based method to conduct simple conversations with people. In this paper, we extend Eliza for a novel application. We propose a system to assist with program synthesis called Programming in Eliza (PiE). According to a set of rules, PiE can automatically synthesize programs from natural language conversations between Eliza and users. PiE is useful for programming in domain-specific languages. We have implemented PiE to synthesize programs in the LOGO programming language, and 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.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: [Program synthesis]; I.2.7 [**Natural Language Processing**]: [Language generation]; H.5.2 [**User Interfaces**]: [Natural language]

General Terms

Design, Languages

Keywords

Eliza, LOGO, program synthesis, natural language, rule-based

1. INTRODUCTION

Eliza is an early example of natural language processing that serves as a psychotherapist to interact with patients [22]. Based on a rule-based method, Eliza makes simple conversations with people in natural language. Flexibly, people try to communicate with Eliza in sentences with few constraints and Eliza responds according to a set of rules based on keywords. With such a system of "intelligence", we make an extension for a new application called Programming in Eliza (PiE), with the assistance of which, programs in target programming languages will be synthesized from the natural language commands provided by users. PiE demonstrates a new way for program synthesis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE'14, September 15–19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642967.

Program Synthesis is a concept that comes from the earlier automatic programming, the fundamental of which relies on the mechanical theorem-proving techniques [21]. However, the problem of automatic programming is much broader than expected since the proof of theorems involving existential quantifiers scarcely succeeds. In this case, some constraints have been appended to the automatic programming, which is gradually considered as program synthesis [15]. Program synthesis is more likely a concept that defines the automation of programs in some specific domains, such as robotics [11, 14], and spreadsheet programming [7].

Because of the versatility and applicability, natural language programming has been widely discussed as an easy way for novices to learn how to program [1, 5, 3]. 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. Definitely, this goal has not been achieved yet but it is feasible for machine to partially understand people [13], especially in a typically area. 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 [2]. 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 usually 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 bottom line. 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. Thus, the demand for programming in natural language arises again.

Aiming at lowering the entrance bar for novice programmers or children who cannot program, 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 the LOGO programming language to draw graphs. Figure 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. 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.

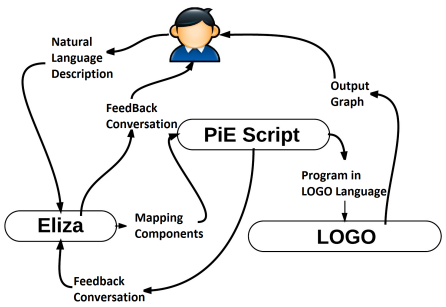

Figure 1: The PiE system architecture

- 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 rest of this paper is structured as follows. We first summarize the background of the PiE system: Eliza and LOGO. Then we demonstrate an example to illustrate a complete working flow of the PiE system. Next we introduce the PiE script used in this specific domain that we design, and elaborate the key techniques for program synthesis in our system. To evaluate the system, we use *Learning Efficiency* and *Success Ratio*. Related work and conclusion are presented in the last two sections.

2. BACKGROUND

In this section, we introduce the background on Eliza and the LOGO programming language.

2.1 Eliza and Rule-base method

Eliza, a primitive prototype of natural language processing, plays the role of a psychotherapist to communicate with patients [22]. 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 are made up 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 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 NLP 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.

2.2 LOGO

LOGO is a graphic-oriented education programming language [6]. The well-known application of LOGO is the Turtle Graphics [16], 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 special designed graph. For example, if a child wants the turtle to move forward 10 steps, she may use the command *FORWARD 10*. Table 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 in Eliza.

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: Web Turtle tutorial example: draw a square

3. EXAMPLE

In this section, we motivate our system with the same example from a web turtle tutorial [10] as described in Table 1. The objective is to generate LOGO programs when the input are sentences in natural language with few constraints.

As described in Table 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.

4. 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.

4.1 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 2. 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

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 2: The syntax of PiE-LOGO

or "Make a right turn" 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.

4.1.1 Statement

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

4.1.2 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 $number$, $STEP$ = $STEP1$ + $last[n]$

4.1.3 Repeat

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

4.2 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 predefined 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.

5. SYNTHESIS TECHNIQUES

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. 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.

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

Table 2: Regular expression matching

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: Natural command mapped to predicate "Forward"

5.1 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 2.

5.2 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 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 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 [17], 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.

5.2.1 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

(\.*) 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 4: Mapping rules to the "FORWARD" class

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 (\d)* (\.)*" stays before the rule "(\.)* color of the pen (\.)*(\.)*".

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

5.2.2 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 4.

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.

5.3 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 ratio. 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.

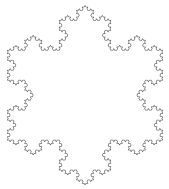

Figure 3: Output graph of demo

5.4 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.

6. EVALUATION

In this section, we first provide an demonstration 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 Ratio* 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.

6.1 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 demonstration 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 4 and the output graph is Figure 3. This demonstration 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.

6.2 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

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

Figure 4: Interaction with PiE for Demo

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 §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. They need to draw a base unit recursively and the base unit is a diamond as shown in Figure 6. The edge length of the diamond is 100 steps and the two different angles are 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 4, 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. PiE saves 43.8%, 21.8% and 22.4% of time, respectively, for non-programmers, novices and experienced programmers to draw the same graph. 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.

6.3 Success Ratio

We further evaluate the PiE system in terms of *Success Ratio* for the synthesized program. *Success Ratio* 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

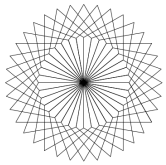

Figure 5: Learning efficiency test

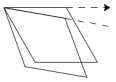

Figure 6: Base unit

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

Table 5: Time used in the Programming with LOGO and PiE

in §6.2, and to just play with the turtle. The collection includes 19 types of commands as shown in Table 6 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.

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 §5.2.1, we first test our system with unranked rules to get the *Success Ratio*. Then, after applying our ranking algorithm with the rules, we test the system again to get another set of *Success Ratio* and compare with the previous unranked results.

In Figure 7, we show the *Success Ratio* 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 Ratios* are achieved. Our system performs well as it achieves average success ratio 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 6: Benchmarks for synthesis

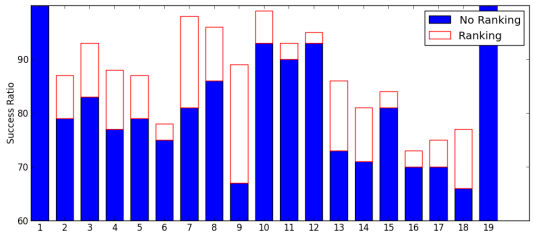

Figure 7: Success ratios for synthesis

7. RELATED WORK

7.1 Rule-based Artificial Intelligence

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 [4] adopts the rules for automatically tag speeches and it works as well as the stochastic tagger as proved by the author. Vlas [20] develops a rule-based natural language technique for classification of open-source software according to the informal documents. Le [12] is a case that programs can be synthesized for the windows phone commands using the rule-based relation detection algorithm to map the natural language descriptions to the relationship between incidents. Compared with statistical-based methods, the data "training" in the rule-based methods are often more efficient. In this paper, the rule-based method as Eliza does to synthesize programs performs quite well in the specific domain that we implement.

7.2 Program Synthesis & Applications

Program synthesis has been widely used in many domains. Gulwani [7] presents an application of program synthesis based on examples for the Microsoft spreadsheet. He also demonstrates a rule-based programming environment for spreadsheet data analysis [9]. Singh [19] shows the utilization of program synthesis as a method to generate feedback for simple coding assignments of introductory programming courses. After the first try, Singh again [18] applies the program synthesis method to automatically generate algebra problems. Besides the synthesis of algebra problems, Gulwani [8] displays a method to automatically solve ruler based geometry construction problems. In this paper, we present a system called Programming in Eliza (PiE) which automatically generates LOGO programs using conversations between users and the computer. It demonstrates a novel way for program synthesis.

8. CONCLUSION AND FUTURE WORK

In this paper, we have developed a system called Programming in Eliza (PiE) to synthesize LOGO programs from natural language conversations between users and computer. We adopts Eliza for a novel application on program synthesis. Our preliminary experience shows that PiE can assist and enhance programming experience of novices as well as experiences programmers.

In the future, we will design more experiments to test the PiE system with a more complete feedback. We also plan to experiment with PiE in the context of K-12 education as the first programming system for children. Furthermore, it would be interesting to apply PiE to domains other than LOGO.

9. REFERENCES

[1] B. W. Ballard and A. W. Biermann. Programming in natural language: "NLC" as a prototype. In *Proc. of the 1979 annual conference*. ACM, 1979.

[2] A. Begel. Programming by voice: A domain-specific application of speech recognition. In *AVIOS Speech Tech. Symposium–SpeechTek West*, 2005.

[3] A. W. Biermann, B. W. Ballard, and A. H. Sigmon. An experimental study of natural language programming. *International J. of man-machine studies*, 1983.

[4] E. Brill. A simple rule-based part of speech tagger. In *Proc. of the workshop on Speech and Natural Language*, 1992.

[5] E. W. Dijkstra. On the foolishness of "natural language programming". *Program Construction*, 1979.

[6] W. Feurzeig and S. Papert. LOGO. *ODP-Open Directory Project*, 1967.

[7] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[8] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[9] S. Gulwani and M. Marron. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, 2014.

[10] B. Kendrick. Web turtle, 1997. sonic.net/˜nbs/webturtle.

[11] H. Kress-Gazit, T. Wongpiromsarn, and U. Topcu. Correct, reactive, high-level robot control. *Robotics & Automation Magazine, IEEE*, 2011.

[12] V. Le, S. Gulwani, and Z. Su. SmartSynth: Synthesizing smartphone automation scripts from natural language. In *Proc. of the 11th annual int'l conf. on Mobile systems, applications, and services*. ACM, 2013.

[13] H. Lieberman and H. Liu. Feasibility studies for programming in natural language. In *End User Development*, pages 459–473. Springer, 2006.

[14] M. R. Maly, M. Lahijanian, L. E. Kavraki, H. K. Gazit, and M. Y. Vardi. Iterative temporal motion planning for hybrid systems in partially unknown environments. In *Proc. of Int'l conf. on Hybrid systems: computation & control*, 2013.

[15] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 1980.

[16] S. Papert. A computer laboratory for elementary schools. *Artificial Intelligence Memo*, 1971.

[17] Roget. *Roget's Thesaurus*. Longman Group, 1982.

[18] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[19] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*. ACM, 2013.

[20] R. Vlas and W. N. Robinson. A rule-based natural language technique for requirements discovery and classification in open-source software development projects. In *System Sciences (HICSS), 2011 44th Hawaii Int'l Conf. on*, 2011.

[21] R. J. Waldinger and R. C. Lee. PROW: A step toward automatic program writing. In *IJCAI*, 1969.

[22] J. Weizenbaum. ELIZA—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 1966.