Syntactic Generation of Practice Novice Programs in Python

Abejide Ade-Ibijola

Formal Structures, Algorithms, and Industrial Applications Research Cluster
Department of Applied Information Systems
School of Consumer Intelligence and Information Systems
University of Johannesburg, Auckland Park Bunting Road Campus
Johannesburg, South Africa
abejideai@uj.ac.za, www.abejide.org

Abstract. In the present day, computer programs are written in high level languages and parsed syntactically as part of a compilation process. These parsers are defined with context-free grammars (CFGs), a language recogniser for the respective programming language. Formal grammars in general are used for language *recognition* or *generation*. In this paper, we present the automatic generation of procedural programs in Python using a CFG. We have defined CFG rules to model program templates and implemented these rules to produce infinitely many distinct practice programs in Python. Each generated program is designed to test a novice programmer's knowledge of functions, expressions, loops, and/or conditional statements. The CFG rules are highly generic and can be extended to generate programs in other procedural languages. The resulting programs can be used as practice, test or examination problems in introductory programming courses. 500,000 iterations of generated programs can be found at: `https://tinyurl.com/pythonprogramgenerator`. A survey of 103 students' perception showed that 93.1% strongly agreed that these programs can help them in practice and improve their programming skills.

Keywords: Synthesis of things, program synthesis, practice python programs, novice programmers, context-free grammar applications.

1 Introduction

Teaching novices how to program takes up a great deal of time and it requires a lot of patience [20]. Similarly, learning how to program is difficult for novice programmers, with evidences of high drop out rates in introductory programming courses [8, 17, 9, 35]. A lot of research has gone into, both: pedagogy models of teaching this subject, and software tools for aiding the learning process. This has given rise to the domain of *novice program comprehension and automatic tutoring* [34, 27] — the study of:

1. what misconceptions (or difficulties) novice programmers have [19],

2. what pedagogy models can help them [11], and
3. what technological interventions can be used to support the learning process of the subject [34].

Novice Misconceptions or Difficulties Novice programmers often struggle with comprehending the syntax of programming languages [11], having to learn a number of different set of skills at the same time [35], the debugging process [6], lack of practice [26], the complexity of certain topics (such as arrays, loops, and relational or boolean algebra for conditional statements) that are difficult to understand [10, 37, 7].

Pedagogy Models Some popular pedagogy models that have been adopted in teaching programming include: teaching without the vehicle of a language (e.g. using textual algorithms) [11], adopting the productive failure technique, i.e. giving students complex problems to solve while hoping they form their own solutions before giving them direct instructions [30], and teaching problem solving before programming [16].

Technological Interventions Several approaches have been employed to aid novice program comprehension using technology, such as introducing serious games [25] with findings revealing that these games add to the fun element in learning, and students rated the game as an effective way to learn programming [24]. Automatic program summarisation [15], automatic program narration [3] and program visualisation [33] aids have also been proposed to aid program comprehension.

One major way to aid novice program comprehension is to get novices to practice more [12, 16, 22, 29], as emphases has been laid on the lack of practice as one major reason for high failure rates [26]. This is not only true for programming, as it has been proven that acquiring long-term knowledge and skill often depend on the frequency of practice [21].

This work presents the syntactic generation of programming exercises — that can be completed with pen and paper — as a practice aid for novice programmers. To achieve this, we have adopted random context-free grammars (CFG) in the formalisation of programming templates and implemented these formalisms to produce unique instances of Python programs. These programs cover basic programming concepts such as assignment statement, function calls, evaluation of arithmetic expressions and predefined functions, conditional statements and loops. This process is described in Figure 1.

In Figure 1, we take a category of the desired programs (e.g. simple arithmetic programs, programs with loops, programs with if-statements, etc.) and an integer indicating the number of iterations of such programs that is to be generated, and use the predefined CFG rules to generate new and unique instances of the requested program.

We have leveraged on findings that claim that practice can aid novice program comprehension and hence, made the following contributions. We have:

Fig. 1. Process of Python program generation using CFG rules

1. designed a random context-free grammar (a set of rules) for the automatic generation of practice programming problems in Python — a programming language widely used in teaching introductory programming,
2. implemented the grammar rules and shown that it generates infinitely many Python programs that can be given to novice programmers as pen-and-paper practice problems, and
3. evaluated the usefulness of the generated programs and shown that novice programmers across two universities find them very helpful.

The remainder of this paper is organised as follows. Section 2 presents the background and related works. Section 3 presents the CFG design for Python programs generation. Section 4 presents the implementation of the CFG rules and iterations of generated programs. Section 5 discusses an evaluation of this idea, while Section 6 presents the conclusion and future work.

2 Background and Related Work

In this section we introduce the problem, justify the choice of Python, discuss the motivation and related works.

2.1 Problem Statement

The problem tackled in this paper is summed up in the following questions.

1. Can we aid program comprehension by algorithmically generating more practice programs?
2. How do we design and implement context-free grammars to answer the Question 1 above?
3. What is the perception of novice programmers about automatically generated programs as practice and/or comprehension aids?

These questions are answered in Sections 3, 4, and 5 of this paper respectively.

2.2 Motivation

The following are known challenges of teaching and learning introductory programming:

Programming is Difficult Learning to program is difficult [8, 9, 35], and more practice can aid this process [12, 16, 22, 29].

High Failure Rates There are high failure and drop-out rates from introductory programming courses around the world, despite extensive research which attempts to address the issue [17, 9].

Wrong Learning Style Students often find programming difficult when they adopt the wrong style of learning (e.g. memorising programs, practising with past questions, etc.) or have the wrong motivation (e.g. wanting to pass the course and proceed with their mainstream courses — if its an elective[1] [18].

These challenges have motivated this work, resulting in a technique for the automatic generation of Python programs that can be used as practice problems by novice programmers.

2.3 Why Python?

In 2016, Python was ranked the second[2] most widely used programming language in teaching introductory programming [31], and recent assessments show that the use of Python has resulted in better successes in teaching programming at first year [36]. This language is used in the University where we have evaluated the results in this paper. All Python programs generated and presented in this paper are of Version `3.x`[3].

2.4 Type and Syntax of Python Practice Programs

This work focuses on a specific type of programming practice problems, namely: *program tracing*. Program tracing exercises require a novice to determine the values of variables and validity of statements at every program state and hence, determine the final output (if any) of a program or report a bug. These types of exercises are well used to test a novice's knowledge of basic concepts such as functions, loops, conditional statements, etc. An example of this type of programming problem shown in Listing 1.1.

Listing 1.1. Sample program tracing exercise 1

```
#Determine the output of the following program fragment:
x = 5
for y in range(1,6):
        print(x * y)
```

[1] Electives are courses not in the mainstream of the offered degree. An example is a student studying towards a Bachelor's degree in Chemistry, who takes an introductory programming course in Python.

[2] After Java programming language

[3] Versions of Python can be found here: `https://www.python.org/doc/versions/`

Listing 1.1, the user is given a program fragment to test the knowledge of: assignment of values to variables, range in Python, the use of `for` loops, and print statements. Here, the novice is expected to produce the set of results: $5, 10, 15, 20, 25$ on separate lines.

Listing 1.2. Sample program tracing exercise 2

```
i = -11
u4 = 5
w = (i - u4)
print(w)
```

In Listing 1.2, the task is a lot simpler. A novice is expected to add the values of two variables together i and $u4$, and display the result of the addition. In order to generate this type of a practice problem, we just need to generate a "random program" that has the structure shown in Syntax 1.

Syntax 1:

initialise identifiers $n \in N$**, assign values** $v \in V$**:**
$n_1 = v_1$
$n_2 = v_2$
$\vdots$
$n_k = v_k, 1 \leq k < |N|$
set $n_i \in N \longleftarrow$ **expr(**$n_j \in N | i \neq j$**)**
display n_i

The `expr` function in this syntax is a recursive function that generates simple mathematical expressions as a continuous concatenation of `terms`. This syntax is generic, and can produce the instance shown in Listing 1.3.

Listing 1.3. A random instance of Syntax 1.

```
x = 4
t1 = 20
j9 = -19
v = ((x + j9)) - ((b - t1))
print(v)
```

However, it will be a tedious process to define many of this type of syntax for the automatic generation of practice python programs. Hence, we have adopted CFGs to formalise this process. With CFGs, non-terminal symbols are created to abstract the repeating components (e.g. identifiers, expressions, conditions, etc) of templates, therefore making it easier to describe the syntax of these programs.

2.5 Target Learning Outcomes

The learning outcomes that we target in this work cover the evaluations of:

1. in-built functions — mostly mathematical functions (e.g. `floor`, `ceiling`, `abs`, etc.),
2. expressions (arthimetic, logical and relational),
3. loops (`for`-loops, and `while`-loops), and
4. conditional statements, i.e. `if`, `elif`, `else`.

2.6 Related Work

While there is no work mainly in the automatic generation of practice python programs, there have been related work done (using similar techniques) in the areas of:

Synthesis of problems Generating exercises in algebra for MOOCs [28], generation of problems and solutions for natural deductions and proofs [4], generation of algebra problems to help with mathematics pedagogy [32] using a syntax-directed approach, and grammar-driven generation of regular expression problems and solutions [2].

Synthesis of artefacts Synthesis of geometry constructions [13, 14] and synthesis of social media profiles using probabilistic context-free grammars (PCFGs) [1].

2.7 Definition of Terms

Here we define some terms used in this paper.

Definition 1 (Symbol, Alphabet, and String [23]). *A symbol is an item or a single token. An alphabet, denoted by Σ is any finite set of symbols. A string is formulated from concatenation of zero or more symbols.*

Definition 2 (Context-free grammar [5]). *A context-free grammar (or CFG) G is a four-tuple: $G = (N, \Sigma, P, S)$ where*

1. *N is a set of nonterminals, also known as "syntactic variables". Nonterminal represent phrases/clauses in a sentence. Hence, nonterminals are sometimes referred to as syntactic categories, with every nonterminal defining a sublanguage of the language G.*
2. *Σ is a finite set of terminal symbols, disjoint from N, from which the actual content of a sentence is composed. Σ is referred to as the alphabet of the language defined by the grammar G.*
3. *P is the set of productions, each production consisting of a nonterminal, called the left hand side of the production, a forward arrow, and a sequence of terminal and/or nonterminal symbols, called the right hand side of the production.*
4. *S is the start nonterminal (or start symbol), used to denote the entire sentence. The relation $S \in N$ must always hold.*

More on CFGs can be found in Aho *et al* [5] and Martin [23].

3 Grammar Design for Python Practice Programs

In this section we present the design of a CFG for the automatic generation of Python practice programs. In program design and/or compilation, two classes of languages are often used, namely: regular and context-free languages. Regular

languages are used for lexical analysis while context-free languages (CFLs) are used to describe syntax or used for syntax analysis. In this work, the rules for our templates are mere concatenations of the smaller rules for identifiers, control structures, etc of the programming language, hence, it suffices to say that the concatenation of two or more CFLs will always produce a CFL. This is why we have chosen to model these templates using a CFG.

3.1 Building Block

First, we begin the design of our grammar $G = (N, \Sigma, P, S)$ with a building block of production rules $p \in P$ that result in terminal symbols $\alpha \in \Sigma$, such as letters, digits, etc. (see Productions 1 to 13). Production 1 defines letters that may appear in the formulations of identifier names — this excludes the letters $l, o \in \Sigma_{F_{set}}$. $\Sigma_{F_{set}} = \{l, o\}$ is a forbidden set of alphabets and this constraint is included because of the similarity between $l, 1$ and $o, 0$. In Production 4, we have abstracted a list of parameters to $\hat{p}$, and used this in defining the parameters taken by pre-defined functions in Production 10. Productions 5 to 9 defines operators (arithmetic, logical, and relational). Productions 11 to 13 are for formatting the final derived string (in this case, a string in this language is a complete Python program).

$$\texttt{<letter>} \longrightarrow l \in \left[\Sigma \cap \Sigma_{F_{set}}{}'\right] \tag{1}$$

$$\texttt{<digit>} \longrightarrow d \in 0 \mid \ldots \mid 9 \tag{2}$$

$$\texttt{<value>} \longrightarrow v \tag{3}$$

$$\texttt{<parameter\_list>} \longrightarrow \hat{p} \tag{4}$$

$$\texttt{<rel\_op>} \longrightarrow < \mid > \mid <= \mid >= \mid != \mid == \tag{5}$$

$$\texttt{<arth\_op>} \longrightarrow + \mid - \mid * \mid / \mid \% \tag{6}$$

$$\texttt{<logi\_op\_infix>} \longrightarrow and \mid or \mid \wedge \tag{7}$$

$$\texttt{<logi\_op\_prefix>} \longrightarrow not \tag{8}$$

$$\texttt{<logi\_op>} \longrightarrow \texttt{<logi\_op\_infix>} \mid \texttt{<logi\_op\_prefix>} \tag{9}$$

$$\texttt{<pd\_fxns>} \longrightarrow (pow \mid sqrt \mid trunc \mid floor \mid ceil \mid ...)\texttt{<parameter\_list>} \tag{10}$$

$$\texttt{<nl>} \longrightarrow newline \tag{11}$$

$$\texttt{<tab\_in>} \longrightarrow tab \tag{12}$$

$$\texttt{<spc>} \longrightarrow spc \tag{13}$$

We proceed and define more rules for identifiers, terms, operators, and expressions in Productions 14 to 18.

$$\texttt{<ident>} \longrightarrow \texttt{<letter>(<letter>|<digit>)}^* \quad (14)$$

$$\texttt{<term>} \longrightarrow \texttt{<ident>} \mid \texttt{<value>} \quad (15)$$

$$\texttt{<operator>} \longrightarrow \texttt{<arth\_op>} \mid \texttt{<rel\_op>} \mid \texttt{<logi\_op>} \quad (16)$$

$$\texttt{<expr>} \longrightarrow \texttt{<term><operator><term>} \quad (17)$$

$$\texttt{<enclosed\_expr>} \longrightarrow \texttt{<bra\_op><expr><bra\_cl>} \quad (18)$$

$$\texttt{<ident\_init>} \longrightarrow \texttt{<ident\_init><init>} \mid \quad (19)$$

$$\longrightarrow \texttt{<init>} \quad (20)$$

$$\texttt{<init>} \longrightarrow \texttt{<ident><=><value><nl>} \quad (21)$$

At many points in the programs, we desire to display/print outputs in the form of values, expressions or variables. Hence we define a symbol for displaying as follows:

$$\texttt{<display>} \longrightarrow \texttt{<value>} \mid \texttt{<term>} \mid \texttt{<expr>} \quad (22)$$

We proceed to build on this and specify productions for arithmetic, conditional and looping structures.

3.2 Arithmetic Expressions

Simple arithmetic operations are defined recursively in Productions 23 to 27, allowing occurrences of expressions in enclosed brackets, predefined and functions. This is used in Production 24 for assignment statements.

$$\texttt{<assignments>} \longrightarrow \texttt{<ident><=><sim\_arth\_eval>} \quad (23)$$

$$\texttt{<sim\_arth\_eval>} \longrightarrow \texttt{<sim\_arth\_eval><arth\_op><enclosed\_expr>} \mid \quad (24)$$

$$\longrightarrow \texttt{<sim\_arth\_eval><arth\_op><pd\_fxns>} \mid \quad (25)$$

$$\longrightarrow \texttt{<enclosed\_expr>} \mid \quad (26)$$

$$\longrightarrow \texttt{<pd\_fxns>} \quad (27)$$

3.3 If Statement Blocks

Here we describe productions for `if` statements.

$$\texttt{<if\_stmt>} \longrightarrow \texttt{<if><chain\_cond><:><nl>} \quad (28)$$

$$\texttt{<elif\_stmt>} \longrightarrow \texttt{<elif><chain\_cond><:><nl>} \quad (29)$$

$$\texttt{<else\_stmt>} \longrightarrow \texttt{<else><:><nl>} \quad (30)$$

The <chain_cond> symbol used by the if productions is described with Productions 31 to 37. Production 35 allows the not operator to appear in front of some relational conditions in order to negate these statements.

$$\begin{aligned}
\texttt{<chain_cond>} &\longrightarrow \texttt{<chain_cond><logi_op_infix><encl_cond>} \mid && (31)\\
&\longrightarrow \texttt{<logi_op_infix><encl_cond>} && (32)\\
&\longrightarrow \texttt{<encl_cond>} && (33)\\
\texttt{<encl_cond>} &\longrightarrow \texttt{<bra_op><condition><bra_cl>} && (34)\\
\texttt{<condition>} &\longrightarrow \texttt{<opt_not><cond_expr>} && (35)\\
\texttt{<cond_expr>} &\longrightarrow \texttt{<ident><rel_op>(<ident>} \mid \texttt{<value>)} && (36)\\
\texttt{<opt_not>} &\longrightarrow \texttt{<logi_op_prefix>} \mid \lambda && (37)
\end{aligned}$$

3.4 Loops

Here we describe two types of loops, for, and while loops. The symbol <initial> is given as a random number. Since we do not want exercises that will take a lot of time for a novice to complete, it is important to cap the number of iterations they will have to carry out in the process of tracing the loops. Hence, Production 40 computes the final values of the for loop using random numbers of step length and desired number of executions, these are bounded in the ranges shown in Productions 41 and 42 respectively.

$$\begin{aligned}
\texttt{<for_hdr>} &\longrightarrow \texttt{for<spc>in<spc>range<bra_op><initial>,<final>,}\\
&\qquad (\texttt{<step>} \mid \lambda)\texttt{<bra_cl><:>} && (38)\\
\texttt{<initial>} &\longrightarrow \texttt{<value>} && (39)\\
\texttt{<final>} &\longrightarrow \texttt{<step>} * \texttt{<exe_count>} + \texttt{<initial>} - 1 && (40)\\
\texttt{<step>} &\longrightarrow 1 \mid \ldots \mid 10 && (41)\\
\texttt{<exe_count>} &\longrightarrow 2 \mid \ldots \mid 4 && (42)\\
\texttt{<while_hdr>} &\longrightarrow \texttt{while<bra_op><condition><bra_cl><:>} && (43)
\end{aligned}$$

$$\begin{aligned}
\texttt{<for_loop>} &\longrightarrow \texttt{<for_hdr><nl><tab_in><display>} && (44)\\
\texttt{<while_loop>} &\longrightarrow \texttt{<while_hdr><nl><tab_in><display><adj_cond>} && (45)
\end{aligned}$$

In Production 44 and 45, the entire loop structures are defined, allowing for indentations with the <tab_in> symbol. <adj_cond> is a symbol derived with a function that adjusts the variables within the loop to ensure that the loop is not infinite and that every execution takes it closer to its termination.

3.5 Complete Python Programs

Now we give productions for three types of complete programs. Programs that tests knowledge of: arithmetic operations, conditional statements, and loops. Production 46 is straightforward, initialises identifiers, does assignments, and displays related contents. Similarly, Production 47 does initialisations and then allows an `if` statement block to appear. Production 48 makes sure that the `else-if` part of the if structure is optional.

$$\texttt{<prog\_arth\_expr\_eval>} \longrightarrow \texttt{<ident\_init><assignments><display>} \tag{46}$$

$$\begin{aligned} \texttt{<prog\_cond\_expr\_eval>} &\longrightarrow \texttt{<ident\_init><if\_stmt><tab\_in><display>} \mid \qquad (47)\\ &\longrightarrow \texttt{<ident\_init><if\_stmt><tab\_in><display>} \\ &\quad (\texttt{(<elif-stmt><tab\_in><display>)} \mid \lambda) \\ &\quad \texttt{<else\_stmt><tab\_in><display>} \qquad (48) \end{aligned}$$

$$\texttt{<prog\_loop\_expr\_eval>} \longrightarrow \texttt{<ident\_init><for\_loop>} \mid \texttt{<while\_loop>} \tag{49}$$

In conclusion to the rules of G, we define the start symbol $S \in P$ in Production 50 to 52.

$$\begin{aligned} \texttt{<prog>} &\longrightarrow \texttt{<prog\_arth\_expr\_eval>} \mid \qquad (50)\\ &\longrightarrow \texttt{<prog\_cond\_expr\_eval>} \mid \qquad (51)\\ &\longrightarrow \texttt{<prog\_loop\_expr\_eval>} \qquad (52) \end{aligned}$$

4 Implementation and Results

The grammar rules described in this work were implemented in a tool called the Python Code Generator, using .Net framework Class Library (FCL). The implementation produced a thousand iterations of unique programs in 1.04 seconds, 10,000 iterations in 9.01 seconds, and 100,000 iterations in 1 minute, 30.3 seconds; about half of a minute. We ran the program for one million programs and this completed in 4 minutes, 34 seconds. 500,000 generated programs can be found here: `https://tinyurl.com/pythonprogramgenerator`. Five iterations of generated programs (for the `if` statement category) are shown in Listing 1.4.

Listing 1.4. Sample outputs from python code generator

```
#---------------
# Code Number: 1
p9 = -2
p = 3
h = -17

g = (p - (13%8)) + ((h + p9))

if (h <= -26):
        print(g)
else:
        print(((p + h)) - (p9 + p))

#---------------
# Code Number: 2
k7 = -11
z = 4
b = 18
v = 2
t0 = 19
z6 = 0

p = (((z6 + k7)) - (math.ceil(-76.11) - v)) - (t0 + b) + (math.sqrt(16))

if not (k7 != -3) or (z != 4):
        print(p)
else:
        print((math.floor(-23.46) + t0) - ((v - b)))

#---------------
# Code Number: 3
z4 = 10

m = (math.trunc(-25.22) - z4)

if (z4 >= 12):
        print(m)
else:
        print(((z4 + z4)) + ((z4 - math.trunc(-17.10))))

#---------------
# Code Number: 4
x8 = 6
w5 = 12

a6 = (x8 + w5)

if not (w5 <= 3):
        print(a6)
else:
        print((w5 - math.pow(-1,1)) - ((math.sqrt(169) + w5)))

#---------------
# Code Number: 5
d1 = 12
g = -16

w3 = ((math.trunc(0.50) + g)) + ((30%5) + d1)

if not (d1 < 0) or not (g != -9):
        print(w3)
else:
        print(((math.pow(-1,2) - d1)) + (g - d1))
```

4.1 Solution Generation

For the generated problems to be very useful for novice programmers, it is important to also generate solutions that will serve as a benchmark. This is a relatively trivial task. This is because our grammar generates valid Python 3 programs, hence, passing this programs to a Python interpreter give us the output. In Figure 2, we describe how we have generated solutions to every Python file that was generated.

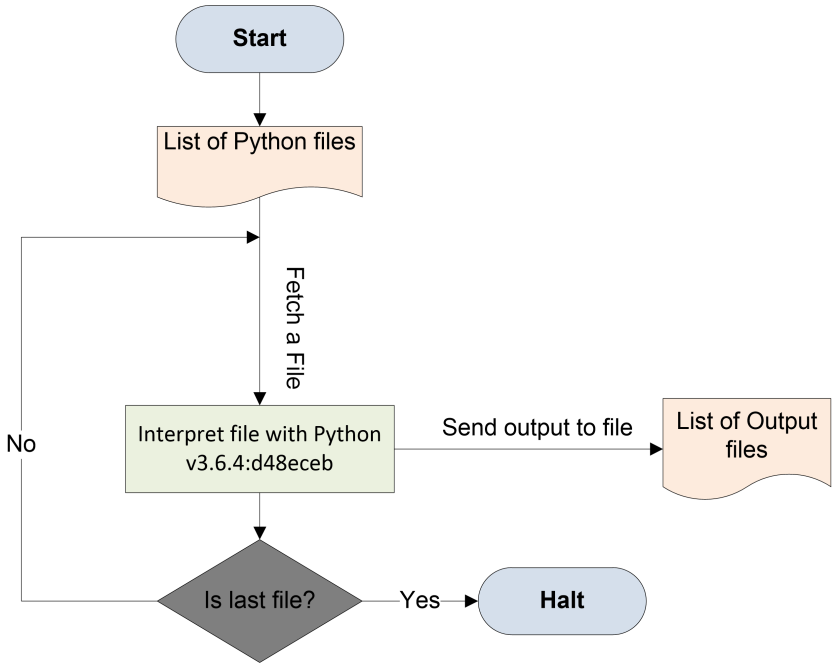


Fig. 2. Process of solution generation

Each python file is fetched in succession, and interpreted until there is no more file left.

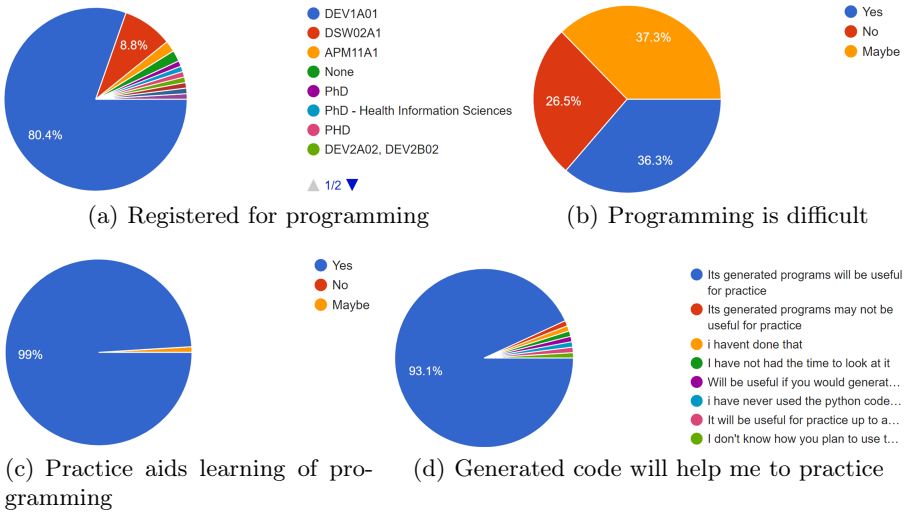
4.2 Experimental Proof of Uniqueness

For each category (`<prog_arth_expr_eval>`, `<prog_cond_expr_eval>`, and `<prog_loop_expr_eval>`) we experimented by generating one billion program instances and there was no repeated programs during execution. This can be explained with the large amount of possible permutations of identifiers, initialisations, conditions, and expressions that are derivable from the start symbol. It is possible to conduct a theoretical proof of uniqueness (or a very small number — close to zero — representing the probability of a program recurring) by

constructing a parse tree with the production rules, and computing the product of all possible branch of decisions. This is discussed further in the future work section of this paper.

5 Evaluation

In this section we present results from a survey-based evaluation of the students' perception of the generated Python programs, and its possible usefulness. We conducted an online survey at two main Universities in South Africa, namely: the University of Johannesburg and the University of the Witwatersrand. The respondents were mostly students that were registered for Computer Science or Information Systems degrees. Survey can be found here: `https://tinyurl.com/pcg-survey2018`.



(a) Registered for programming

(b) Programming is difficult

(c) Practice aids learning of programming

(d) Generated code will help me to practice

Fig. 3. Survey: relevance of generated programs

In total, we received 103 responses. 79.6% of the students were in first year currently taking a Python programming course, and a total of 92.1% are currently doing taking a programming course across different degrees and levels. Others are Masters and PhD students who admit that they have programmed at some point (See Figure 3(a)). We asked the students if they find programming difficult or too technical, and 73.6% believed it was either too difficult or sometimes too difficult. 26.5% of the students claimed they do not think programming is difficult (See Figure 3(b))— this difficulty spread agrees with the literatures in program comprehension as previously discussed in Section 2.

An overwhelming 99% agreed that practice can help them in learning programming better (See Figure 3(c)), with 93.1% strongly believing that the generated programs can help them in practice and improve their performance in programming (See Figure 3(d)). An interesting question is if the generated programs for each level are of the same complexity or difficulty. 50.5% of the students strongly thought the difficulty of the programs were the same, despite the difference in the instances. Total of 79.6% agree to some degree that the programs are of similar difficulty. A total of 90.2% agreed that the programs can be used for tests and examinations. With these feedbacks, we conclude that generating programs for practice, tests, and possibly examinations is worthwhile.

6 Conclusion and Future work

6.1 Conclusion

In this paper we have presented a CFG-based technique for the syntactic generation of practice python programs and solutions that can be administered to students in pen-and-paper program tracing sessions. We have shown that this technique can generate millions of practice programs in few minutes. Half a million of sample generated programs can be viewed or downloaded here: `https://tinyurl.com/pythonprogramgenerator`. We have also presented an evaluation that shows an overwhelming majority of students agreeing that the generated programs can help them in practising, and can be used in test, and/or examination questions.

6.2 Future Work

From here, we will explore the generation of buggy novice programs as debugging is one of the well known activities that improves programming knowledge of novice programmers. We will also make this tool available on a website to the Computer Science Education community. On the formal aspect, we will work on proving (theoretically) that it is possible (or impossible) to have repeated programs after a larger number of iterations.

References

1. Ade-Ibijola, A.: Synthesis of social media profiles using a probabilistic context-free grammar. In: Pattern Recognition Association of South Africa and Robotics and Mechatronics (PRASA-RobMech), 2017. pp. 104–109. IEEE (2017)
2. Ade-Ibijola, A.: Synthesis of regular expression problems and solutions. International Journal of Computers and Applications pp. 1–17 (2018), `https://doi.org/10.1080/1206212X.2018.1482398`
3. Ade-Ibijola, A., Ewert, S., Sanders, I.: Abstracting and narrating novice programs using regular expressions. In: Proceedings of the Annual Conference of the South African Institute for Computer Scientists and Information Technologists. pp. 19–28. ACM (2014)

4. Ahmed, U.Z., Gulwani, S., Karkare, A.: Automatically generating problems and solutions for natural deduction. In: IJCAI. pp. 1968–1975 (2013)
5. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (1986)
6. Alqadi, B.S., Maletic, J.I.: An empirical study of debugging patterns among novices programmers. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education. pp. 15–20. ACM (2017)
7. Baker, A., Zhang, J., Caldwell, E.R.: Reinforcing array and loop concepts through a game-like module. In: 17th International Conference on Computer Games (CGAMES). pp. 175–179. IEEE (2012)
8. Bergin, S., Mooney, A., Ghent, J., Quille, K.: Using machine learning techniques to predict introductory programming performance. International Journal of Computer Science and Software Engineering (IJCSSE) 4(12), 323–328 (2015)
9. Butler, M., Morgan, M., et al.: Learning challenges faced by novice programming students studying high level and low feedback concepts. In: Proceedings of the 24th ascilite Conference. pp. 2–5 (2007)
10. Dale, N.B.: Most difficult topics in cs1: results of an online survey of educators. ACM SIGCSE Bulletin 38(2), 49–53 (2006)
11. Fincher, S.: What are we doing when we teach programming? 29th Annual Frontiers in Education Conference 1, 12A4–1 (1999)
12. Foote, S.: Learning to program. Addison-Wesley Professional (2014)
13. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. In: ACM SIGPLAN Notices. vol. 46, pp. 50–61. ACM (2011)
14. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. SIGPLAN Not. 46(6), 50–61 (Jun 2011), http://doi.acm.org/10.1145/1993316.1993505
15. Haiduc, S., Aponte, J., Marcus, A.: Supporting program comprehension with source code summarization. ACM/IEEE 32nd International Conference on Software Engineering 2, 223–226 (2010)
16. Hill, G.J.: Review of a problems-first approach to first year undergraduate programming. In: Software Engineering Education Going Agile, pp. 73–80. Springer (2016)
17. Iqbal Malik, S.: Role of adri model in teaching and assessing novice programmers. Tech. rep., Deakin University (2016)
18. Jenkins, T.: On the difficulty of learning to program. In: Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences. vol. 4, pp. 53–58. Citeseer (2002)
19. Johnson, W.L.: Understanding and debugging novice programs. Artificial Intelligence 42(1), 51–97 (1990)
20. Lahtinen, E., Ala-Mutka, K., Järvinen, H.M.: A study of the difficulties of novice programmers, vol. 37, pp. 14–18. ACM (2005)
21. Lucariello, J.M., Nastasi, B.K., Anderman, E.M., Dwyer, C., Ormiston, H., Skiba, R.: Science supports education: The behavioral research base for psychology's top 20 principles for enhancing teaching and learning. Mind, Brain, and Education 10(1), 55–67 (2016)
22. Malik, S.I., Coldwell-Neilson, J.: A model for teaching an introductory programming course using adri. Education and Information Technologies 22(3), 1089–1120 (2017)
23. Martin, J.: Introduction to Languages and the Theory of Computation. McGraw-Hill, New York (2003)

24. Mathrani, A., Christian, S., Ponder-Sutton, A.: PlayIT: Game based learning approach for teaching programming concepts. Educational Technology & Society 19(2), 5–17 (2016)
25. Miljanovic, M.A., Bradbury, J.S.: Robot on!: A serious game for improving programming comprehension. In: Proceedings of the 5th International Workshop on Games and Software Engineering. pp. 33–36. GAS '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2896958.2896962
26. Özmen, B., Altun, A.: Undergraduate students' experiences in programming: Difficulties and obstacles. Turkish Online Journal of Qualitative Inquiry 5(3), 1–27 (2014)
27. Ramalingam, V., Wiedenbeck, S.: An empirical study of novice program comprehension in the imperative and object-oriented styles. In: Seventh workshop on Empirical studies of programmers. pp. 124–139. ACM (1997)
28. Sadigh, D., Seshia, S.A., Gupta, M.: Automating exercise generation: A step towards meeting the mooc challenge for embedded systems. In: Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education. p. 2. ACM (2012)
29. Shargabi, A., Aljunid, S.A., Annamalai, M., Shuhidan, S.M., Zin, A.M.: Tasks that can improve novices' program comprehension. In: IEEE Conference on e-Learning, e-Management and e-Services (IC3e). pp. 32–37. IEEE (2015)
30. Sharples, M., de Roock, R., Ferguson, R., Gaved, M., Herodotou, C., Koh, E., Kukulska-Hulme, A., Looi, C.K., McAndrew, P., Rienties, B., et al.: Innovating pedagogy 2016: Open university innovation report 5 (2016)
31. Siegfried, R.M., Siegfried, J., Alexandro, G.: A longitudinal analysis of the reid list of first programming languages. Information Systems Education Journal 14(6), 47 (2016)
32. Singh, R., Gulwani, S., Rajamani, S.K.: Automatically generating algebra problems. In: AAAI (2012)
33. Storey, M., Best, C., Michand, J.: SHriMP views: An interactive environment for exploring java programs. In: Proceedings of the 9th International Workshop on Program Comprehension. pp. 111–112. IEEE (2001)
34. Storey, M.A.: Theories, tools and research methods in program comprehension: past, present and future. Software Quality Journal 14(3), 187–208 (2006)
35. Wang, T., Su, X., Ma, P., Wang, Y., Wang, K.: Ability-training-oriented automated assessment in introductory programming course. Computers & Education 56(1), 220–226 (2011)
36. Yadin, A.: Reducing the dropout rate in an introductory programming course. ACM inroads 2(4), 71–76 (2011)
37. Zhang, J., Atay, M., Caldwell, E.R., Jones, E.J.: Visualizing loops using a game-like instructional module. In: Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on. pp. 448–450. IEEE (2013)