Chapter 4
From Natural Language to Programming Language

Xiao Liu
Pennsylvania State University, USA

Dinghao Wu
Pennsylvania State University, USA

ABSTRACT

Programming remains a dark art for beginners or even professional programmers. Experience indicates that one of the first barriers for learning a new programming language is the rigid and unnatural syntax and semantics. After analysis of research on the language features used by non-programmers in describing problem solving, the authors propose a new program synthesis framework, dialog-based programming, which interprets natural language descriptions into computer programs without forcing the input formats. In this chapter, they describe three case studies that demonstrate the functionalities of this program synthesis framework and show how natural language alleviates challenges for novice programmers to conduct software development, scripting, and verification.

INTRODUCTION

Programming languages are formal languages with precise instructions for different software development purposes such as software implementation and verification. Due to its conciseness, the absence of redundancy causes less ambiguity in describing problems but on the other hand, reduces the expressiveness. Since the early days of automatic computing, researchers have considered the shortcomings

DOI: 10.4018/978-1-5225-5969-6.ch004

that programming requires to accommodate the precision with the adoption of formal symbolism (Myers, Pane, & Ko, 2004). They have been exploring techniques that could help untrained and lightly trained users to write programming code in a more natural way, and natural programming is then proposed (Biermann, 1983; Pollock, Vijay-Shanker, Hill, Sridhara, & Shepherd, 2013).

Natural language, on the contrary, is excessive but in its expressiveness lacks precision (Biermann, Ballard, & Sigmon, 1983). Describing problems in natural language gives a considerable freedom in clarifying requirements closer to practice, but specifications will contain ambiguities which are fatal to problem-solving. The errors result from two perspectives: structural errors and descriptive errors. Structural errors are caused by language designs. For instance, "then" is used for describing sequential events but is considered only as the "consequence" construct in those programming languages (Pane et al., 2001). Descriptive errors are those brought by participants in specific problem descriptions which contain errors and ambiguities as well.

To achieve a balance between programming languages that contain rigid symbolism and syntaxes and natural language that contains ambiguities. We discuss the question of what is natural to end-users by reviewing a few papers on the language features in non-programmers' descriptions to problem-solving. On top of the central finding on these features, we proposed a general framework for understanding natural language descriptions and automatically synthesizing programs for different software engineering purposes. With the implications of the proposed general framework, we take a closer look at different scenarios and conduct case studies on synthesizing a few domain specific languages. At the last, we discuss the potential limitations of the current framework and propose future works, before drawing a few conclusions.

BACKGROUND

Natural Programming, according to the definition from Brad Myer is "working on making programming languages, APIs, and environments easier to learn, more effective, and less error-prone". To achieve the goal, researchers have conducted studies on various methods to make the programming process more natural. But what is natural to end-users? A few terms, including closeness of mapping (Green & Petre, 1996) and cognitive dimension (Bonar & Soloway, 1985) were created to evaluate the learnability of a programming environment or its language syntax. The closer a programming method is to the problem world, the easier the solution can be composed.

Natural language programming is one of the significant directions being discussed that creates an easier way for people to compose the solutions for a programming task with limited programming experience. Great efforts have been made by researchers and some new language syntaxes have been created during the past two decades. In spite of the features of these new syntaxes, we are interested in essentially how natural they are when fitted into practical problem-solving. We ground our analysis in findings on vocabulary and structure features and discuss the implications for future designs of end-user programming environments.

PROGRAMMING PARADIGMS

On practical problem-solving, a recent study (Pane et al., 2001) reveals a preference for event-based descriptions such as statements that start with *if* or *when*. Meanwhile, a remarkable number of other paradigms are also observed such as constraint programming, declarative programming and imperative programming. Language paradigms are often discussed by researchers about their privileges and people are creating new programming environments for their specific domain of uses. We will get a closer view on these paradigms with a literature walk-through.

In event-based programming or event-driven programming, the program state goes to an event queue to find the next event and then calls some code to process that event (Lee, 2011). This paradigm is widely adopted in programming for graphical user interfaces and game designs due to its naturalness in describing the state machine transitions. E.g., Node.js, the popular cross-platform runtime environment for developing server-side Web applications is event-based. Many end-user programming environments also take this paradigm as their main feature. Pane et al. proposed HAND (Pane, Myers, & Miller, 2002), which is event-based, motivated by his findings. In recent studies, this paradigm is more popular in the domain of Internet of Things where IFTTT (If-this-then-that) recipes are created the most by users (Tibbets, 2010). Their preference to use event-based descriptions in this domain is confirmed with Blackwell's findings with an empirical field study as well (Perera, Aghaee, & Blackwell, 2015). Imperative language is one of the earliest paradigms in the programming history that uses statements to change a program's state. Typical programming languages that are considered imperative include Fortran, C, and Shell programming. LOGO (Feurzeig & Papert, 1967) is a successful and popular language for children to draw graphics. LOGO is based on Lisp and the imperative style of it enables users to describe a procedure of a Turtle movements. The imperative style programming is incorporated in some educational languages, such as Alice (Dann, Cooper, & Pausch, 2011), Looking Glass (Kelleher, 2008), and Moodie (Lieberman & Ahmad, 2010), which are designed for storytelling. These environments share a common feature that users have the solutions/algorithms in detailed procedures before they start to program.

Declarative language describes a problem rather than defining a solution, which is opposite to the imperative paradigm. SQL is an example of declarative languages that specifies the results one requires instead of the method to get it. The constraint-based style is a special kind of the declarative paradigm and Prolog may be the most famous representative. By specifying the true assertions in facts and rules, the logic system can solve problems automatically for users. There are also some end-user development environments use this kind of approach, mostly for the domain of testing and debugging. Burnett (Burnett et al., 2003) proposed an assertion-based method for spreadsheet debugging which allows users to find errors in spreadsheets by continuously adding constraints to cells. It is later improved by Erwig (Abraham & Erwig, 2008) with a more powerful reasoning system. These systems provide users an easier way to solve a problem with knowledge/facts but not reasoning/algorithms.

Programming environments with other styles, e.g., object-oriented (Price et al., 2000), runtime coding (Rode & Rosson, 2003), interactive programming (Liu & Wu, 2014), are proposed by researchers for different domains of uses. Hot topics including spreadsheet programming (Gulwani & Marron, 2014), web application development (Chang & Myers, 2014), and data modeling (Sarkar et al., 2014) have been touched in recent years. One apparent reason that so many languages exist is that it is difficult to decide how to evaluate programming languages, let alone which individual to evaluate. There is no single better language paradigm, just those more suitable for ones purpose. Therefore, it is of critical importance to analyze the domain of use and possible users' preference with empirical studies before designing a new environment. A good way to start any implementations is to collect some user descriptions of domain problems within a small-scale lab study or experimental walkthrough.

PROGRAMMING WORLD VS. REAL WORLD

Pane et al. (2002) observed some interesting findings on the distinctions between the programming world and real life from users' problem-solving descriptions. Based on our analysis, these distinctions can be categorized into four levels from the natural language perspective: word, sentence, abstraction, and precision. To create a successful natural programming environment, the first step is to correctly understand users' expressions and these cases should be handled appropriately.

Word-level distinctions are mostly observed in users' descriptions. A word can be used with a different meaning in users' descriptions compared with its definition in the programming world. For example, "then" is treated as an adverb for connecting two events in sequence by users which is consistent with its usage in daily life. But as a programming term, it means consequently and usually goes along with the term IF...THEN... The word "and" is another example which is a Boolean operator in programming but used as a sequence word quite often in users' descriptions. In addition to the distinctions between the programming world and real world, Ko et al. (2006) also point out a large number of variations among different users according to his analysis of people's descriptions of software problems, especially for noun words.

For the word-level distinctions, both rule-based and statistical-based NLP methods are adopted as solutions. In rule-based methods, the basic idea is to process words or sentences with same meanings according to a pre-defined dictionary, such as WordNet (Miller, 1995) which is a general synonym dictionary. Statistical methods usually look for some surrounding words as the context. It will pick up the closest semantic meaning for a word with the highest probability with the co-occurred words (Wang, Berant, & Liang, 2015).

Sentence-level distinctions are the differences in the structures of sentences with a same semantic meaning. For example, some users would say "if A do something unless B" while the others prefer "if A and not B do something". These distinctions come from different user habits and the various natures of programming tasks. Similar as the word-level distinctions, sentence-level distinctions can also be analyzed with natural language processing techniques. Dictionary-based techniques are widely used in early artificial intelligence, e.g., Eliza (Weizenbaum, 1966) and IBM Watson (Wikipedia, 2016). The pre-defined grammars in the dictionary are usually constructed by experts through empirical studies on the regularity of sentences. Recently, gradually more end-user programming systems tend to process natural language descriptions with statistical methods such as FlashFill (Gulwani & Marron, 2014) that enables end-users to program in a spreadsheet in natural language. Primary NLP models, such as SVM, are good enough for handling the sentence-level distinction problems (Mihalcea, Liu, & Lieberman, 2006).

Abstraction-level distinctions usually happen in describing an operation on some objects. Programs use data structures to organize objects in a programming task, which are usually defined by default. However, users are more accustomed to the data structures that are closer to life. For example, when applying the operation "cut" to a bag of apples, users are more likely to say "cut the bag of apples" rather than "pick an apple and cut it, do this for all apples in the bag". A natural programming design should appeal to these data abstractions that are more often used in real-life scenarios.

To tackle the problem from Abstraction-level distinctions, ideas from the programming language community can be borrowed. As a part of programming language designs, libraries are built for different usages. The same idea applies to end-user programming environments. A general library with programming data structures will be supported by default. Both designers and users can extend the library with new abstracted data structures and operations which are more natural to them if necessary.

Precision-level distinctions are the last kind of distinctions we want to discuss. It represents the distinctions caused by users' coarse expressions. Similarly, Ko et al. (2006) also report the inaccuracy in verb usages in his study. The implicit descriptions can cause problems like range with holes or overlaps which seems a minor issue, but will lead to big problems such as overflows. Many reasons account for these distinctions and the most significant one may be the difference between symbolic expressions and literal expressions. End-users prefer literal expressions in most cases, while programs only take in symbolic expressions. Literal expressions are more expressive than symbols but lose precision to some extent.

For the precision-level distinctions, one solution is making the environment interactive. Whenever the system gets a sentence that it cannot parse or an implicit expression contains ambiguities, it will interactively clarify the requirements with users' additional response. A basic type of interactions can be a reported exception or warning. To make it more user-friendly, a dialog-based interface will be a better choice.

GENERAL FRAMEWORK

On top of these requirements, we proposed a general framework that synthesizes programs from natural language descriptions based on Eliza. Eliza, a primitive prototype of natural language processing, plays the role of a therapist 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 the 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, answer in natural language will be generated automatically. Following is an example of how Eliza works (Table 1).

Table 1.

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?

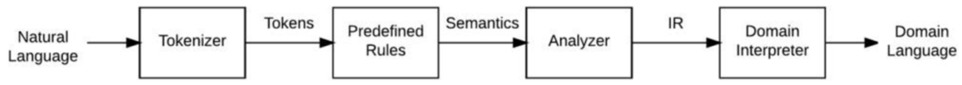
Although Eliza belongs to the first-generation NLP techniques using a rule-based method to understand users, it works quite well in specific domains. The proposed general framework works as in Figure 1. We encode the Decomposition Rules as the Tokenizer which breaks down the sentences into chunks. The mechanism is similar to conventional tokenizer in modern compilers. With preset regular expressions for string matching, we extract valid tokens and construct the symbol sequence that represents the given sentence. In order to get the semantics, the predefined rules will catch the sequential tokens and accurately locate the semantics with provided reassemble methods. After analysis on the selected reassemble method, the analyzer will generate the Intermediate Representation (IR) and it will then be ported to any specific domains with the assistance of the domain interpreter.

The proposed framework allows any kind of translation between natural language descriptions and formal languages. For different domains, we are supposed to analyze the possible descriptions and construct the domain-specific modules. The general framework has been applied in a few practical tools that serve different software engineering purposes, including the software implementation, scripting and verification. In the following part of this paper, we will discuss the functionality of the general framework and special defined modules with domain studies.

CASE STUDY

In this section, we will discuss specific designs which implement the general framework for different software engineering purposes. PiE (Liu & Wu, 2014) is an educational programming tool that synthesizes programs to draw graphs; Natural Shell (Liu et al., 2016) is an enabling tool that assists with scripting in various platforms; and EasyACL (Liu, Holden, & Wu, 2017) functions for constructing and verifying access control list for both Cisco and Juniper systems. The abovementioned tools are designed based on translating natural language descriptions into programming languages. We will detail the motivations, implementations and conducted studies as follows.

Figure 1. General framework for program synthesis



PiE

Aiming at lowering the entrance bar for novice programmers or children, we proposed a domain-specific program synthesis system called Programming in Eliza (PiE), with which, only the conversations in natural language with the computer are required. PiE is capable of synthesizing programs in the LOGO programming language, to draw graphs from the English conversation between Eliza and users. 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 feedback in natural language to users via the Eliza module.

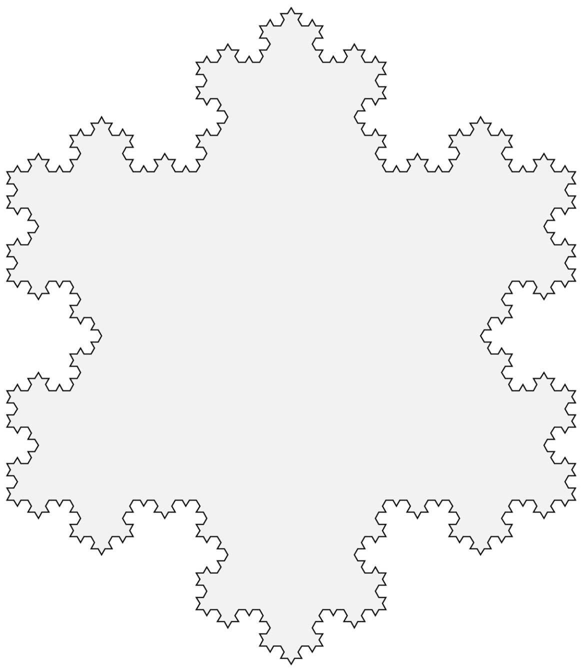
A user starts with a conversation with our prototype system, PiE (Programming in Eliza) by telling what she would like to code, and at the end of the dialogue, the system outputs a program. We started with the LOGO programming language. Our prototype system is based on Eliza, a primitive AI prototype. Original Eliza was designed to be a therapist. We make it a programming robot. A set of rules is developed for accepting LOGO commands in natural languages, and after a conversation, a LOGO program that draws a turtle graphics picture is produced.

Our study starts from a collection of 877 use descriptions of LOGO commands. The descriptions are collected from a lab study. The commands from the sampling cover all the implemented LOGO instructions. By analyzing the existing descriptions, combined with basic paraphrase assumptions, we implemented the basic library of decomposition rules which can catch 88.4% success ratio when applied to a larger data set which is consisted of 1,877 natural language descriptions. The implemented prototype is currently adopted as an assisting tool for introductory programming courses (Liu & Wu, 2016).

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. The input natural language is not case-sensitive and by using the regular expression matching, the system can tolerate some spelling mistakes as well. This demonstration is designed to show that PiE can handle complex tasks. The program goes a little bit further than a beginner can understand, but with a designed algorithm in hand, he could make simple conversations with PiE and draw a Koch Curve without much difficulty.

The interaction between the user and the PiE system is in Table 2 and the output graph is in Figure 2. The Koch Curve, also known as Koch snowflake (Von Koch, 1993), is one of the earliest fractal curves. It demonstrates a normal pattern which adopts the idea of recursion in the drawing. Table 2 shows the dialogs between the

Figure 2. Koch Curve



user and PiE system together with the LOGO program generated. Learners can parallel their natural language commands and this helps a lot since these abstract syntaxes are not easy to learn at the very beginning.

Our system can take in sentences in natural language with few constraints. Phrases like “move ahead 100 steps”, “Go on 100 steps”, “draw a line 100 inches long” will all be recognized and translated to the same LOGO command “FD 100”. In fact, there are still three statements and one loop in the natural language descriptions according to the user, but for beginners, they may focus more on how to design the “process” which can be seen as the algorithm but not the rigid syntax of a specific language at the very beginning.

We proposed PiE to alleviate the anxiety and enhance the learning experience with dialog in natural language. Without the necessity to remember a lot of command syntaxes, it will make the learning experience less stressful compared with traditional introductory programming courses. It will streamline students learning efficiency by teaching algorithm which is the intrinsic in programming but not the various standards. Meanwhile, it will create a lot of fun during the learning process as students can try anything they want to draw without the interruptions from many

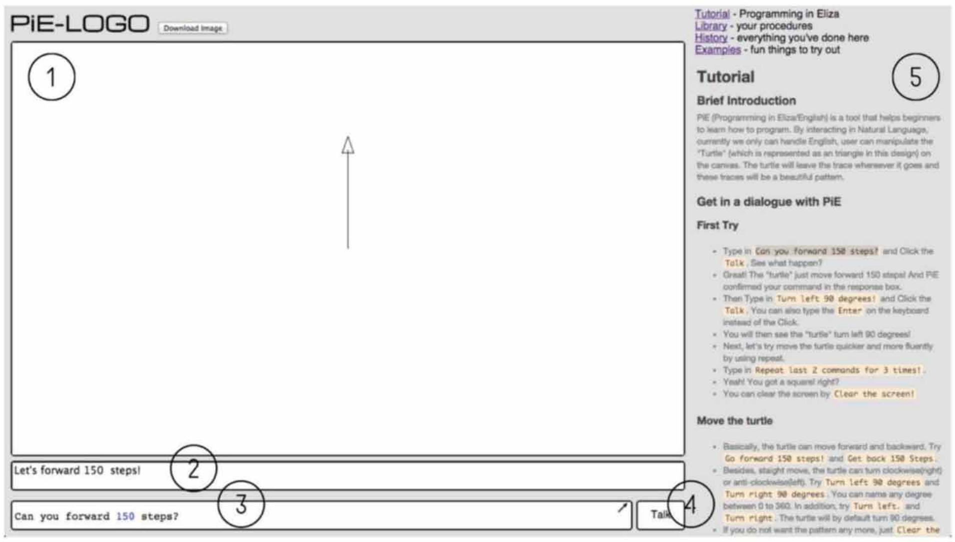
Table 2. Transcripts of Using PiE for synthesizing LOGO programs to draw Koch Curve

User Descriptions [PiE Start]	PiE Response	LOGO Program
> Let's first draw a line	Thanks for trying PiE. How can I help you?	
> Oh, I mean, to define a function called line	Okay, what is the length of that line? I see, what are the arguments of the function Line?	to line :count :n
> They are count and n	Okay, I have designed a function called Line with input arguments called count and n. Then?	ifelse :count = 1 [fd :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	
> In other cases, do the following things:	I see. do what else?	make "count :count - 1
> let count decrease by one	Okay, Count equals to Count -1	line :count :n
> repeat the line function with new count	I see. Function Line is called with the new argument count	lt 60
> turn left 60 degrees	Okay, the turtle turns 60 degrees to the left	line :count :n
> repeat the function line	Okay, Function Line is called	rt 120 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.	lt 60 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	end
> 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?	to koch :count :n
> still count and n	Okay, I have designed a function called Koch with input arguments called count and n. Then?	rt 30 line :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 120 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 30 line :count :n
> repeat the last command	I see. The turtle turns 120 degrees to the right. Function Line is called.	
> Let's call it Koch	I see, Function Koch is end. 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	Done

compiler errors and warnings. Consider the education use of the PiE system, when using this system to learn how to program, the non-experienced learners, especially for the young children, may experience a better interaction other than coding directly on their own.

To demonstrate its educational use, we built a web application as shown in Figure 3 and composed a complete introduction tutorial for entry-level programming courses. During a two-week summer school for high school students, PiE was adopted as a section for an introduction to different programming languages, i.e., Turing Machine, Python, and Prolog. During each lesson, a new programming language was introduced and students were exposed to a few labs to experience how to program with these languages. At the end of the summer school, students were encouraged to provide reflections on these different programming languages and how the interactions were with different environments. According to students' reflections, the experience with PiE-LOGO brought a new understanding of programming language. Students are in favor of PiE-LOGO comparing with the others as one of them commented,

Figure 3. Web App for PiE. (1) Drawing Canvas: The output of executing the synthesized command. (2) Feedback Box: The feedback from the chatbot in response to the natural language commands. (3) Natural Command Box: Type in your natural language commands here. (4) Execute Button: Try your natural language commands. (5) Instruction Panel: Instructions including tutorials for using PiE, some incorporated libraries, history commands, and challenge examples are shown here.



Let's forward 150 steps!

```
Can you forward 150 steps?
```

Tutorial - Programming in Eliza
Library - your procedures
History - everything you've done here
Examples - fun things to try out

Tutorial

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.

Get in a dialogue with PiE

First Try

- Type in `Can you forward 150 steps?` and Click the `Talk`. See what happen?
- Great! The "turtle" just move 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!`

Move the turtle

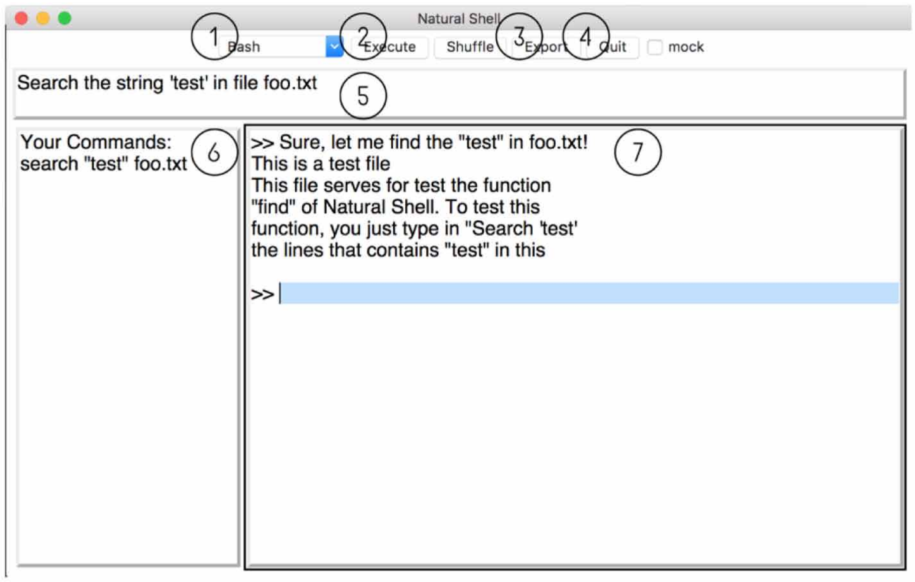
- Basically, the turtle can move forward and backward. Try `Go forward 150 steps!` and `Get back 150 Steps`.
- Besides, staight move, the turtle can turn clockwise(right) or anti-clockwise(left). Try `Turn left 90 degrees` and `Turn right 90 degrees`. You can name any degree between 0 to 360. In addition, 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`

My favorite parts of the course were PiE-LOGO. I enjoyed PiE-LOGO because it was the easiest to understand and I enjoyed the challenges to create the different shapes. I believe I learned a lot in the two weeks, especially since everyone at my school who takes computer science or AP computer science talks about how tough it is. I thoroughly and genuinely enjoyed the course, and I would not change anything about it.

NATURAL SHELL

Natural Shell is a new interface for users to interact with system kernel functions, within the design of a local desktop application. Instead of the numerous command line tools that have a dark background and over-simplified imperative functionality, Natural Shell is more user-friendly, with an interface more akin to a modern application program, as can be seen in Figure 4. There are three main boxes in our design, which are the Natural Command Box, Uni-Shell Command Box and Execution Result Box.

Figure 4. User interface of Natural Shell. (1) Syntax Button: Choose the target syntax. (2) Execute Button: Try out your natural language commands. (3) Export Button: Export the commands in target syntax as a local script. (4) Quit Button: Quit the application. (5) Natural Command Box: Type in your natural language commands here. (6) Uni-Shell Command Box: The synthesized commands are shown here. (7) Execution Result Box: Both the natural language feedback and execution return values are shown here.



```
Search the string 'test' in file foo.txt
```

```
Your Commands:
search "test" foo.txt
```

```
>> Sure, let me find the "test" in foo.txt!
This is a test file
This file serves for test the function
"find" of Natural Shell. To test this
function, you just type in "Search 'test'
the lines that contains "test" in this

>>
```

There are two modes of operation: Novice mode and Apprentice Mode. For novices who are not familiar with any shell syntaxes, Natural Shell enables them to harness the system functions using natural language commands. In this mode, the “natural command box” accepts the user’s natural language descriptions, which either can be a single-line command or multi-line scripts. Unlike the original shell commands, there is no syntax restriction on the natural language inputs, which provides users with more flexibility in composing commands and generates fewer errors as well. After typing in their natural language commands, the user can expect: (1) commands in the Natural Shell syntax which are presented in the “Uni-Shell command box”; (2) natural language feedback that confirms the entered commands which are shown in the “execution result box”; and (3) return values from execution of the synthesized script commands which is also shown in the “execution result box”. However, some users may have gained familiarity with Uni-Shell and it may be redundant for them to start with natural language descriptions. Therefore, in the

Apprentice Mode, these users can directly compose their script in Uni-Shell syntax inside the "Uni-Shell command box" and they can either execute the script in Natural Shell or export the script to local storage in any target syntax.

To demonstrate the ability of our system for end-users to write shell scripts/ batch files, we show two examples for users in different scripting skill levels: a non-experienced user only knows basic file operations in Windows; and an experienced cross-platform user knows a few scripting languages. We provided them with descriptions of the script functions in natural language or demonstrate by operations and what they were supposed to do step by step. We assigned them with 2 different tasks and recorded their behaviors, including (1) can they write the task scripts? (2) how can they understand the given materials? and (3) do they refer to any additional materials? We built our system to create a dialog-based scene for users and outputs the final scripts in a text file as well as execution.

In the first case, the user had no previous knowledge of any scripting languages, including the shell for Unix and batch for Windows, but he was asked to write a batch file with a series of basic file operations in the Windows operating system. We provided him a copy of instructions of a task in which, there were only a few conditions and several file operations like creating a folder or deleting a folder. We obtained this piece of script from a shell scripting online forum and it served for deleting current directory entries except for specified files. Before scripting, we taught the user some basic concepts like arguments, labels and directories. Then he was allowed to describe the commands in natural language unrestrictedly and interactively with the system but one after another in case of some conflicts. The interaction was conducted by the user under the supervision of an expert. During the interaction, he asked about the meaning of "shift" and "display". As a result, he generated the target batch file which is shown in Table 3 along with the interaction transcript. The task was completed without any syntax errors in the batch file. The user found our tool helpful, with which the process developed his interest in learning how to use scripts for file operation automation.

In the second case, the user was familiar with both the Windows and Unix operating systems and had written scripts in a few common-used scripting languages like Bash, WinBat and Tcsh. But he cannot remember the syntax quite well which leads to frequent reference on manual books. We retrieved a case from recent problem list on a shell programming online forum (Ramirez, 2015) which asked for getting the list of unique entries from different files and their counts. The questioner also provided a few input and output examples to detail the requirements. We asked the user to write a script in bash with and without our tool. After the task, we measured both the generated scripts and the behavior when he was conducting the task. He composed the scripts function correctly in both cases. However, we noticed that he referred

Table 3. Interaction of the use-case scenario

User Transcript	Response From Natural Shell	Batch File
If there is no specified file name, end the script	> Okay, Let's end the scripts when no argument specified for the file names.	IF "%1" == "" GOTO END
Create a directory called SAVE	> Okay, Let's create a directory called SAVE.	MD SAVE
Label the following as MOVE-FILE	> Okay, Let's label this part as MOVE-FILES	:MOVE-FILES
Copy the specified files to the SAVE folder	> Okay, Let's copy the files specified to the folder called SAVE.	XCOPY %1 SAVE
Shift to the next specified file	> Okay, Let's consider the next file.	SHIFT
If no files specified, then go to FILE-DELETE	> Okay, Let's do FILE-DELETE of no files specified	IF "%1"=="" GOTO FILE-DELETE
Else, go back to MOVE-FILE	> Okay, else we go back to the MOVE-FILES.	GOTO MOVE-FILES
End this part	> Okay, Let's end MOVE-FILES.	
Label the following as FILE-DELETE	> Okay, Let's label this part as FILE-DELETE.	:FILE-DELETE
Delete all the files in current folder	> Okay, Let's delete all files in current folder.	DEL .
End the FILE-DELETE	> Okay, Let's end FILE-DELETE.	
Please move all the files in SAVE to current folder	> Okay, Let's move all files in folder SAVE to current folder.	MOVE SAVE˙ .
Delete the directory called SAVE	> Okay, Let's delete the folder called SAVE.	RD SAVE
Label here as the END	> Okay, Let's label here for the ending.	:END

to the manual four times searching the usage of certain commands like "sed" and "wc", which did not happen when he was assisted by Natural Shell. Additionally, he claimed that taking advantage of our tool, it saved a lot of time from the worries about command options.

One of the students commented on Natural Shell as "Toy Scripting". She just started to learn Java and had no experience with shell scripting. She said, "*I will never use shell scripting without Natural Shell because the original syntax is too complex*". The simplicity of Natural Shell commands will be the most reason that novices will try scripting or she would prefer conducting the system tasks without a script. Her voice represents most of the non-experienced participants in our study and we believe that Natural Shell can encourage students to try system scripting which will enhance their daily working efficiency. Another student commented that "*It is great to know that there is no syntax for a programming language. According to my previous experience, only how to print 'hello world' can be remembered during the first round going through the manual book of any programming languages.*" As described by these students, Natural Shell will assist their experience with scripting in the first a few weeks as an introductory. It will be easier for novices to start with compared with any syntax provided by the system.

EasyACL

To lower the bar of entry for network administration training and reduce the configuration complexity, we also propose EasyACL, a tool that synthesizes ACL configuration commands for different platforms directly from natural language descriptions. To overcome the challenge from option specifications, we introduce a natural language interpretation system which accepts descriptions in a considerable flexibility and synthesize the target commands directly by extracting the semantics with a rule-based natural language processing method. To avoid redundant training processes, EasyACL can port the synthesized commands to different platforms, namely, Cisco and Juniper, for the current stage.

With an empirical analysis over the configuration process plus a thorough survey with network administrators in our department, a few complications are the main trouble-makers. The misuse of configuration options is one of the main issues that were brought up. For ACL configuration commands, options are critical which enlarge the semantic sets with succinct syntax. However, the large number of abbreviated options are disturbing for network administrators, especially those who are not apprenticed. Take the IP permit command for example; there are in total six options that it would accept. To permit the network flows of a specific type needs a few options filled out by the administrator who is required to read the specification carefully to avoid possible mistakes.

To write a correct permit, commands are difficult with these complex options to consider. Additionally, the command is case-sensitive, therefore, it is very likely for a careless administrator to misuse an option or two. On the other hand, the platform dependency of syntax makes the case more complicated. Network companies hold their own operating systems and with the design of platform-directed syntaxes, the ACL configurations are entirely different. Cisco and Juniper are two main network device manufacturers and they have their own configuration syntaxes respectively. If one wants to “permit HTTP traffic from IP 172.21.1.1 to IP 172.21.1.15”, the Cisco configuration and Juniper configuration are as follows:

```
# Cisco command
permit tcp host 172.21.1.1 host 172.21.1.15 eq 443

# Juniper command
filter 1 {
    term T1 {
        from {
            source-address {
                172.21.1.1/32; }
            destination-address {
                172.21.1.15/32; }
            protocol tcp;
            destination-port 443; }
        then {
            accept; }
    }
}
```

As demonstrated in this example, the syntax of Cisco ACL configuration commands is distinct from what Juniper system adopts. While Cisco commands are designed to be imperative, Juniper's are object-oriented (Davies et al., 2012). Because of these distinctions, people have to learn different syntaxes when they change platforms which brings more hurdles to the networking engineering training process.

Troubleshooting is one of the main motivations that we propose EasyACL. This tool will handle natural language descriptions with no ambiguity and then make interpretations. Therefore, we can leverage this feature for troubleshooting. To be more specific, when we find any errors in the network, such as no connectivity, we can describe the requirements in natural language and re-implement an access control list quickly. Problems should be resolved if it is simply an implementation error; otherwise, it should be a design problem. Next, we present an analysis of a practical implementation error (Orbitco, 2015).

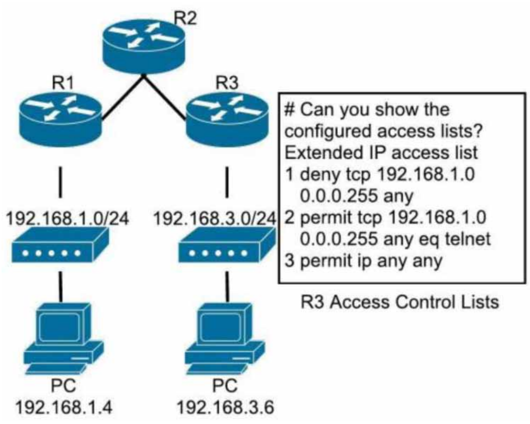
If the configuration requires: any tcp traffic from hosts 192.168.1.0/8 should be permitted through the telnet port; all ip traffic are permitted.

As shown in Figure 5, there is an error in the access control list implementation: Host 192.168.1.4 has no telnet connectivity with 192.168.3.6. It is a common mistake that many network engineers may encounter, because the router processes ACLs from the top down, statement 1 denies host 192.168.1.4, so statement 20 does not get processed. To troubleshoot this problem, we asked a network administrator to try with our proposed method.

He tried with feeding our system with the natural language descriptions one sentence after the other. And EASYACL synthesized the commands:

```
1 permit tcp 192.168.1.0
  0.0.0.255 any eq telnet
2 deny tcp 192.168.1.0
```

Figure 5. Example of Using EasyACL for ACL verification



```
# Can you show the
configured access lists?
Extended IP access list
1 deny tcp 192.168.1.0
  0.0.0.255 any
2 permit tcp 192.168.1.0
  0.0.0.255 any eq telnet
3 permit ip any any
```

```
  0.0.0.255 any
3 permit ip any any
```

Comparing the synthesized commands and the original implementation, statement 1 and 2 are reversed. The last line allows all other non-TCP traffic that falls under IP (ICMP, UDP, and so on). The network administrator corrected the implementation right away and claimed it a nice tool for network troubleshooting.

DISCUSSION

We have proposed a general framework that synthesizes programming languages from natural language descriptions. The proposed framework serves for various software engineering purposes, including development, scripting, testing, and verification. But essentially, we are providing more natural interfaces for cases where formal languages were adopted previously. Compared against tutorial-style training process, our framework has some key advantages. Most importantly, it conveys the meaning of programming concepts in plain language. The translation makes the training contents closer to users' daily language and it reduces the cost in debugging an incorrect command with syntactic errors. However, the end-users learn a substitution instead of original development. Therefore, they cannot learn a programming language if they cannot assign semantics to commands. Without such an understanding end-users are less able to remember syntax or make adaptations.

In addition to ensuring that the end-user understands the meaning of tasks in a programming language, the use of natural language translation also reduces the time it takes to learn. The casual and conversational dialogue included in our framework allows the end-user to transition from English to programming syntax at his or her own pace. Natural Shell and EasyACL both provide an intermediate phase, which accepts the formally defined language for users to switch to. Although, it is the syntactic sugar for the original development language, this intermediate representation has two benefits. First, the intermediate representation is platform-independent. That is to say, the commands can be ported to any platforms with same functionalities, i.e., Bash, Csh and Winbat for Natural Shell; Cisco and Juniper systems for EasyACL. Second, it allows users to adopt the syntaxes which are more natural towards themselves. Taking advantage of the function define in PiE, users can customize the languages caters to their daily uses.

Furthermore, because the generated framework responses in natural language and program syntax, the end-user is always able to easily refer between the two, should they become confused. Synthesizing programming language from natural language descriptions has been discussed widely. While natural language translation is largely advantageous over tutorial or training software as a method of programming education, there are some drawbacks of natural translation, as well as some instances where a tutorial-model is more effective. If an end-user is not sufficiently attentive, a natural language translation and program synthesis system has the potential to be too overly flexible, such that it prohibits learning. For example, if an end-user can use a wide range of natural language commands to synthesize a target program, and he or she is not attentive to recognize the associated program syntax, this may inhibit their learning by not directly forcing them to learn the syntax. Because tutorial software most often attempts to mimic real programming scenario, it does not typically have this issue.

CONCLUSION

Programming is difficult. We analyzed what is more natural to end-users with respect to language features and on top of the central finding on these features, we proposed a general framework that builds a bridge between the natural language and the programming language. Based on this framework, we conducted three case studies with our deployments for different software engineering purposes, including software development, scripting and verification. We discussed the motivations, functionalities of these studies and evaluated user reflections in practice. Our approach has shown a great improvement in terms of efficiency and usability, across all three case studies.

REFERENCES

Abraham, R., & Erwig, M. (2008). Test-driven goal-directed debugging in spreadsheets. *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 131–138. 10.1109/VLHCC.2008.4639073

Biermann, A. W. (1983). Natural language programming. In *Computer Program Synthesis Methodologies* (pp. 335–368). Springer. doi:10.1007/978-94-009-7019-9_10

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. doi:10.1016/S0020-7373(83)80005-4

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, *1*(2), 133–161. doi:10.120715327051hci0102_3

Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., & Wallace, C. (2003). End-user software engineering with assertions in the spreadsheet paradigm. *Proceedings of the 25th International Conference on Software Engineering*, 93–103. 10.1109/ICSE.2003.1201191

Chang, K. S.-P., & Myers, B. A. (2014). Creating interactive web data applications with spreadsheets. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, 87–96. 10.1145/2642918.2647371

Dann, W. P., Cooper, S., & Pausch, R. (2011). *Learning to Program with Alice*. Prentice Hall Press.

Davies, J. N., Comerford, P., Grout, V., Rvachova, N., & Korkh, O. (2012). An investigation into the effect of rule complexity in access control list. *Радіоелектронні I Комп'ютерні Системи*, (5), 33–38.

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

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A cognitive dimensions framework. *Journal of Visual Languages and Computing*, *7*(2), 131–174. doi:10.1006/jvlc.1996.0009

Gulwani, S., & Marron, M. (2014). NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 803–814. 10.1145/2588555.2612177

Kelleher, C. (2008). Using storytelling to introduce girls to computer programming. *Beyond Barbie & Mortal Kombat: New Perspectives on Gender and Gaming*, 247–264.

Ko, A. J., Myers, B. A., & Chau, D. H. (2006). A linguistic analysis of how people describe software problems. In Visual Languages and Human-Centric Computing (VL/HCC'06) (pp. 127–134). Academic Press. doi:10.1109/VLHCC.2006.3

Lee, K. D. (2011). Event-Driven Programming. In *Python Programming Fundamentals* (pp. 149–165). Springer. doi:10.1007/978-1-84996-537-8_6

Lieberman, H., & Ahmad, M. (2010). *Knowing what you're talking about: Natural language programming of a multi-player online game. In No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann.

Liu, X., Holden, B., & Wu, D. (2017). Automated Synthesis of Access Control Lists. In *Software Security and Assurance (ICSSA), 2017 International Conference on*. IEEE.

Liu, X., Jiang, Y., Wu, L., & Wu, D. (2016). Natural Shell: An Assistant for End-User Scripting. *International Journal of People-Oriented Programming*, *5*(1), 1–18. doi:10.4018/IJPOP.2016010101

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). ACM.

LiuX.WuD. (2016). *PiE-LOGO*. Available: http://bambool.github.io/pie-logo/index.htm

Mihalcea, R., Liu, H., & Lieberman, H. (2006). NLP (natural language processing) for NLP (natural language programming). In *International Conference on Intelligent Text Processing and Computational Linguistics* (pp. 319–330). Academic Press. 10.1007/11671299_34

Miller, G. A. (1995). WordNet: A lexical database for English. *Communications of the ACM*, *38*(11), 39–41. doi:10.1145/219717.219748

Myers, B. A., Pane, J. F., & Ko, A. (2004). Natural programming languages and environments. *Communications of the ACM*, *47*(9), 47–52. doi:10.1145/1015864.1015888

Orbitco. (2015). *What is ACLs Error? Solutions to ACLs errors Examples*. Retrieved from http://www.orbit-computer-solutions.com/network-troubleshooting-access-control-lists-errors/

Pane, J. F., Myers, B. A., & Miller, L. B. (2002). Using HCI techniques to design a more usable programming system. In Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on (pp. 198–206). IEEE. doi:10.1109/HCC.2002.1046372

Pane, J. F., Ratanamahatana, C. A., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*(2), 237–264. doi:10.1006/ijhc.2000.0410

Perera, C., Aghaee, S., & Blackwell, A. (2015). Natural Notation for the Domestic Internet of Things. In *International Symposium on End User Development* (pp. 25–41). Academic Press. 10.1007/978-3-319-18425-8_3

Pollock, L., Vijay-Shanker, K., Hill, E., Sridhara, G., & Shepherd, D. (2013). Natural language-based software analyses and tools for software maintenance. In *Software Engineering* (pp. 94–125). Springer. doi:10.1007/978-3-642-36054-1_4

Price, D., Rilofff, E., Zachary, J., & Harvey, B. (2000). NaturalJava: a natural language interface for programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces* (pp. 207–211). Academic Press. 10.1145/325737.325845

Rode, J., & Rosson, M. B. (2003). Programming at runtime: requirements and paradigms for nonprogrammer web application development. In *Human Centric Computing Languages and Environments, 2003 IEEE Symposium on* (pp. 23–30). IEEE.

Sarkar, A., Blackwell, A. F., Jamnik, M., & Spott, M. (2014). Teach and try: A simple interaction technique for exploratory data modelling by end users. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 53–56). IEEE. 10.1109/VLHCC.2014.6883022

Tibbets, L. (2010). *IFTTT the beginning*. Available: https://ifttt.com/blog/2010/12/ifttt-the-beginning

Von Koch, H. (1993). On a continuous curve without tangents constructible from elementary geometry. In *Classics on Fractals.* Westview Press.

Wang, Y., Berant, J., & Liang, P. (2015). Building a semantic parser overnight. Association for Computational Linguistic (ACL). doi:10.3115/v1/P15-1129

Weizenbaum, J. (1966). ELIZA. *Communications of the ACM*, *9*(1), 36–45. doi:10.1145/365153.365168

Wikipedia. (2016). *IBM Watson*. Author.