4-19-2021

Using Program Sythesis to Teach Testing in Introductory CS Classes

Li Mengzhen

College of Saint Benedict/Saint John's University, mli001@csbsju.edu

Using Program Synthesis to Teach Testing in Introductory CS Classes

Mengzhen Li

The College of Saint Benedict and Saint John's University

May 11, 2020

Using Program Synthesis to Teach Testing in Introductory CS Classes
By Mengzhen Li

Approved by:

Dr. Peter Ohmann

Thesis Advisor
Assistant Professor of Computer Science

Dr. Mike Heroux

Faculty Reader
Scientist-in-residence of Computer Science

Dr. Jeremy Iverson

Faculty Reader
Assistant Professor of Computer Science

Dr. Noreen Herzfeld

Chair, Department of Computer Science

Lindsey Gunnerson Gutsch, M.S.

Program Director
CSB/SJU Distinguished Thesis

Abstract:

Testing is a crucial and basic skill for programmers. Computer Science students should build a good sense of testing when they start to learn a new language. However, for most of the students in CS150, an introductory class of Python at the College of Saint Benedict and Saint John's University, it is hard to think about testing or writing tests because they worry too much about coding correctly. In order to enhance their understanding of testing, we built a system named SynpleTest using program synthesis techniques. It helps students to think more about testing because they don't need to worry about coding correctly when they are using SynpleTest. The system will take care of coding for them, and the only thing they need to do is to think of good test cases, (e.g., boundary cases) so that the system can generate programs that satisfy their requirements.

It can help students learn to write better tests by asking them to provide good test cases to make sure that the SynpleTest can exactly generate the programs that users want. Users will be asked to enter test cases repeatedly until they have entered enough good test cases (e.g., boundary cases) so that the generated program can satisfy the requirements. What's more, the generated Python programs consist of a different number of statements with different types. The SynpleTest now can generate simple Python programs combined with linear statements, square root statements and if-else statements. In this paper, we present how we built the SynpleTest system using program synthesis, and quantitative results measuring the efficiency of the system[1].

Keywords: Program Synthesis, Testing, Education

[1] I would like to thank my research advisor, Dr. Peter Ohmann, for guiding and supporting me over this year. I appreciate his generous help and constant encouragement. I also would like to thank my thesis committee members for their invaluable ideas and feedback.

Table of Contents

1. Introduction

There is no doubt that testing is important in programming. In 1978, Schneider put forward ten basic principles of introductory programming courses, and testing was one of them [4]. Two computer science professors from DePaul University, Will Marrero and Amber Settle, also believed that the emphasis on testing has been qualitatively beneficial for students in the introductory programming courses [6]. However, in reality, the importance of testing is often ignored in introductory programming courses. For example, computer science students at the College of Saint Benedict and Saint John's University might be taught a little about testing in the introductory classes. However, they will not officially be introduced to rigorous and formal testing until they can take a second-level computer science class. Besides, it is common that students in the introductory programming classes focus more on writing runnable programs rather than correct programs. They are satisfied if the programs they wrote can run and work for the one or two test cases provided by instructors. But passing few test cases cannot guarantee that the program is fully correct; we have to test more cases, especially boundary cases. For beginners, testing boundary cases are always the most overlooked part. Therefore, we decided to develop a system named SynpleTest to help students in the introductory Python classes to enhance their understanding of testing. Our system can achieve this goal by asking student users to provide valuable test cases so that the system can generate programs that are not only runnable but also totally correct. We used program synthesis to generate these programs based on the test cases users provided, because program synthesis is "the task of automatically finding programs from the underlying programming language that satisfy user intent expressed in some form of constraints" [10].

A possible way which SynpleTest can be applied in introductory computer science classes is that instructors provide students with a sample problem to solve and a program that solves the problem, then ask students to record the test cases they entered. To enable students to use the system more effectively, instructors might also limit the number of test cases students can enter to get the same program. Instructor can either use it as an in-classes activity or an after-class assignment. For example, suppose the instructor gave the following problem statement: "Create a program that takes two inputs and calculates the absolute difference of the two inputs". Then the student might enter the following test cases {7, 4, 3}, {5, 1, 4}, {-8, 5, 13}. SynpleTest might then generate a Python program, which subtracts the second input from the first input, to satisfy the three test cases the students entered. This would help the student see that these test cases are not sufficient. The program does not satisfy the original problem, but it does satisfy their test cases, so the student knows they need to enter more or better cases. By using SynpleTest, computer science beginners can enhance their understanding of testing and they can also develop a good habit of thinking thoroughly when writing programs. This will be helpful for their future studies and career.

In this paper, we will present the processing of building the SynpleTest system, and the results of this system.

2. Background

We begin by introducing some important concepts in program synthesis: specifications, search space and types. We then introduce a program representation, SSA form, which is often used for code optimization. Finally, we introduce a powerful SMT solver, Z3.

Program synthesis

"It is the task of automatically finding programs from the underlying programming language that satisfy user intent expressed in some form of constraints" [10]. In other words, program synthesis means automatically generating programs which satisfy the given specifications about what a program should do. Program synthesis can be implemented in various ways; one example is via logical specification. A logical specification is "a logical relation between the inputs and the outputs of a program" [9], it is one of the various forms that the user intern can be expressed in [10]. Test cases are examples of logical specifications, and SynpleTest is implemented in this way. Users of SynpleTest supply multiple test cases which includes several inputs and one expected output, and the SynpleTest will produce a program that correctly translates the inputs to output. We decided to use program synthesis because in this way users don't need to worry about writing runnable programs. The only thing they should care about is providing valuable test cases for SynpleTest to generate a totally correct program. Search Space is also an important term in program synthesis. The search space for a program synthesizer is all programs that could possibly be found. "In its most general formulation program synthesis is undecidable, thus almost all successful synthesis approaches perform some kind of search over the program space" [10]. The number of programs in any non-trivial programming language quickly grows exponentially with program size.

CEGIS

Counterexample-Guided Inductive Synthesis is a technique used in program synthesis and it is an iterative process. "Each iteration performs inductive generalization based on counterexamples provided by a verification oracle" [1]. CEGIS is illustrated in Figure 2.1. It starts from some specifications of the desired program. Specifications demonstrate what programs should do. They can either be a logical formula such as $\forall x\ P(x) = x+5$ or a set of input/output pairs such as (5, 10) for a program that adds five to its input [5]. The synthesizer can generate a candidate program which might satisfy the specifications, and the verifier will check whether the candidate program actually satisfies the specification. If it does, then we have the final program. Otherwise, the verifier will send specification elements that the candidate program failed to satisfy (here we call them counterexamples) to the synthesizer and start the process again. This time, the synthesizer will use counterexamples as guides to generate new candidate programs. One thing that needs to be mentioned here is that the techniques we used in our system are technically not CEGIS but are instead inspired by the CEGIS technique. This will be discussed in section 3.

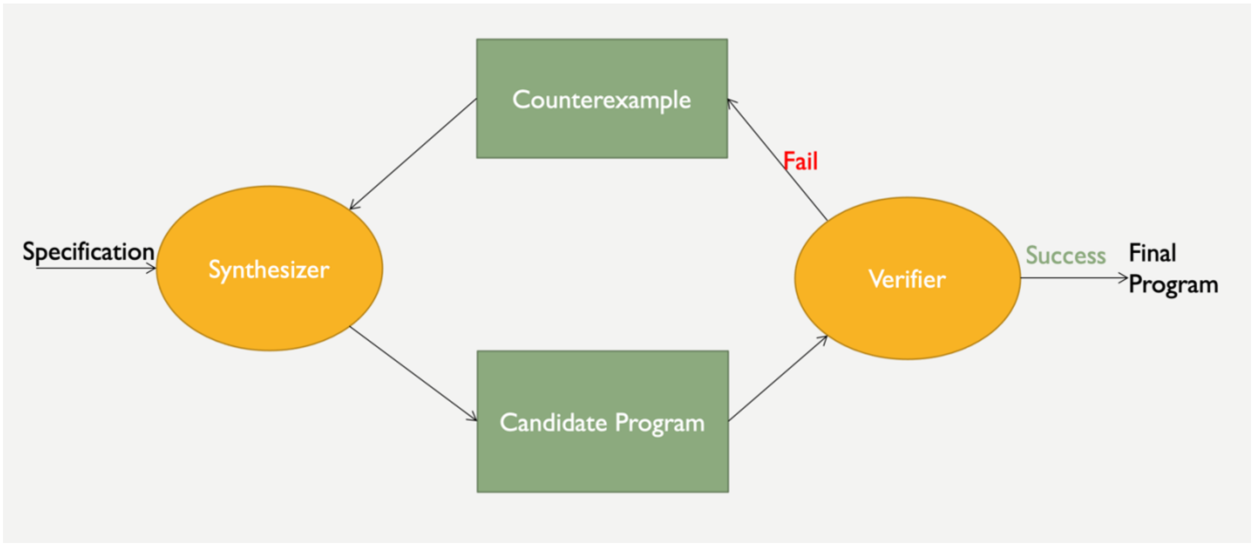


Figure 2.1 CEGIS Structure

SSA form:

SSA form stands for Static Single Assignment form, which requires that each variable be assigned exactly once, and every variable must be defined before it is used. “In optimizing compilers, data structure choices directly influence the power and efficiency of practical program optimization. It lends efficiency and power to a useful class of program optimizations” [7]. Therefore, SSA form is nearly always used for optimization. One of SSA form’s features is that all variables have distinct names, which means each variable can be declared exactly once in a program. The following figure shows an example of linear statements in SSA form. Note that in its single assignment version, each variable was differentiated from others by adding a suffix.

```
x = 5                      x1 = 5
x = x + 8                  x2 = x1 + 8
```

FIGURE 2.2 A linear statement and its single assignment version.

Z3

Z3 is a Satisfiability-Modulo-Theory (SMT) solver, which “determines the truth of a given logical formula built from typed variables, logical connectives, and typical operations such as arithmetic and array accesses” [3]. Z3 is a high-performance theorem prover developed at Microsoft Research. Z3 is used in many applications such as software verification, testing, and constraint solving [11]. Z3 can be used to solve mathematical problems. For example, it can solve a math problem with the following constrains: “x > 2”, “y < 10” and “x + 2*y == 7” “x, y are both integers”. There are three solutions to this problem, “x = 3, y = 2”, “x = 5, y = 1” and “x = 7, y = 0”. Z3 will return one of them to us. See section 3.3.5 for how we apply Z3 to our system.

3. APPROACH

3.1 Structure

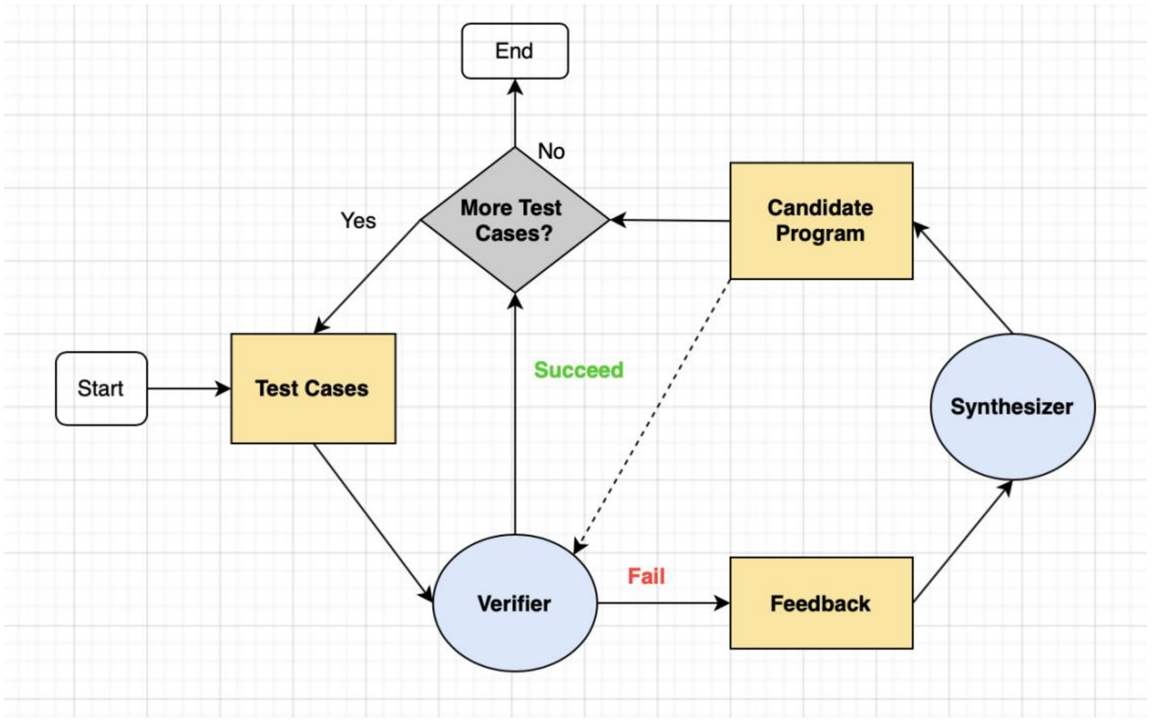


FIGURE 3.1

We designed a structure that is similar to CEGIS for the SynpleTest system. Based on the structure, the system will continue to produce new programs that satisfy the provided test cases until the user stops providing more test cases.

As is shown in Figure 3.1, the structure starts from feeding test cases, which is the yellow rectangle in the picture, to the verifier. The verifier's job is to help us verify whether the candidate program is valid. It has two inputs, test cases, and a candidate program. The test cases are provided by users and consist of user inputs and expected outputs. Each test case can have several user inputs but can only have one expected output. The candidate program is a Python program that might satisfy the test cases. It is not only the input of the verifier but also the output of the synthesizer. When users first start to feed test cases, the synthesizer has not started to work yet, so it cannot generate a candidate program for the verifier at this time. Therefore, the first candidate program will always be empty. In other words, when users first enter test cases, the verifier always fails because the candidate program is always empty. A valid candidate program can satisfy all of the test cases. To be more specific, it can take the user inputs from the test cases and produce the correct output. If the verifier proves that the candidate program is valid, congratulations! We found the satisfying program. Then we can either choose to keep feeding more test cases or stop the process, and accept the candidate program we have so far. Our SynpleTest system will keep iterating in this loop until the verifier fails, which means the candidate program cannot satisfy the new test cases. At that time, it is time for the synthesizer to work.

Then the verifier will pass the failed test cases (in the picture we call it feedback) to the synthesizer, which is the blue oval on the right side of the picture. The synthesizer takes feedback, the failed test cases, as inputs and then analyzes them. Finally, it will generate a candidate program as an output. After generating a new candidate program, the SynpleTest system will ask users if they want to keep entering more test cases. If the answer is yes, the system will start the loop again until users stop entering new test cases, and then it will report the current candidate program for users.

3.2 Verifier

3.2.1 Overview

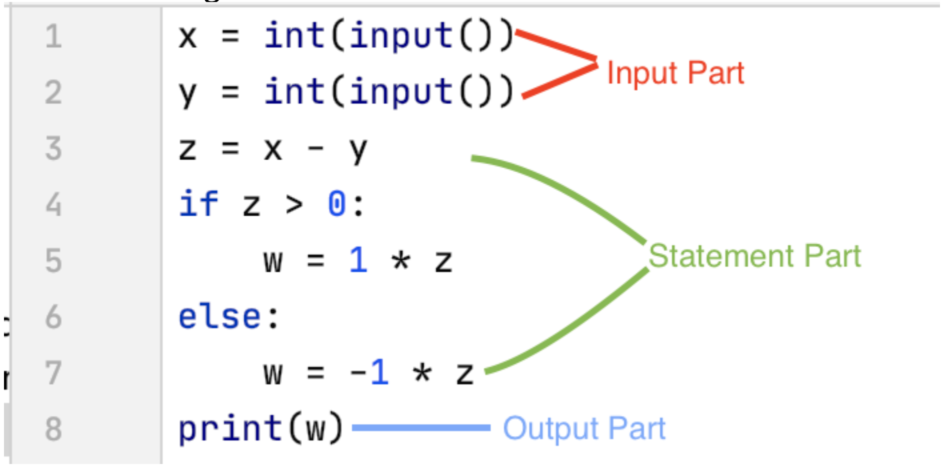
The purpose of the verifier is to check whether the candidate program satisfies the test cases, in other words, if the candidate program is valid. A valid candidate program can properly translate inputs to outputs for all provided test cases. Here is an example: suppose that a test case has a user input, 2, and an expected output, 3. If the actual output value of the candidate program is also 3, then the candidate program is valid. If the actual output value of the candidate program is 5, which doesn't match the expected output, then this is an invalid candidate program.

The verifier will take test cases and the candidate program as inputs. If the candidate program satisfies the test cases, there will be no need to pass the test cases to the synthesizer. Otherwise, the verifier will send feedback (all the failed test cases) to the synthesizer so that the synthesizer can use them to generate a better candidate program that satisfies all of the test cases. More explanation about the inputs of the verifier and how it works will be discussed in the following sections.

3.2.2 Inputs of Verifier

There are two inputs of the verifier: candidate program and test cases. The following paragraphs will introduce these two inputs in detail.

Candidate Program



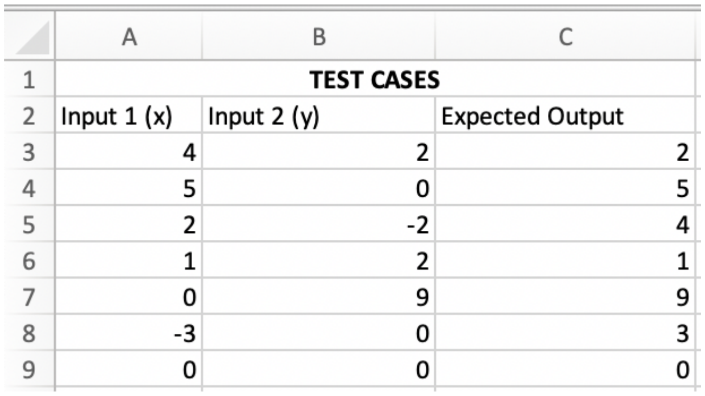
```
x = int(input())
y = int(input())
z = x - y
if z > 0:
    w = 1 * z
else:
    w = -1 * z
print(w)
```

FIGURE 3.2 A Simple Candidate Program

The candidate program is actually the input of the verifier and the output of the synthesizer. Figure 3.2 is a simple candidate program which can calculate the absolute difference of two inputs: *x* and *y*. We first ask users to enter the value of *x* and *y*. Then, we subtract *y* from *x* and get a value; we named the value as *z*. Since we want to get the absolute difference of *x* and *y*, we need to know whether the value of *z* is negative. If it is negative, we multiply it by negative one. If not, we multiply it by positive one. We assign both of the results a new name, *w*. At the end of the program, we print the value of *w*.

To make it easier to understand, we divided the candidate program into three parts: input part, statement part and output part. Firstly, the input part only has input statements. In Figure 3.2, the first two lines belong to the input part. The number of input statements must be decided by users when SynpleTest starts to run. For example, if users want to have two inputs then there will be two input statements at the beginning of the candidate program; accordingly, users will have to provide two inputs for each test case they provide. Secondly, the statement part differs for each candidate program and is actually generated by the synthesizer. The last part is the output part. Our system supports exactly one integer output. So, there is always only one print statement in the output part, such as line 8 in Figure 3.2. Currently, the system can generate candidate programs which have any permutation of linear, square root, and if-else statements using several integer inputs and one integer output.

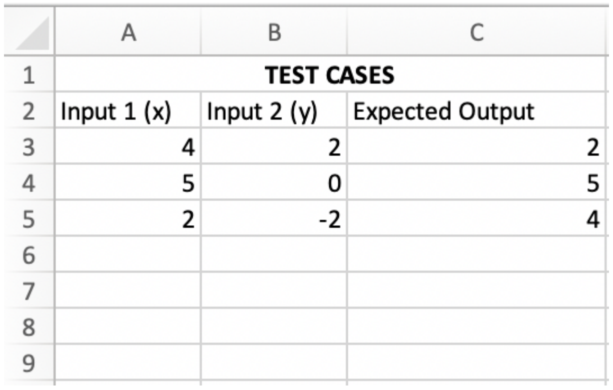
Test Cases



	A	B	C
1	TEST CASES		
2	Input 1 (x)	Input 2 (y)	Expected Output
3	4	2	2
4	5	0	5
5	2	-2	4
6	1	2	1
7	0	9	9
8	-3	0	3
9	0	0	0

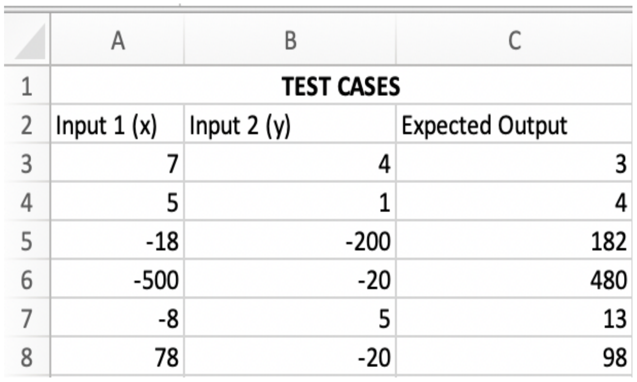
FIGURE 3.3 Examples of Test Cases

In the SynpleTest system, a test case specifically refers to a series of user inputs and a single expected output in integer form. To be more specific, user inputs are the integers that users want to provide to the candidate program; the expected output is the integer that they expect the candidate program can generate after doing some calculations based on the inputs they provided. In line 3 of Figure 3.3, Cell A3, B3, and C3 make up an example of a test case. Column A contains the values of Input 1, which is variable x of the program in Figure 3.2. Column B contains the value of Input 2, which is variable y of the program. The output of the program in Figure 3.2 is variable w. The values in Column C are the expected value of the output w. Users expect to get the value 2 for w from the candidate program when it has 4 as the first input value and 2 as the second input value. We show these values in a table for presentation purposes, though they are not ever actually stored or imported to the system in this format. A test case collection is a collection that may have a lot of test cases. It is very important to note that all of the test cases in the collection must have the same number of user inputs. For example, the test case collection in Figure 3.3 has 6 test cases. For each test case, there are two user inputs and one expected output. If users want the synthesizer to generate a more accurate candidate program, they have to provide valuable test cases. Figure 3.4 and Figure 3.5 are examples of less valuable test cases and valuable test cases respectively.



	A	B	C
1	TEST CASES		
2	Input 1 (x)	Input 2 (y)	Expected Output
3	4	2	2
4	5	0	5
5	2	-2	4
6			
7			
8			
9			

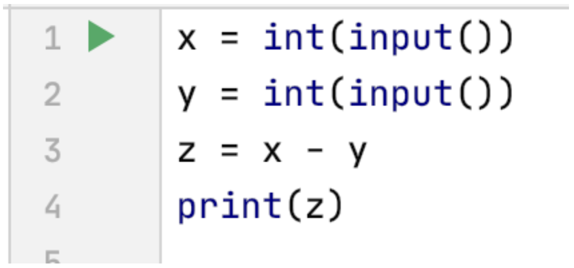
FIGURE 3.4 Less Valuable Test Cases



	A	B	C
1	TEST CASES		
2	Input 1 (x)	Input 2 (y)	Expected Output
3	7	4	3
4	5	1	4
5	-18	-200	182
6	-500	-20	480
7	-8	5	13
8	78	-20	98

FIGURE 3.5 Valuable Test Cases

Compared to the test cases in Figure 3.5, the test cases in Figure 3.4 are accurate but less valuable. This is because they can also be used to generate the candidate program in Figure 3.6, which only calculates the difference of two inputs instead of the absolute difference. The test cases in Figure 3.5 are more valuable because they test the boundary cases and they have more tests, so it does a better job of applying only to the candidate program in Figure 3.2.



```
x = int(input())
y = int(input())
z = x - y
print(z)
```

FIGURE 3.6

3.2.3 How Verifier Works

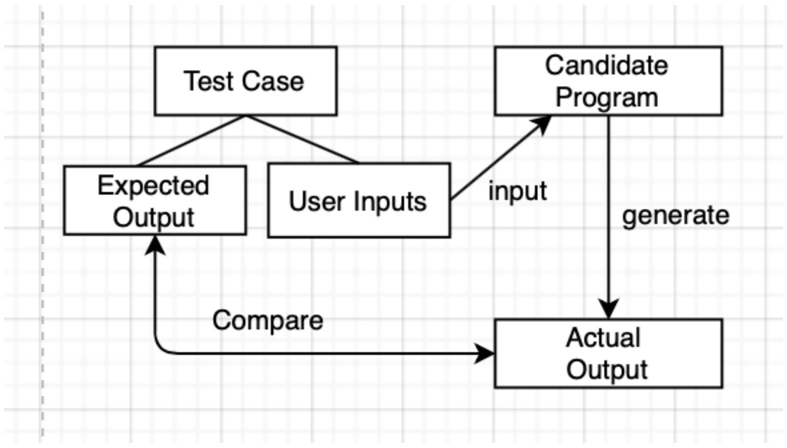


FIGURE 3.7 Workflow

In our SynpleTest system, the verifier we built is very simple. It will first take the user inputs from a test case as the inputs of the candidate program. Then, the verifier will run the candidate program on the user inputs, and the candidate program will generate an output. This is the actual output as it shown in Figure 3.7. After that, the verifier will compare the actual output to the expected output from the test case. If they match, it means the candidate program satisfies that test case. If they do not match, then the candidate program cannot satisfy this test case.

To be more specific, we created a verifier class and passed the test cases entered by users. We used the subprocess module in Python to make the verifier. It allows “a program to create new processes and connects to their input/output pipe and obtain their results” [8]. The verifier class has a method that passes the candidate program’s name as well as input and output as parameters. It then creates a subprocess to run the candidate program. At last, the verifier compares the output generated by the candidate program to the output in the test cases. If they are the same, that means the candidate program satisfies the user’s requirements, and it will return True. Otherwise, it will return False. The verifier will return True or False for each test case and the entire verifier will return all of these failed test cases.

3.3 Synthesizer

3.3.1 Overview

The synthesizer is the most complex part of the whole SynpleTest system, and it is also the place to generate the candidate program which will satisfy the test cases. The first step of the synthesizer is candidate generation, which generates all possible permutations of candidate code skeletons in the search space. A code skeleton is an incomplete code and has some holes to be filled in. Then the synthesizer filters them by syntax analysis, verifying whether the code skeleton we created is proper Python code or not. Finally, the synthesizer makes it semantically right by completing the code skeletons. After making sure that each statement is both syntactically and semantically correct, the synthesizer will put it into the candidate program.

3.3.2 Inputs of Synthesizer

The synthesizer will not start to work until the verifier finds that the candidate program doesn't satisfy the test case. From Figure 3.1, we can learn that the failed test cases are the inputs of the synthesizer. Conceptually, you can think of the synthesizer's job as coming up with a new program to satisfy the test cases that failed before. Technically, the synthesizer needs all the test cases, so it does not accidentally create a candidate program that only works for failed test cases. Even if it has to take all the test cases, the failed ones are still key. After all, the synthesizer's job is to make the failed test cases succeed.

3.3.3 Candidate Generation

To find a satisfying candidate program, we have to first get all of the possible candidate programs. Obviously, there are infinitely many possible candidate programs if programs can be of any length. Thus, we need to set a maximum size on the length of candidate programs. To begin, assume that the system is told exactly how many statements are needed in the candidate program.

Given a user-specified number of inputs, the input part of a candidate program is fixed. Moreover, the output part is always fixed because it has only one print statement. Therefore, both of them can be generated easily. Compared to these two parts, the statement part is harder to be generated because it can include any number of statements of any type. Therefore, this step mainly focuses on how to fill in the statement part.

First of all, to generate the statement part, we have to know the number of statements. The number of statements decides the number of possible permutations. For example, if there is only one statement in the statement part, then there will be 3 possible choices of statements in the search space: <linear>, <sqrt> and <if>. If there are two statements, then there will be eight possible permutations in search space: {linear, linear}, {linear, sqrt}, {linear, if}, {sqrt, linear}, {sqrt, sqrt}, {sqrt, if}, {if, linear}, {if, sqrt}. In this example, you will notice that there is one permutation missing, which is {if, if}. We exclude this for practical reasons: from experiments detailed in section 4, we found that the synthesizer goes really slow if we allow multiple if-else statements. This issue will be discussed further in the result section 4.3.1. So, we do not consider the permutations with multiple if-else statements. This does limit our results as there are certain programs that we will never find. However, it can help us to maintain the efficiency of the SynpleTest. Moreover, the programs we can find are complicated enough for Python beginners. For example, the program in Figure 3.2, which calculates the absolute difference between the two user inputs, is complicated for Python beginners.

Actually, the size of the search space is even larger than we listed. There are many options for chosen variables. For example, the eight permutations of two statements do not represent all possible programs in the search space. There are actually more, but it depends on the number of user inputs. For example, Figure 3.8 shows all of the possible options with one statement and two user inputs. If we only consider the type of statement, as we mentioned in the previous paragraph, there will be only three possibilities, <linear>, <sqrt> and <if>, in the search space. However, if we take the number of user inputs into account, as seen below, the size of search space will grow to 11.

```
x = int(input())
y = int(input())
```

$$\text{1 statement}\begin{cases} & linear \begin{cases} z = x + y \\ z = x + x \\ z = y + y \end{cases} \\ & square\ root \begin{cases} z = \sqrt{x} \\ z = \sqrt{y} \end{cases} \\ if - else & \begin{cases} if(x \geq 0) \begin{cases} if: z = x + y \mid else: z = x - y \\ if: z = x + x \mid else: z = x - x \\ if: z = y + y \mid else: z = y - y \end{cases} \\ if(y \geq 0) \begin{cases} if: z = x + y \mid else: z = x - y \\ if: z = x + x \mid else: z = x - x \\ if: z = y + y \mid else: z = y - y \end{cases} \end{cases} \end{cases}$$

```
print(z)
```

FIGURE 3.8

Figure 3.8 shows all 11 possibilities. For linear statements, the two variables on the right side of "=" can be x + y, x + x, or y + y. The square root statement can either be the square root of *x* or the square root of *y*. For if-else statements, the variables of the linear statements in the if branch and else branch are the same; the only difference is how they calculate the two variables. Since there are three possible choices for the linear statement, there are also three choices for if and else branches. Additionally, since there are two different conditions associated with those three choices, there are six possible choices for the if-else statement in total.

One thing that needs to be noted here is that when we actually run the SynpleTest system from the user interface, users don't have to decide the number of statements in the statement part because the system will determine this automatically. To be more specific, the synthesizer will try to find a solution with one statement. If it could not find a solution, then it will automatically start to find the solution with two statements and so on until it reaches a maximum number of statements. In theory, the SynpleTest doesn't have an upper bound for the maximum number of statements. However, in the current implementation, the maximum number of statements is set to be five. Again, this is not anything theoretically essential to the system; five is just determined through experiments to be a satisfactory number.

Many questions and details must still be discussed. Why must the linear statement have two variables on the right side of the "=" sign? Why can't it have one variable? Why can the if-else statement only have one linear statement in each branch? Why do the linear statements in both branches have the same defined and generated variables? And finally, why do *x*/*y* have to be bigger than or equal to 0 in the condition of the if-else statement? The answers to these questions will be answered in the following paragraphs, but before that let us introduce an important term: candidate code skeleton.

As the name suggests, a candidate code skeleton is an incomplete statement. It is important because it is the key to generate a complete statement. Figure 3.9 is an example of a candidate code skeleton.

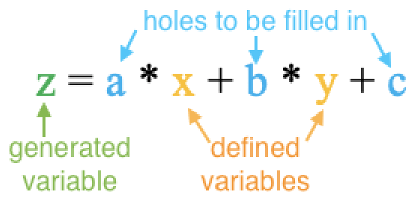
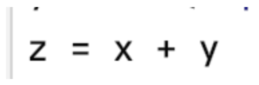


FIGURE 3.9 Linear Code Skeleton



```
z = x + y
```

FIGURE 3.10 Filled Linear Code Skeleton

The incomplete statement in Figure 3.9 is a skeleton because certain parts are not filled in. The green ***z***, the orange *x* and *y* are variables; they are the bones of the statement. In a code skeleton, they are filled in already. Compared to a complete statement, what we are missing in a code skeleton are the values of blue *a*, *b* and *c*. They are not variables; instead they are holes that need to be filled in as we complete the statement. Figure 3.10 is a filled-in linear code skeleton; notice that the holes have all been filled in by values: *a* and *b* are replaced by 1 and *c* is replaced by 0.

We named the variable on the left side of the "=" sign in Figure 3.9 the generated variable. It is a new variable generated by the statement. The variables on the right side of the "=' sign are defined variables, which means they were already defined in the former statements. In Figure 3.9, the orange *x* and orange *y* are defined variables. The holes in the skeleton to be filled in are coefficients. The blue characters *a* and *b* are coefficients of the defined variables and the blue character *c* is the constant. Our mission for candidate generation is to generate all possible permutations of candidate code skeletons in the statement part; these skeletons will have "holes" filled in section 3.3.5.

As we mentioned in section 3.2.2, SynpleTest now can only support three types of statements: linear, square root and if-else. This means the candidate code skeleton can only be permutations of these statement types based on the number of statements. The following are detailed introductions of these three statements.

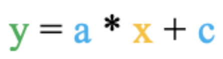
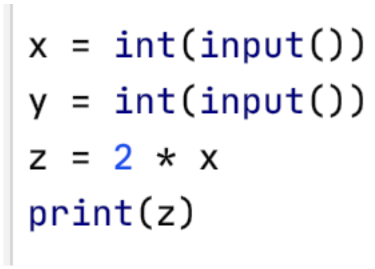


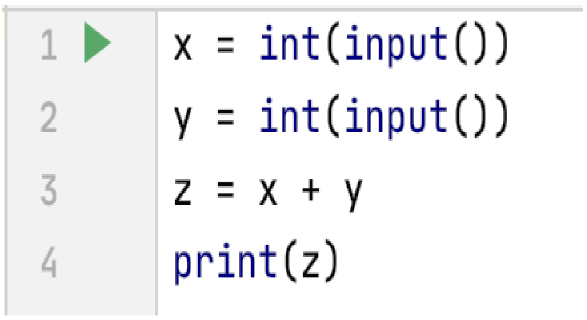
FIGURE 3.11



```
x = int(input())
y = int(input())
z = 2 * x
print(z)
```

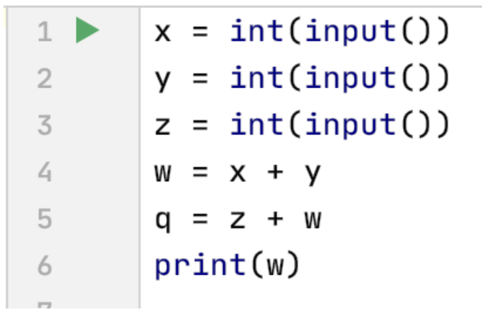
FIGURE 3.12

Linear Statement: Each linear statement in the candidate code skeleton has two defined variables and one generated variable. Figure 3.9 is a template of a linear statement skeleton. We designed the linear statement to have up to two defined variables because it can handle the problem which has more than two user inputs. For example, the problem in Figure 3.13 cannot be solved when linear statements only have one defined variable. If we only have one defined variable, when we have two user inputs and we want to add them up, there is no way to figure it out. As Figure 3.11 shows, remember that the blue a and c are holes that need to be filled in, and the values of them are fixed for all the test cases. Therefore, the value of the defined variable *x* can only be multiplied by a certain number, and the computation cannot involve any other variables. But when the program has more than one user input and we want to add them together, we cannot find a way to do that because the value of the second input is not fixed. For example, in Figure 3.12, the statement “z = 2 * x” would work for some inputs, for example, when *x* and *y* have the same value. In fact, the value of *y* is not always the same as the value of *x*; it can be any number. This is the restriction of the linear statements which only have one defined variable.



```
x = int(input())
y = int(input())
z = x + y
print(z)
```

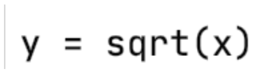
FIGURE 3.13



```
x = int(input())
y = int(input())
z = int(input())
w = x + y
q = z + w
print(w)
```

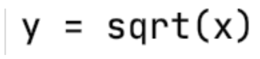
FIGURE 3.14

From the above example, we can see that the linear statement with one defined variable cannot add two user inputs together. However, if we have two defined variables on the right-hand side of the formula, this problem can be solved. As Figure 3.13 shows, when we have two user inputs, we can easily add them up by using the linear statement with two defined variables. Moreover, we can also represent many other computations involving three or more inputs. Figure 3.14 gives us an idea about how to add three user inputs together. We can add the first two user inputs, *x* and *y*, and assign the result to a new variable, *w*. Then, we add *w* to the last user input, *z*. In this way, we can handle problems with multiple inputs. Therefore, we decided to allow two defined variables in the linear statement.



```
y = sqrt(x)
```

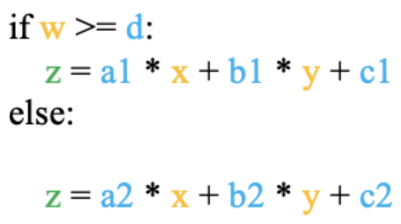
FIGURE 3.15 Square Root Skeleton



```
y = sqrt(x)
```

FIGURE 3.16 Filled in Square Root Skeleton

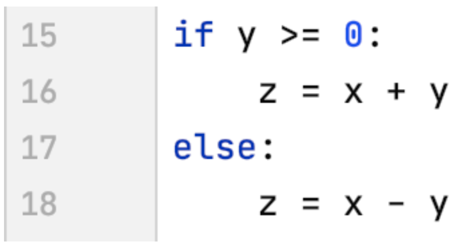
Square Root Statement: Compared to the linear statement, generating candidate code skeletons of square root statements is relatively simple. As shown in Figure 3.15, it only has one defined variable and one generated variable, and there are no holes to be filled in. Technically, as Figure 3.15 and Figure 3.16 show, the square root skeleton is identical to a complete square root statement.



```
if w >= d:
    z = a1 * x + b1 * y + c1
else:

    z = a2 * x + b2 * y + c2
```

FIGURE 3.17 If-Else Skeleton



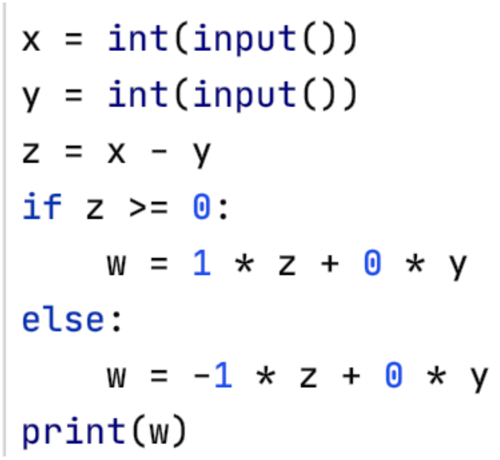
```
if y >= 0:
    z = x + y
else:
    z = x - y
```

FIGURE 3.18 Filled in If-Else Skeleton

If-Else Statement: Compared to linear and square root skeletons, if-else statement skeletons are more complicated. As Figure 3.17 shows, one if-else statement has two branches: the if branch and the else branch. Each branch has a linear statement. It also has a condition which is on the first line in Figure 3.17; we named the orange *w* in the condition as "condition variable". The condition variable can be any one of the defined variables. The blue *d* after the *w* is the hole we need to fill in within the condition. Just like *a1*, *b1* and *c1*, it is also an integer and can be any number. In the implementation, we set *d* to be a specific number, because if we don't do this, *d* can be any integer from negative infinity to positive infinity, and the search space will be extremely large. In order to cut down the size of the search space and save time for more meaningful searches, we decided to assign *d* a specific number. We noted that the values of *a*, *b* and *c* need not always be the same in the two branches; therefore we add suffixes to differentiate them. If-else skeletons are complicated because they have more variables to be generated and more holes to be filled in. Figure 3.18 is a complete if-else skeleton.

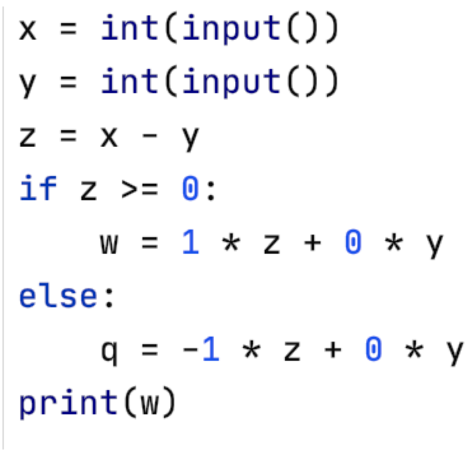
As you can see in Figure 3.18, both statements in the if branch (line 16) and the else branch (line 18) are linear statements. For now, our SynpleTest system only supports this limited form. Even if it can't do everything now, it is nevertheless able to express many programs relevant for Python beginners. For example, in Figure 3.19, this program calculates the absolute difference between *x* and *y*.

Another thing that needs to be noted here is that the statements of two branches must have the same defined and generated variables so that the later statements will not have undeclared variables. Taking Figure 3.20 as an example, *w* can be generated in the if branch and *q* can be found in the else branch. The variable that exists after this statement will depend on which branch you took. If *z* is less than 0, then the program will not reach the if branch. Thus, variable *w* will not be generated. However, in the following statement the program is supposed to print the variable *w*, and apparently *w* doesn't exist. If we keep the generated variable the same for both branches, e.g., Figure 3.19, we then don't need to worry about this problem. Therefore, to avoid the appearance of undeclared variables, all of the defined and generated variables in the two branches have to be the same.



```
x = int(input())
y = int(input())
z = x - y
if z >= 0:
    w = 1 * z + 0 * y
else:
    w = -1 * z + 0 * y
print(w)
```

FIGURE 3.19



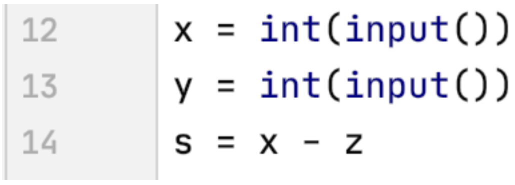
```
x = int(input())
y = int(input())
z = x - y
if z >= 0:
    w = 1 * z + 0 * y
else:
    q = -1 * z + 0 * y
print(w)
```

FIGURE 3.20

At this step, the synthesizer only generates all possible permutations of the candidate code skeletons. The algorithm we used to generate all candidate code skeletons is brute force, therefore the efficiency of our SynpleTest is limited; see section 4.3 for more information. We built a class called Statement to store them. For each Statement object in the Statement class, it stores the types of the candidate code skeleton, generated variables and defined variables.

3.3.4 Syntax Analysis

This step aims to filter all possible permutations of candidate code skeletons by checking if they are syntactically valid. There are two problems we need to figure out in this section: undeclared variables and SSA form. These two problems are also the criteria we need to check to determine whether the statements are syntactically valid. If all of the statements in the candidate program do not have undeclared variables and the candidate program itself is in SSA form, then it is syntactically valid by our definition.

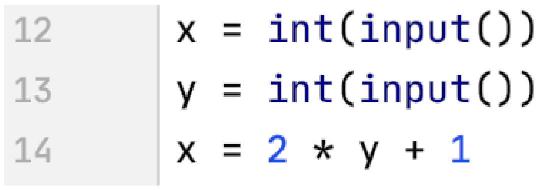


```
12    x = int(input())
13    y = int(input())
14    s = x - z
```

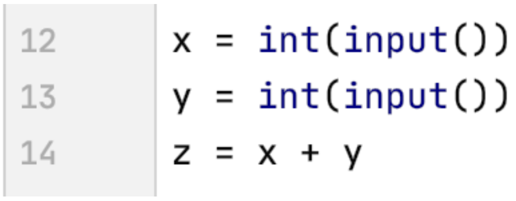
FIGURE 3.21

The first problem is if there are undeclared variables in the statement. Recall from section 3.3.3 that the variables on the left side of the "=" sign are generated variables, and the variables on the right side are defined variables. The first *x* on the 12th line in Figure 3.21 is a generated variable. However, when *x* shows in line 14, it is not a generated variable anymore; it is now a defined variable. An undeclared variable is a variable which has never been generated in the former statements but is used as a defined variable in a statement. For example, the last variable *z* on the 14th line in Figure 3.21 is undeclared: it is never generated in the first two statements, but it was used to generate a new variable *s*.

To solve this problem, we need to use a standard notion from compilers called a symbol table. According to Alfred V. Aho, who is the Lawrence Gussman Professor of Computer Science at Columbia University, "Symbol tables are data structures that are used by compilers to hold information about source program constructs. … Entries in the symbol table contain information about an identifier such as its character string, its type, its position in storage and any other relevant information" [2]. In SynpleTest, the symbol table is mainly used to avoid situations where undefined variables appear on the right side of the "=" sign. The symbol table we built in SynpleTest is simply a list that stores all of the defined variables, which are less than what the compiler stores. Every time we need to choose defined variables to generate a new variable, we will choose the defined variables from the symbol table so that the undeclared variables will never appear on the right side of the "=" sign. After generating a new variable, the newly generated variable now becomes a defined variable, and it will be added to the symbol table for future use.



```
12    x = int(input())
13    y = int(input())
14    x = 2 * y + 1
```



```
12    x = int(input())
13    y = int(input())
14    z = x + y
```

FIGURE 3.22 FIGURE 3.23

The second problem is that the programs should not break SSA form. Recalling from the background section, SSA form is "an intermediate representation that facilitates certain code optimizations" [2]. In our case, we need our code to be in the SSA form for the encoding we are going to attempt for Z3 in the following section. SSA form is essential to encode our programs as a mathematical formula to Z3 and fill in the holes of the programs. If we have statements which cause the program to break SSA form, then Z3 will get confused. For example, in Figure 3.22, variable *x* has two definitions: one is user input (line 12) and the other is the sum of "2y + 1" (line 14). As humans, we can tell that the *x* on line 12 is different from the *x* on line 14. However, Z3 is not as smart as us; it will end up asserting that both *x's* have the same value so that we cannot get the correct values for the holes.

To solve this problem, we have to verify that the whole candidate program is in SSA form and each statement never breaks it. If a variable is used as a generated variable in two or more statements in a program, this program then is not in SSA form. For example, in Figure 3.22 the variable *x* is first used as a generated variable in line 12. In theory, it should not appear on the left side of the "=" sign anymore. However, it is used as the generated variable one more time in line 14. Therefore, the program in Figure 3.22 breaks SSA form. Compared to the program in Figure 3.22, the program in Figure 3.23 doesn't break SSA form because it uses a new variable z to represent the value of x + y instead of reusing the variable *x*.

In reality, we simply need to count the number of times each variable appears on the left side of "=" to check for SSA form. If any variables are used more than once, the program is not in SSA form. Also, note that the syntax analysis need not actually be a separate step. Instead, in our implementation, we actually do this as a part of the candidate generation.

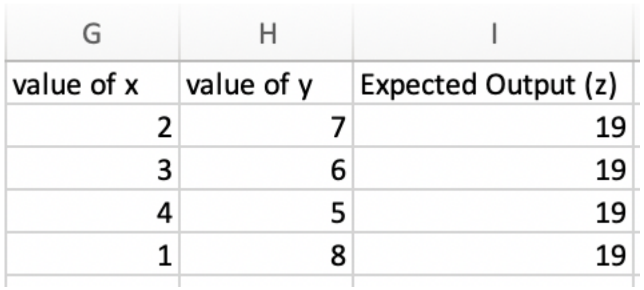
3.3.5 Code Skeletons Completion

This step aims to complete code skeleton using Z3 and produce the complete program for users. Recall from the background section that Z3 can help us figure out mathematical problems. It can fill in the holes for all of the possible candidate code skeletons we generated in section 3.3.3. As stated in the candidate generation section, there are several candidate code skeletons, and then in the syntax analysis section, we filtered them by checking whether they are syntactically valid. For each syntactically valid code skeleton, we will encode it into Z3. If we find the correct values to fill in the holes of the skeletons, we will put the values of the holes into the candidate code skeletons to generate a complete statement. If we go through all the skeletons and don't find the values to fit in the holes, it means our SynpleTest cannot generate a satisfying candidate program based on the test cases the user provided.

To fill in the holes of the skeletons, we first need to know the type of the skeleton: linear, square root or if-else. Then we can put the according candidate code skeletons into Z3. For example, if the type is linear, we will put the code skeleton from Figure 3.24 into Z3. We then will put the values of the generated variable and defined variables into Z3. The code skeletons and values of variables are called constraints in Z3. Once Z3 contains all of the constraints, it is able to solve the values of holes in the skeleton. The following example will go through this process.

$z = a * x + b * y + c$

FIGURE 3.24



G	H	I
value of x	value of y	Expected Output (z)
2	7	19
3	6	19
4	5	19
1	8	19

FIGURE 3.25

Suppose we are going to figure out the values of a, b and c in Figure 3.24, and there are some test cases provided in Figure 3.25. In Figure 3.24 the green *z* is the generated variable, and the orange *x* and *y* are defined variables. The rest—blue *a*, *b* and *c*—are coefficients and are also the holes in the skeleton that need to be filled in. When looking at the test cases in Figure 3.25, Columns G and H are values of defined variables *x* and *y*. Column I contains the values of expected output provided by users. They are values of the generated variable *z*. Each row is a test case.

The following figure can help us to have a better understanding of how we encode the test cases into Z3. Figure 3.26 is not code; it is just the mathematical description of what we do to encode the test cases into Z3. For more details about the code, see Appendix A.

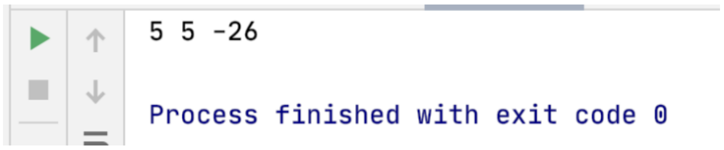
```
x1 == 2
y1 == 7
z1 == 19
a * x1 + b * y1 + c = z1

x2 == 3
y2 == 6
z2 == 19
a * x2 + b * y2 + c = z2

x3 == 4
y3 == 5
z3 == 19
a * x3 + b * y3 + c = z3

x4 == 1
y4 == 8
z4 == 19
a * x4 + b * y4 + c = z4
```

FIGURE 3.26



```
5 5 -26

Process finished with exit code 0
```

FIGURE 3.27

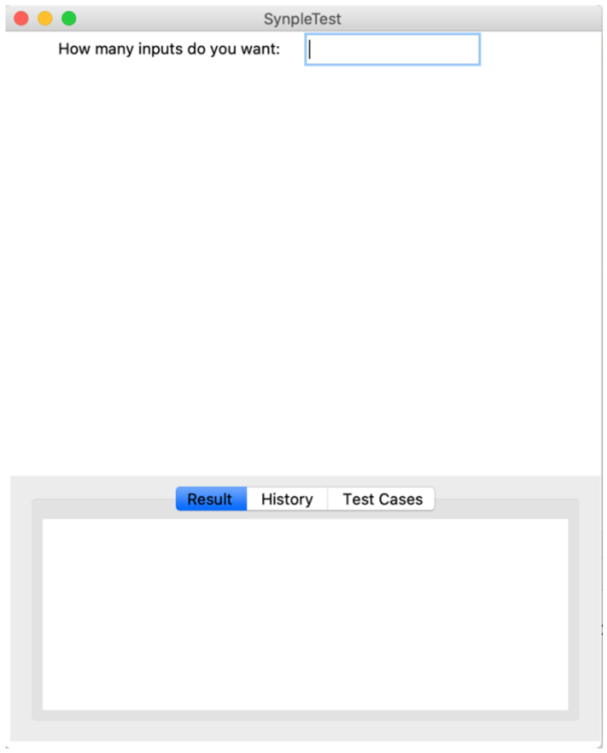
As we can see, each of the test cases in Figure 3.26 are shown in different colors. Also, in order to not break SSA form in Z3, we added a suffix to all generated variables and defined variables. For each test case, the suffixes of a generated variable and defined variables are the same.

In Figure 3.27, the values of a, b and c are 5, 5 and -26 accordingly, which means the complete statement is “5x + 5y - 26 = z”. However, “2x + 2y + 1 = z” and “x + y +10 = z” are also complete statements that satisfy the program. Z3 is amazing because it can give you a solution that you may never think of. For our use case, as long as the solution works, it doesn’t matter which one Z3 finally returns. This feature of Z3 can help us to identify good test cases, because if the test cases are hard to be differentiated from others, then they must not be good test cases.

To find the satisfying program, the synthesizer will go through all possible permutations of the candidate code skeletons and use Z3 to check if the encoding of that permutation has a solution. If Z3 finds a solution, the synthesizer will stop searching and complete the code skeletons, otherwise it will keep searching until it finds one. If the synthesizer goes through all the permutations and does not find a solution, it will give users an error message. Once Z3 finds a solution, it will be easy to complete the statement. As we mentioned at the end of candidate generation section, we created a Statement class for the code skeletons to store the type, defined variables and generated variables. Now we just need to modify them and fill in the holes with the values we got from Z3 to complete the statement and then write the complete statement into a Python file to generate a Python program.

3.4 User Interface

To make the SynpleTest easier for users, we designed a simple user interface. When users run SynpleTest, there will first be a pop-up window (Figure 3.28). It has an entry box in the top of the window for users to enter the number of inputs in each test case. At the bottom of the window, there is a rectangle containing three tabs which can show Result, which shows the current candidate program; History, which shows all the candidate programs that have been generated so far; and Test Cases, which shows the test cases that users have entered. After users enter the number of inputs, they have to hit the return button to continue. The entry box will be locked so that users cannot change it during the process as they enter more test cases. Then the other entry boxes will show up, as the Figure 3.29 shows.



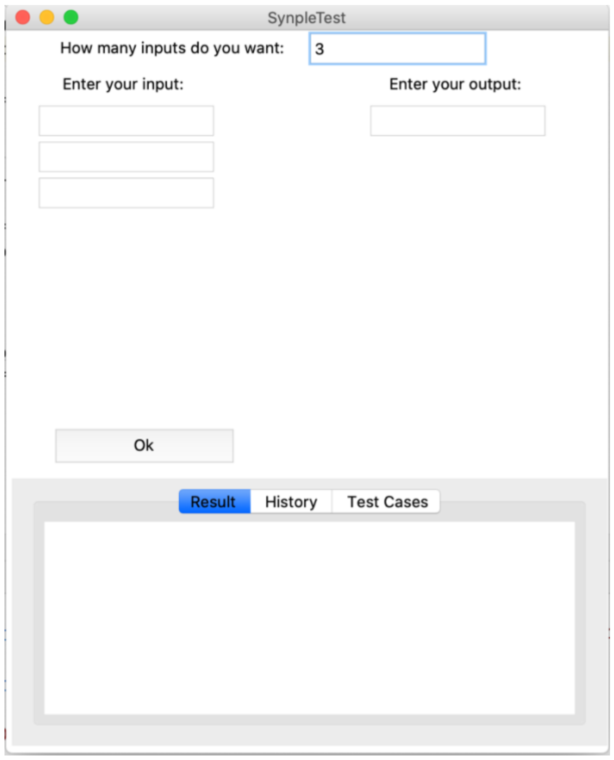
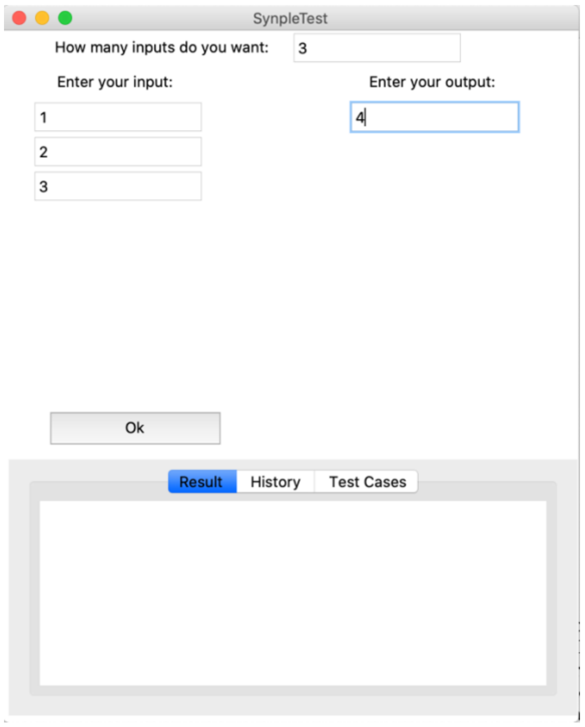
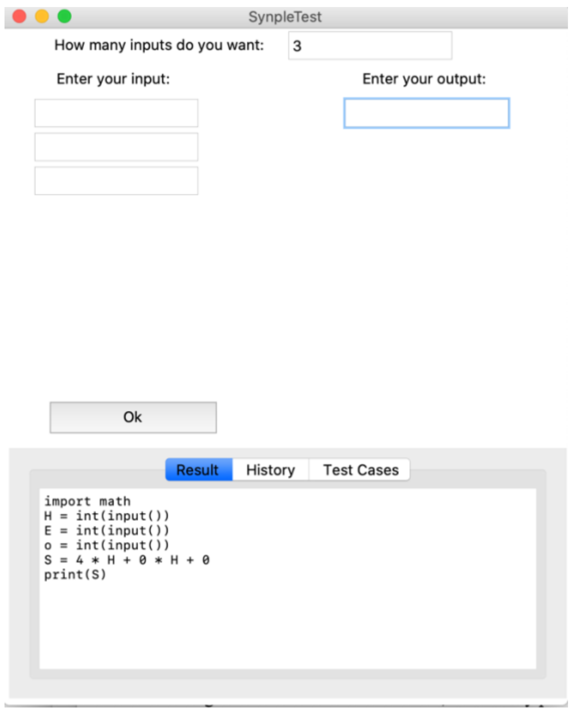


FIGURE 3.28 Initial User Interact Window FIGURE 3.29 After Hit Return Button

Under the top entry box, there are two columns (Figure 3.29). The left side of the column has several entry boxes for users to enter the inputs, and the number of entry boxes is decided by the number of inputs the users entered. For example, in Figure 3.29, one user entered the number three into the top entry box and hit the return button. Thus, the window listed three entry boxes on the left column. Since our SynpleTest only accepts one output, there is only one entry box for users to enter the expected output. There is an OK button under the two columns.

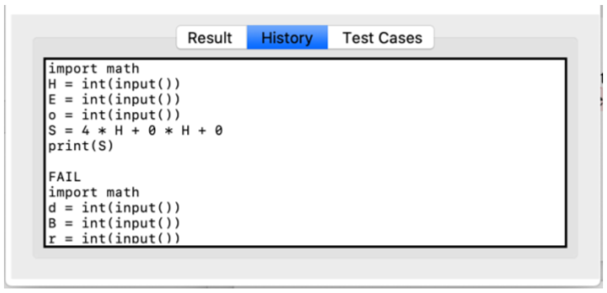




```
import math
H = int(input())
E = int(input())
o = int(input())
S = 4 * H + 0 * H + 0
print(S)
```

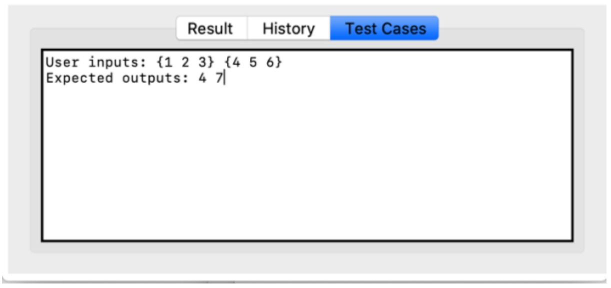
FIGURE 3.30 Enter the Test Case FIGURE 3.31 After Click OK Button

After finishing entering a test case (Figure 3.30), users have to click the OK button to continue the process. This causes the verifier to run on the current candidate program, which may result in running the synthesizer if the verifier gets a failed result. Every time users click the OK button, the entry boxes in the two columns will be cleared automatically so that users can enter a new test case. The current candidate program will be shown in the Result tab in the big rectangle at the bottom of the window (Figure 3.31). There are three tabs in the big rectangle. The first tab is the Result tab, and it will only print the code of the current candidate program. Whenever a new test case fails to satisfy the current candidate program, the user interface will update the new candidate program for users in this tab. If users want to know how candidate programs change as they keep providing test cases, they can switch to the second tab (Figure 3.32). It contains all the candidate programs which SynpleTest had generated. The last tab is the history of test cases (Figure 3.33), and users can click this tab to track the test cases they have entered.



```
import math
H = int(input())
E = int(input())
o = int(input())
S = 4 * H + 0 * H + 0
print(S)

FAIL
import math
d = int(input())
B = int(input())
r = int(input())
```



```
User inputs: {1 2 3} {4 5 6}
Expected outputs: 4 7
```

FIGURE 3.32 History Tab FIGURE 3.33 Test Cases Tab

4. Experiments

4.1 Set Up

All experiments used a dual-core Intel Core i5 (2.3 GHz) with 8 GB of RAM running Mac OS. We developed and ran all of the experiments within the PyCharm IDE. All experiments in this section did not use the user interface, and instead ran from the synthesizer directly for accurate timing.

Throughout this section, the following research questions will be addressed. Q1 is about the product of the system. Q2 addresses scalability based on configurations for the system. While doing the experiments for Q2, we noticed that the number of if-else statements generated in the candidate generation step also affects the efficiency of the system. Therefore, we raised Q3. At last, Q4 addresses the input size provided by the user.

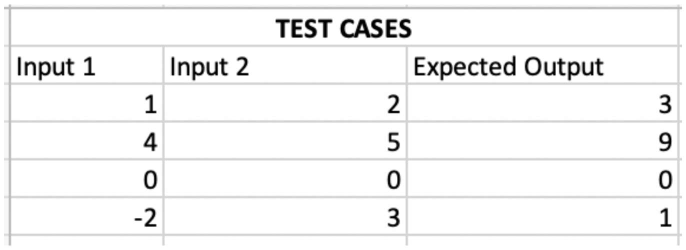
Q1: What kind of candidate programs can be generated by SynpleTest?
Q2: How will the number of statements in the statement part influence the efficiency of the SynpleTest?
Q3: How will the number of if-else statements influence the efficiency of the SynpleTest?
Q4: How will the number of inputs influence the efficiency of the SynpleTest?

4.2 Candidate Program Generation

As we mentioned in section 3.2.2, a candidate program has three parts: input part, statement part, and output part. The input part only has input statements, and the number of them will be decided by users when SynpleTest starts to run. For example, if users want to have two inputs then there will be two input statements in the input part. Accordingly, users have to provide two inputs for each test case. The statement part contains statements with types of linear, square root or if-else. The output part only has one print statement. Currently, the system supports algorithms that use any permutation of linear, square root, and if-else statements using integers for inputs and outputs.

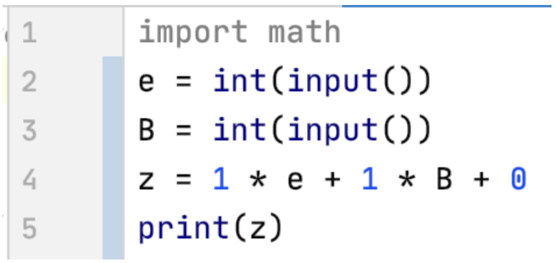
Here are some sample candidate programs with one, two and three statements in the statement part and the test cases that were used to generate them. For each sample candidate program, we set the number of test cases to be four in order to make sure that the synthesizer has enough test cases to generate a satisfying candidate program.

Candidate Program 1:



TEST CASES		
Input 1	Input 2	Expected Output
1	2	3
4	5	9
0	0	0
-2	3	1

FIGURE 4.1.

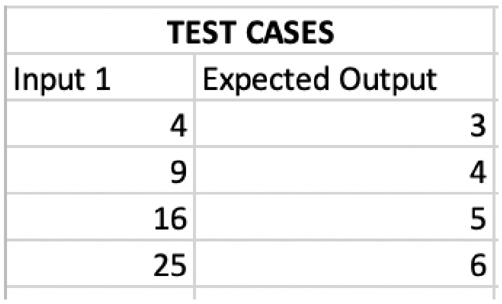


```
import math
e = int(input())
B = int(input())
z = 1 * e + 1 * B + 0
print(z)
```

FIGURE 4.2

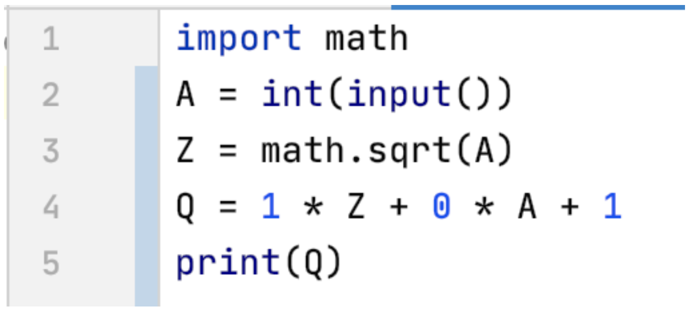
This candidate program has two user inputs and one statement in the statement part. Figure 4.2 is the candidate program generated when we actually run SynpleTest based on the test cases in Figure 4.1. This candidate program adds two user inputs together and prints the sum.

Candidate Program 2:



TEST CASES	
Input 1	Expected Output
4	3
9	4
16	5
25	6

FIGURE 4.3

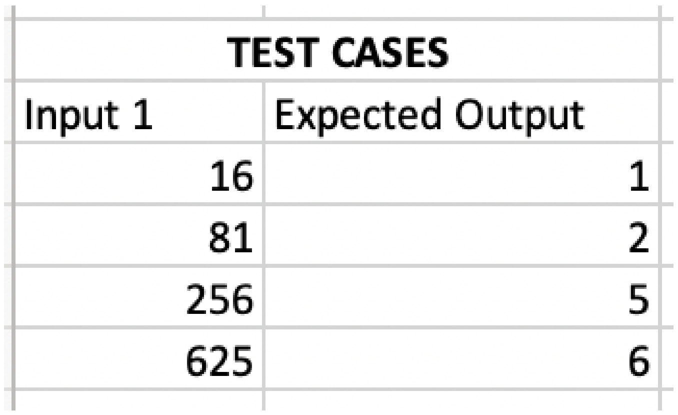


```
import math
A = int(input())
Z = math.sqrt(A)
Q = 1 * Z + 0 * A + 1
print(Q)
```

FIGURE 4.4

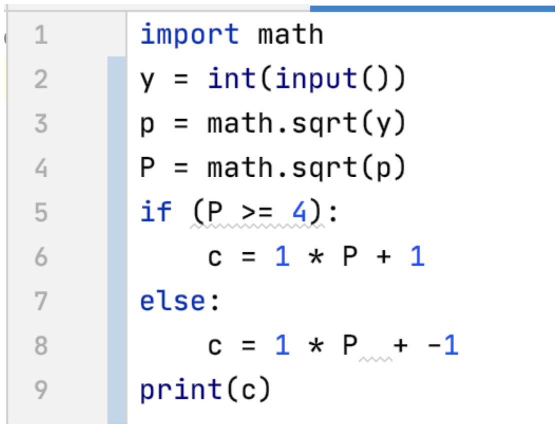
This candidate program has one user input and two statements in the statement part. Figure 4.4 is the candidate program generated when we actually run SynpleTest based on the test cases in Figure 4.3. The candidate program takes the square root of the user input first and then adds 1 to the result of the square root.

Candidate Program 3:



TEST CASES	
Input 1	Expected Output
16	1
81	2
256	5
625	6

FIGURE 4.5



```
import math
y = int(input())
p = math.sqrt(y)
P = math.sqrt(p)
if (P >= 4):
    c = 1 * P + 1
else:
    c = 1 * P  + -1
print(c)
```

FIGURE 4.6

This candidate program has one user input and three statements in the statement part. Figure 4.5 is the candidate program generated when we actually run SynpleTest based on the test cases in Figure 4.6. The candidate program takes the square root of the user input first; it then takes the square root of the former result. If the value of the second square root is bigger than or equal to 4, it adds one to the second result. Otherwise, it subtracts one from the result.

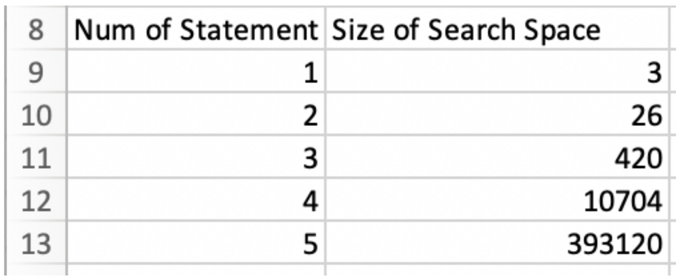
4.3 Efficiency

The following sections present some experiments to answer the research questions Q2, Q3 and Q4. Timing is a great way to evaluate the efficiency of our system, therefore, we will use time to represent the efficiency in the following subsections. Shorter time means higher efficiency, while longer time means lower efficiency.

4.3.1 Number of Statements

In this section, we are going to figure out how the efficiency will be affected if the number of statements in the statement part changes through a set of experiments.

When we do experiments on what candidate programs can be generated, we noticed that when the candidate program has more statements in the statement part, the time it takes to generate the candidate program is longer. Therefore, we inferred that the number of statements is a factor that impacts the efficiency of our SynpleTest system. Based on our inference, we did a set of experiments and collected the results. For this set of experiments, their number of inputs and the number of test cases are fixed.



8	Num of Statement	Size of Search Space
9	1	3
10	2	26
11	3	420
12	4	10704
13	5	393120

FIGURE 4.7

In Figure 4.7, as the number of statements increases, the size of the search space increases exponentially. It should be recognized that the size of the search space cannot help us to know how efficient the SynpleTest is. Therefore, we ran a set of experiments with one user input and four test cases, and recorded the running time. We have to mention that all of them have no solution. We don't want to have a solution in this experiment because then we can run through the whole search space and get an upper bound on the time for these experiments. If the test cases have a solution, we might randomly find it on the very first try, making the time we measure a representation of random chance, rather than a uniform measure of how long a user can expect it to take. We only have one user input because we want to minimize the effect of the number of inputs since it is also a factor for the size of the search space. This makes sure that the changes are mainly caused by the number of statements. Also, we minimize the number of inputs so that the number of statements can be maximized. We chose to enter four test cases because it is a very fair number for candidate program generation. It is also a reasonable number that a student might provide on the first attempt. If the number of test cases is smaller, the constraints on the candidate program is less. It will be easier for SynpleTest to find a solution which satisfies the test cases but is not what users want. If the number of test cases is big, then it will dilute the impact of the number of statements on the change of the size of the search space. Therefore, we chose to enter four test cases. The result is shown in the following chart.

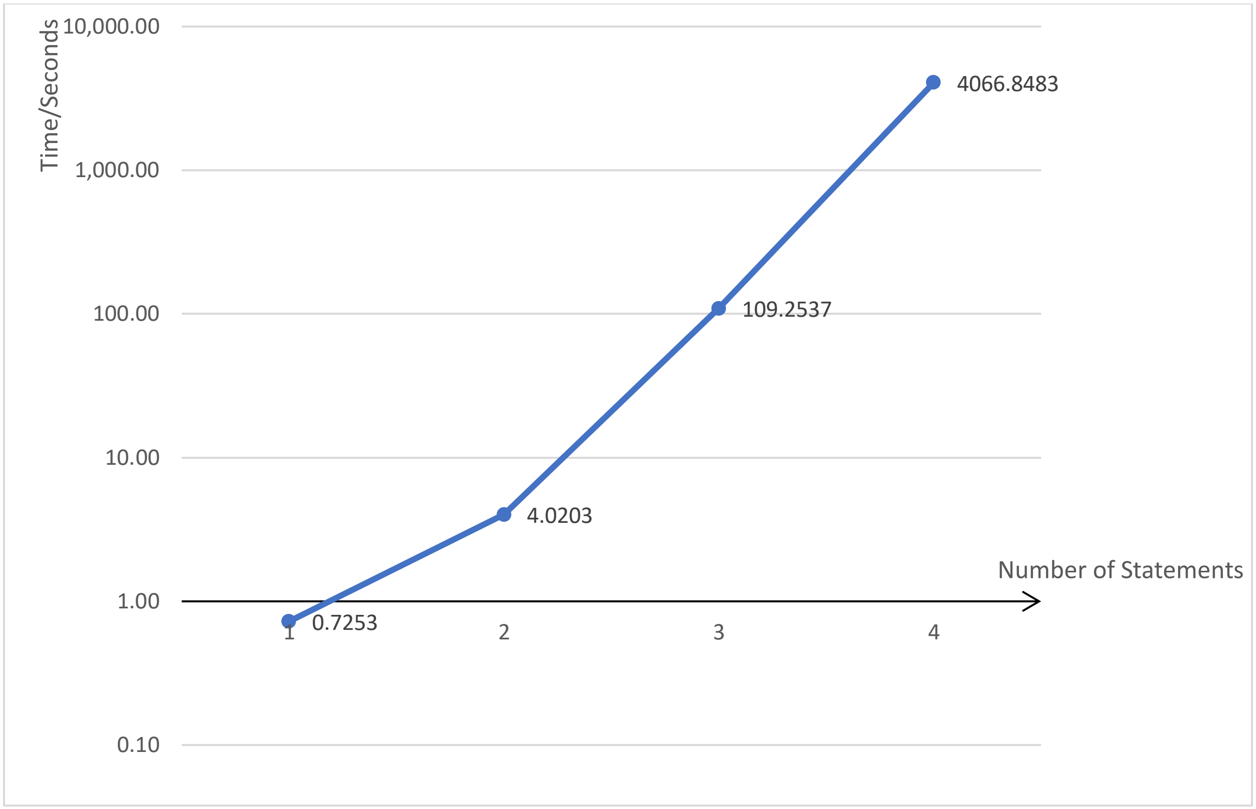


FIGURE 4.8

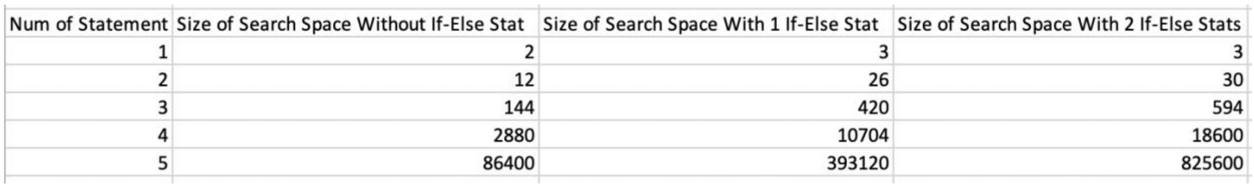
In Figure 4.8, the x-axis represents the number of statements and the y-axis represents time spent. To make our data more impartial, we ran each experiment three times and took the average. We used a log scale chart to represent the results because if the values grow exponentially, then the points that the values represent in the log scale chart will be represented as a straight line. It also can help readers see the changes of time more directly. The line in Figure 4.8 is very close to a straight line, which means as the number of statements grows, the time SynpleTest used grows exponentially.

From Figure 4.7 we can learn that the size of the search space of five statements is 36.7 times that of four statements. From the chart we can learn that it took about 4067 seconds, which is about 68 minutes, to run through all the search space when the number of statements is four. If this trend continues, we can speculate that running through the search space for five statements will approximately take 68 * 36.7 = 2495.6 minutes, which is about 42 hours. It is unrealistic that users would be willing to spend 42 hours waiting for our SynpleTest to give them an answer. Therefore, the best time to stop doing experiments is when the number of statements is four.

In conclusion, our SynpleTest can generate candidate programs with any number of statements literally. However, from the experiments we learn that it will take more than 1 day to generate a candidate program with more than five statements. Therefore, we can conclude that the SynpleTest system without any further modifications is only efficient for generating programs with four statements or less.

4.3.2 Number of If-Else Statements: 0 vs 1 vs 2

In section 3.3.3, we mentioned the fact that the synthesizer goes really slow if we allowed multiple if-else statements. We found this fact because when we looked closer at the data from section 4.3.1, we see that Z3 solve time for if-else statements is particularly long, Therefore, this leads us to our third research question: how will the number of if-else statements influence the efficiency of the SynpleTest? Here the number of if-else statements refers to when we generate the candidate programs how many if-else statements they can have. 0 if-else statement means the candidate program doesn't have any if-else statements; 1 if-else statement means the candidate program has no more than 1 if-else statement no matter how many statements it has in total; the same applies for 2 if-else statements.



Num of Statement	Size of Search Space Without If-Else Stat	Size of Search Space With 1 If-Else Stat	Size of Search Space With 2 If-Else Stats
1	2	3	3
2	12	26	30
3	144	420	594
4	2880	10704	18600
5	86400	393120	825600

FIGURE 4.9

To test how the number of if-else statements will influence the efficiency of the SynpleTest, we experimented with the candidate programs with 0, 1 and 2 if-else statements. The above table shows the size of search space for each case and how the search space will grow as the number of inputs grows. From the table above we can know that, when the number of test cases is constant, the size of the search space in two if-else statements is larger than that in one if-else statement. The size of the search space in one if-else statement is larger than that in zero if-else statement. Like what we did in section 4.3.1, to make our data more impartial, we ran each experiment three times and took the average. To test the upper bound of the time it can spend. All test cases we provided for each experiment don't have a solution. We still provided 4 test cases because it is a fair number, and each test case has only one user input because we want to minimize the effect of the number of user input. We recorded the time for each experiment and the result is shown in the following chart.

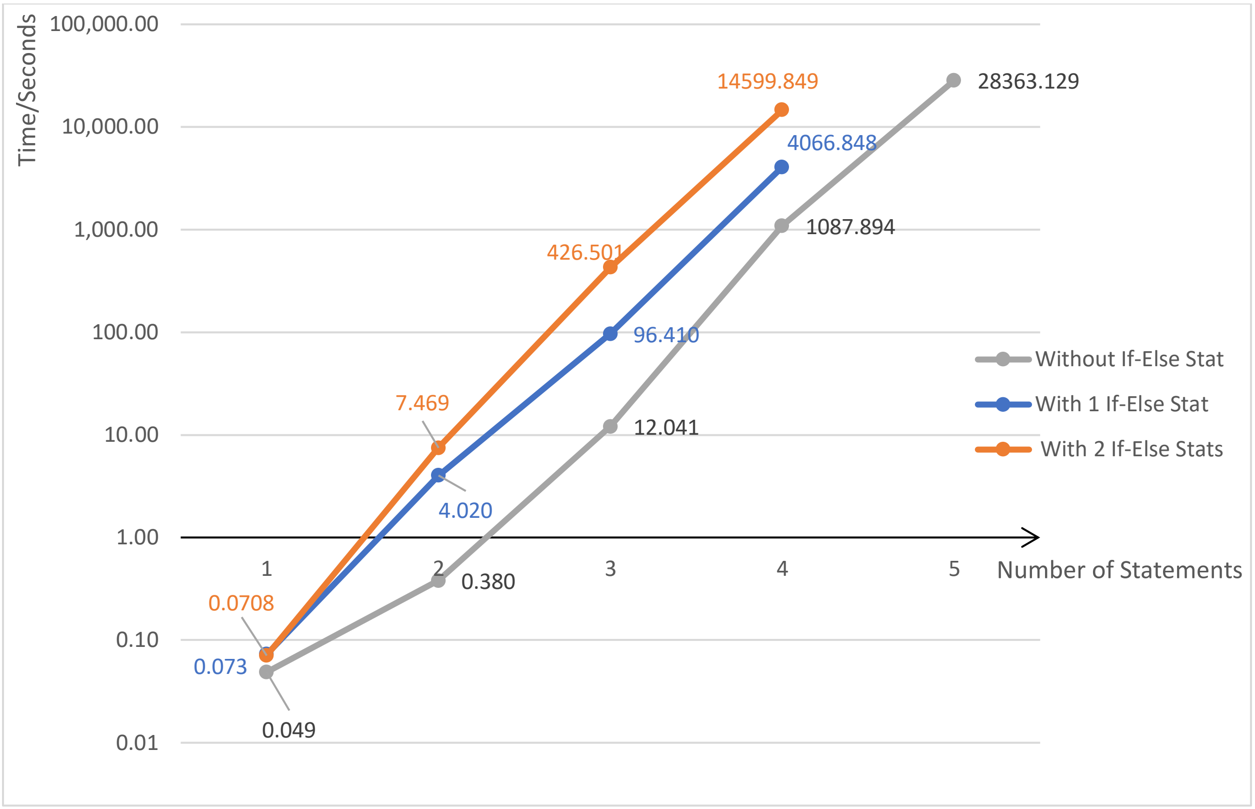


FIGURE 4.10

Again, to let readers see the changes more directly, we used a log scale chart. The x-axis represents the number of statements and the y-axis represents the time spent. The grey line represents the time spent on candidate programs without if-else statements; the blue line is for the time spent on candidate programs with one if-else statement, and the orange line represents the time spent on candidate programs with two if-else statements. We can learn from the chart that the candidate programs with two if-else statements took the most time. For example, when we have three statements in the statement part, the amount of time to generate a candidate program with three statements increases from 12 seconds with no if-else statements to 400 seconds with two if-else statements. We stop collecting data when the number of statements reaches four for them. While candidate programs without if-else statements took the least time: it only takes about 18 minutes. Therefore, we didn't stop collecting data until the number of statements reaches five. The time that the candidate programs with one if-else statement are in the middle. We stop collecting data of them because we think the time we spent is pretty long and there is no need to do more experiments which will take longer.

In conclusion, when the candidate programs have if-else statements, it will take a longer time to find solutions. If the number of if-else statements is more, our SynpleTest would spend longer time while finding the solution, and the efficiency is lower.

4.3.3 Number of Inputs

As we developed our system, we realized that the number of user inputs in the test cases can also affect the efficiency of the SynpleTest. In section 4.2, we set the number of user input to one to minimize the influence of the number of inputs on the size of search space. But we still don't know what will happen to the size of search space if the number of inputs changes, so we experimented to answer this question, which is also Q4.

In this section, we did a set of experiments for the candidate programs which doesn't generate if-else statements, because from the last section we concluded that the number of if-else statements can affect the size of search space. We choose to do so in order to minimize the effect of the number of if-else statements and maximize the effect of user inputs. We chose to show the result of 3-statement candidate programs because the size of search space of 1-statement programs is too small and the size of search space for programs which have more than 3 statements is too big. For each experiment, we provided 4 test cases again as we did in sections 4.3.1 and 4.3.2.

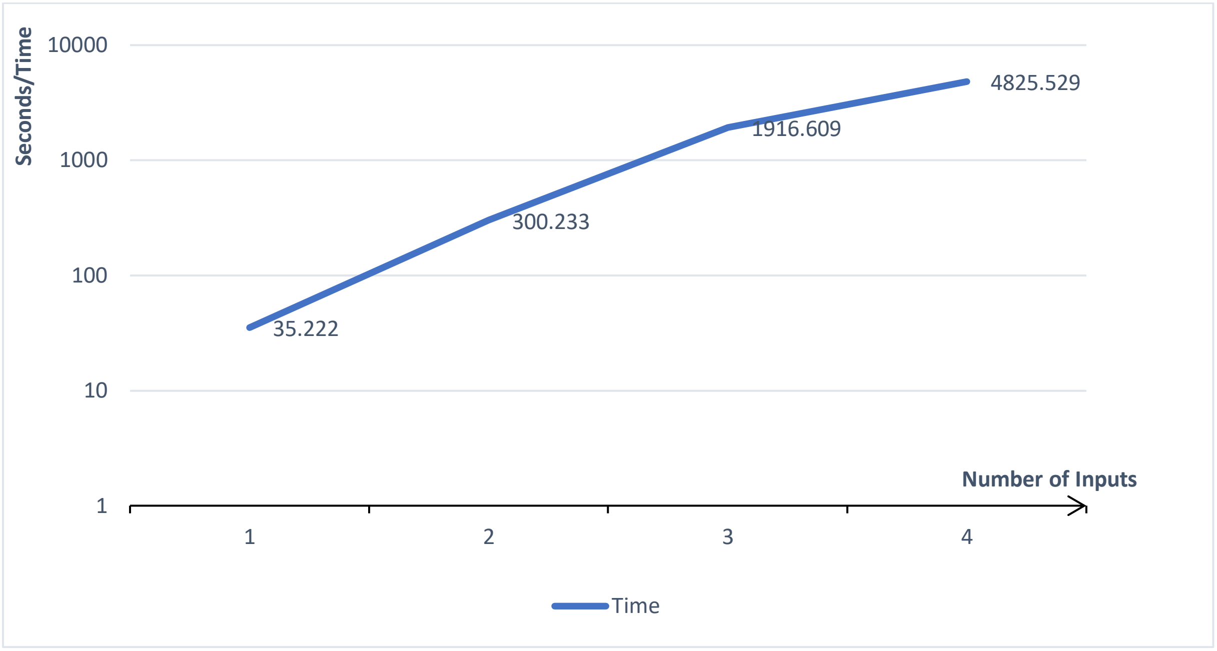


FIGURE 4.11

Again, we chose to use a log scale chart to represent the changes in time. As we can learn from Figure 4.11, as the number of inputs grows, the time SynpleTest spent grows exponentially. However, this line is not as steep as the other lines in the former sections, this line is pretty flat. For example, in this case, going from 3 to 4 inputs approximately doubles the time, but in Figure 4.8, the time increased by about 37 times going from 3 to 4 statements. This means the number of inputs can affect the efficiency of the SynpleTest system but compared to the number of statements and number of if-else statements, the efficiency is less affected by it.

5. Contribution

Program synthesis is not rare and using program synthesis to write programs automatically is also not rare. However, using program synthesis to teach testing in introductory computer science classes is, to our knowledge, novel to this work. We chose to do this project because there are few tools like SynpleTest to help computer science beginners focus on testing. When CS beginners start to learn coding, they care more about how to write code correctly. If their code compiles and passes the few tests the instructors provide, they are satisfied. But when we actually test their program with more test cases, their code may not work anymore. Using SynpleTest can force them to think more about testing and build a good sense of testing because they only need to provide good test cases, and they do not need to worry about all of the details to create working code. This connection between computer science education and program synthesis is our biggest contribution to this field.

There are some other contributions we made: the user interface and the results of our experiments. We developed a user interface for SynpleTest so that it will be easier for users to use. The user interface has specific instructions, so users know what is the next thing to do. The user interface also has tabs containing the code of the current candidate program, the history of candidate program generation and test cases entered so that it is convenient for users to track the changes of the candidate program. Overall, the user interface allows novice users to take advantage of our system. Our experiments show the current possibilities and limits of our enumerative synthesis approach to solving this problem.

6. Conclusion and future work

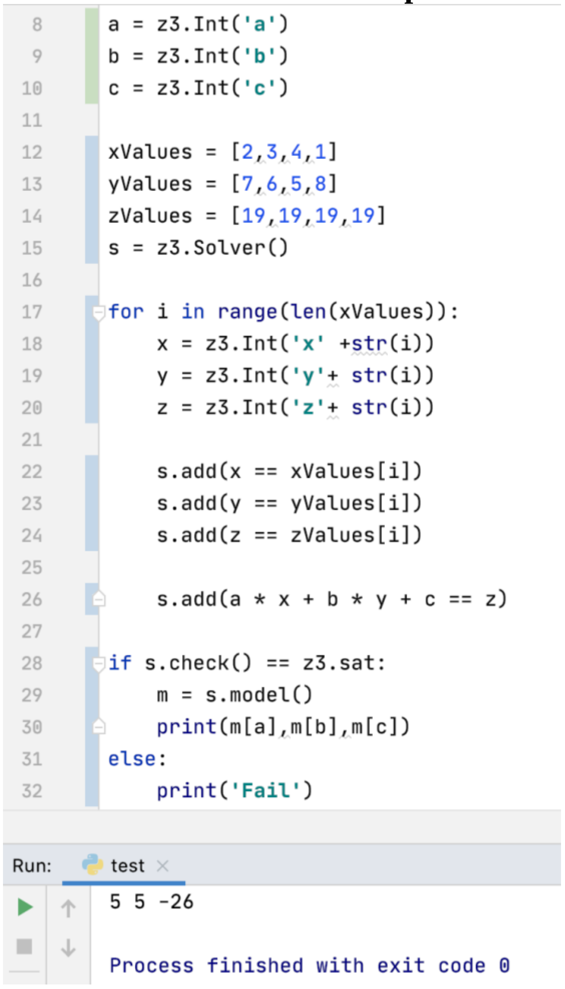
We introduced our SynpleTest system, which can help Python beginners to enhance their understanding of testing. We provided readers with some background information. Moreover, we discussed the structure and important artifacts (e.g., the verifier and synthesizer) of the SynpleTest System; three steps we used to build the system: candidate generation, syntax analysis and code skeletons completion and how efficient it is.

Our SynpleTest System can handle any program containing any number of linear, square root and if-else statements. However, in order to maintain the high efficiency of the system, we suggest users use it to generate a program which has no more than four statements in the statement part. Our system also can generate any programs with any number of user inputs, but for the same reason again we suggest users provide no more than four user inputs for test cases. Because the number of if-else statements can impact the efficiency of the system, we decided that the candidate programs that our system generated can only have one if-else statement. This limits our candidate program generation, but the generated programs are complicated enough for Python beginners.

In the future, we will keep exploring more types of statements such as quadratic statements, cubic statements, loops and so on. We will also keep working on expanding the scope of data type besides integers, such as fractions and decimals. Improving the efficiency of our system is another future goal. Now the algorithm we used to generate candidate programs are reasonably straightforward, it is brute force. In the future, we may use neural networks to generate candidate programs so that the efficiency of candidate generation can be improved. Besides, we might also be able to use larger compute resources (e.g., a computing cluster) in the future to test multiple candidate programs simultaneously.

7. Appendices

A. Code of Skeleton Completion



```
8   a = z3.Int('a')
9   b = z3.Int('b')
10  c = z3.Int('c')
11
12  xValues = [2,3,4,1]
13  yValues = [7,6,5,8]
14  zValues = [19,19,19,19]
15  s = z3.Solver()
16
17  for i in range(len(xValues)):
18      x = z3.Int('x' +str(i))
19      y = z3.Int('y'+ str(i))
20      z = z3.Int('z'+ str(i))
21
22      s.add(x == xValues[i])
23      s.add(y == yValues[i])
24      s.add(z == zValues[i])
25
26      s.add(a * x + b * y + c == z)
27
28  if s.check() == z3.sat:
29      m = s.model()
30      print(m[a],m[b],m[c])
31  else:
32      print('Fail')
```

```
5 5 -26

Process finished with exit code 0
```

Figure A.1

Line 8, 9 and 10 in Figure A.1 first starts to create three integer variables in Z3 named a, b and c to represent the blue a, b and c in Figure 3.3.8. We then put the values of *x*, *y* and *z* into three lists separately, as lines 12,13 and 14 show. In line 15, we create a Solver named s. "The function Solver () creates a general-purpose solver" [z3]. We then create a loop to add all the values of variables. As we mentioned in section 3.3.4, it has to be only one assignment for each variable, but we need to enter multiple values of *x*, *y* and *z*. Therefore, when we add test cases to Solver, we have to create new variables for *x*, *y* and *z*, line 18, 19 and 20 created new variables by adding a suffix to their name. Because we have a new variable name for each test case, we have to update the skeleton to the solver again, as line 26 does in the loop. Every time we add a constraint into the solver, it has been asserted in the solver. And the check() function in line 28 solves the asserted constraints. If the result is "sat", satisfiable, it means there is a solution. If there is no solution, the result will be unsat, unsatisfiable. If the result is "unknown", it means the solver fails to solve a system of constraints. In line 29 we used a new function named model(). In Z3, the solution of a solver "is a model for the set of asserted constraints. A model is an interpretation that makes each asserted constraint true." [z3]. In other words, a model only exists when there is a solution for the solver. The expressions m[a], m[b] and m[c] return the interpretation of *a*, *b* and *c* in the model m. The interpretations of them are also the values of holes that we want to fill in in the skeleton. In the program above, the values of *a*, *b* and *c* are 5, 5, -26 accordingly.

8. References

[1] Abate Alessandro, David Cristina, Kesseli Pascal, Kroening Daniel, Polgreen Elizabeth. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification*, 2018, pp. 270–288., doi:10.1007/978-3-319-96145-3_15.

[2] Alfred Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. Compliers, Principles, Techniques, and Tools. In *Pearson Education*, 1986.

[3] Alur, Rajeev, et al. "Syntax-Guided Synthesis." In *2013 Formal Methods in Computer-Aided Design*, 2013, doi:10.1109/fmcad.2013.6679385.

[4] G. M. Schneider. The introductory programming course in computer science – ten principles. In *Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education*, 1978, pp. 107–114.

[5] James Bornholt. Program Synthesis Explained. *www.cs.utexas.edu/~bornholt/post/synthesis-explained.html*.

[6] Marrero, Will, and Amber Settle. "Testing First: Emphasizing Testing in Early Programming Courses." *ACM SIGCSE Bulletin*, vol. 37, no. 3, 2005, pp. 4–8., doi:10.1145/1151954.1067451.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, 1991, pp. 451–490., doi:10.1145/115372.115320.

[8] Subprocess - Subprocess Management. *Subprocess - Subprocess Management - Python 3.9.2 Documentation*, docs.python.org/3/library/subprocess.html.

[9] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, 2010, pp. 1.

[10] Sumit Gulwani, Oleksandr Polozov and Rishabh Singh. Program Synthesis. *Now*, 2017.

[11] Z3 API in Python. *Z3Py Guide*, ericpony.github.io/z3py-tutorial/guide-examples.htm.