Olympiads in Informatics, 2016, Vol. 10, 111–124

DOI: 10.15388/ioi.2016.08

Examples of Algorithmic Thinking in Programming Education

Juraj Hromkovič, Tobias Kohn, Dennis Komm, Giovanni Serafini
Department of Computer Science, ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
e-mail: {juraj.hromkovic, tobias.kohn, dennis.komm, giovanni.serafini}@inf.ethz.ch

Abstract. Algorithmic thinking and problem solving strategies are essential principles of computer science. Programming education should reflect this and emphasize different aspects of these principles rather than syntactical details of a concrete programming language. In this paper, we identify three major aspects of algorithmic thinking as objectives of our curricula: the notion of a formal language to express algorithms, abstraction and automation to transfer proven strategies to new instances, and the limits of practical computability.

The primary contribution of this paper are three examples that illustrate how general aspects of algorithmic thinking can be incorporated into programming classes. The examples are taken from our teaching materials for K-12 and university non-majors and have been extensively tested in the field.

Keywords: algorithmic thinking, K-12, spiral curriculum, programming education, Logo, Python.

1. Introduction

Algorithmic thinking constitutes one of the core concepts of computer science. It has proven a versatile and indispensable tool for problem solving and found applications far beyond science. Hence, sustainable computer science education should be built upon algorithmic thinking as its primary objective, thus unfolding benefits for a broad and general education. However, how do we bring algorithmic thinking to computer science education? In this paper, we identify a number of principles that we want to deliver to students at different levels. As the main contribution, we describe concrete examples of how to teach these paradigms, which have been proven successful in the past.

Our work is part of ubiquitous efforts towards establishing sustainable computer science in K-12 education. Particularly noteworthy and inspiring are "CS unplugged" approaches as proposed by Bell *et al.* or Gallenbacher, which do completely away with computers and solely focus on the underlying algorithmic principles (Bell *et al.*, 2012; Gallenbacher, 2008). By incorporating such ideas into programming education, we effectively combine the strengths of the two approaches, resulting in a truly sustainable education.

1.1. *The Setting*

The examples presented in this paper stem from teaching materials we have developed for primary school, high school, and university, respectively (Gebauer *et al.*, 2016; Böckenhauer *et al.*, 2015a; Böckenhauer *et al.*, 2015b; Kohn, 2016). The goal of our endeavours is to create a spiral curriculum that starts as early as fifth grade in primary school with iterations throughout mandatory school, and including computer science classes for non-majors at university level.

We use both Logo and Python in our classes and found that the simplicity of Logo is especially well-suited for primary school and complete beginners. At high school and university level, Python then allows us to discuss topics in more depth and to better link our programming classes to mathematics and the sciences. We have also extended our Python interpreter and included Logo's `repeat`-loop into Python. This allows us to introduce iteration at an early stage without the need for variables, getting the best of both worlds.

Our curricula and examples make heavy use of turtle graphics, both in Logo as well as in Python. Apart from the obvious benefits of direct visualization, the turtle is also a source of powerful didactical metaphors. In particular, the examples as presented in this paper all rely on turtle graphics to convey or visualize an algorithmic principle.

1.2. *Objectives*

Computer science is a vast field with algorithmic thinking at its core. Our curricula hence focus on the study of algorithms and its various aspects. Our approach comprises three major aspects of algorithmic thinking, as described in the following paragraphs: the notion of the programming language as a formal language to express algorithms, abstraction and automation as central problem solving strategies, and the limits of practical computability as a motivation for improving existing algorithms. More on the authors' goals, motivation, and approaches can be found in a complementing paper (Hromkovič *et al.*, 2016).

Concept of a Formal Language. Students are introduced to programming as a means to convey instructions to a machine – in our case the turtle. The initial set of instructions is strongly limited and restricted to basic movements such as moving forward and turning. Each instruction has a clearly defined syntax and semantics, avoiding any ambiguity. At first, then, programming is the activity of writing sequences of such instructions, encoding graphical shapes. From our perspective, this is to say that students use a formal language to combine words to sentences. Even though each valid sentence conveys the information of a graphical shape, not every sentence makes sense in the context of the interpretation of the resulting shape.

The initial vocabulary given to the students is not adequate to encode more complex shapes in a human-accessible form. Students are early required to extend the vocabulary by defining new words, i. e., by defining subroutines. In the context of the turtle, this can

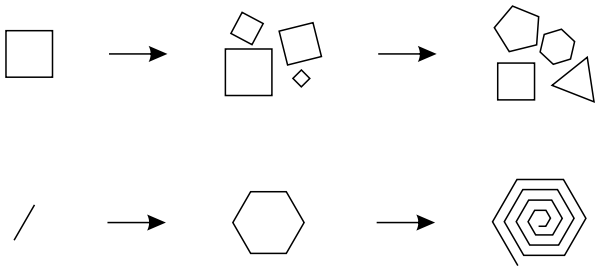

Fig. 1. By parametrizing programs, we gradually gain more versatile algorithms and procedures. Drawing a square of fixed size is the first step towards drawing arbitrary polygons of various sizes (above). Likewise, loops allow us to build ever more complex and larger programs out of simple and small parts (below).

be beautifully explained as "teaching the turtle new words" (Papert, 1993). The semantics of the new words is expressed algorithmically as a sentence over an already existing vocabulary. Think, for instance, of a house consisting of a triangle and a square. Both the triangle and the square themselves might be expressed as sequences of forward- and turning-instructions.

Hence, our objective is to provide students with a simple yet expandable base of instructions, the means to combine these instructions to sentences, and to define new words with associated unambiguous semantics. This way, the students are exposed to the concepts of modularization, formal languages, and expressing semantics in algorithmic form.

Abstraction and Automation. Programming is, of course, much more than combining instructions to form programs. Some of the most essential key concepts are abstraction and automation. Modularization, for instance, only unfolds its full potential in combination with parameters. Having a dedicated instruction to draw a square, say, helps to clarify the intent of a program. Allowing that very same instruction to draw squares of various sizes makes it versatile and open to applications beyond its initial conception. Further abstraction could even introduce a second parameter to pertain to the number of vertices to draw, resulting in one instruction capable of drawing all regular polygons (see Fig. 1).

Abstraction itself also requires the concept of automation. Even drawing a regular polygon with a given number of vertices is a tedious task without the notion of a loop. For the step to an abstract instruction encompassing all polygons, the loop becomes a necessity. Once this level of automation is mastered, students are introduced to loops with variations, allowing for figures such as spirals where even the "parameter" automatically varies (see Fig. 1).

Automation and abstraction are not just core concepts of programming but of computer science and algorithmic thinking in a much wider sense. Expressed in the context of problem solving, abstraction corresponds to the question "Can we adapt an already known or universally available strategy to solve the problem at hand?" Once we know how to solve a single instance, we then employ the concept of automation to apply our solution to a large set of instances.

Limits of Practical Computability. Finding solutions automatically is not always feasible. Indeed, the insight that there are problems that cannot be solved algorithmically (i. e., *undecidable problems*), shown in 1936 by Turing in his seminal paper "On Computable Numbers, with an Application to the Entscheidungsproblem" (Turing, 1936), is one of the deepest results of mathematics and laid the foundation for computer science itself. However, even computable problems can often only be solved under certain restrictions, e. g., using an unacceptable amount of resources (time and space), or without full precision due to numeric errors. This gives rise to numerous interesting research questions and solutions, which can both be explained to non-professionals.

For education, however, we need to make computability and its limitations visible and tangible. A prime example to serve this objective, as taken from turtle graphics, are circles. A computer cannot draw an exact circle, it must be drawn using an approximation such as a polygon (or Bézier curve). The cost of drawing an approximating polygon increases with the number of vertices, mainly due to the fact that the turtle needs time to turn at the vertices. Students therefore must find a compromise between more accurate representations and faster renderings, and eventually realize that the limitations of screen resolution quickly nullify additional precision beyond a certain point.

When seen in the light of modern applications, *intractability* is of particular importance to cryptography. In this regard, the inability to design efficient algorithms has far-reaching implications beyond computer science and its inclusion into the curriculum is well-warranted. At the same time, we found cryptography to be very motivating and well-suited as a subject of its own (Freiermuth *et al.*, 2010).

2. Modular Development: Building a Town Step by Step

Our group is actively involved in introducing young students to programming as soon as at fifth grade. To this end, we developed teaching material, hold classes, and, most importantly, introduce teachers to our didactic approach as well as to fundamental concepts of computer science. These school projects are based on the German textbook *An introduction to programming in Logo* (Hromkovič, 2014, German: *Einführung in die Programmierung mit Logo*), and on a Logo booklet (Gebauer *et al.*, 2016) covering the contents of its first seven chapters. The following example is taken from this booklet.

As already mentioned, one of the main objectives of our programming classes consists in making the students confident with the modular development of programs and teaching them how to systematically apply this problem solving strategy to complex problems. To illustrate our approach and the achievements of the students, we present a sequence of learning activities from the third (out of seven) unit of the courses.

At this point, the students already know how to move the turtle forward and backward on a straight line, how to rotate it as well as how to iterate over a sequence of

instructions for a predefined number of times. More precisely, the current vocabulary of the turtle comprises the following instructions as well as their abbreviations: `forward` (`fd`), `back` (`bk`), `left` (`lt`), `right` (`rt`), `clearscreen` (`cs`), `penup` (`pu`), `pendown` (`pd`), and `repeat`. Furthermore, the students are already used to giving their programs names and to reusing available programs as subprograms within main programs in an elementary way. In subsequent activities, they learn how to develop a table that consists of rows of identical squares in a proper modular way.

To reinforce the concept of modular development, the students are now challenged to write a program for drawing a small town, which consists of streets with identical houses. While the program for drawing a house is already available in the booklet, the students are expected to consequently apply the approach they intensively practiced. They are therefore expected to:

- Identify the next shape or pattern they can systematically reuse.
- Write a sequence of instructions for drawing it.
- Give this subprogram a name.
- Test and iteratively improve the code until the solution meets the assignment.

Afterwards, the students should reflect on how to adjust the position of the turtle in order to draw the pattern by simply reusing the program they developed above, to test and to iteratively improve their approach, and to finally integrate the two modules of their solution. In a following step, this new main program can be reused as a subprogram in other main programs of increased complexity. More specifically, the students are given the program shown in Listing 1.1 accompanied by the following exercise, which asks them to study the effects of each command in detail.

> ***Exercise.*** Where does the turtle start drawing the house? Think about the path the turtle follows when drawing the house using the program `HOUSE`. Where is the turtle located at the end of the execution? Draw the image and describe the effect of each command.

Next, they are told how to design a program `HOUSEROW` (Listing 1.2) that uses `HOUSE` as a subprogram. Here, the most difficult task is to position the turtle in such a way that, after each iteration, the new house is drawn at the correct coordinates.

Listing 1.1: Drawing a house using a simple `repeat`-loop.

```
to HOUSE
  rt 90
  repeat 4 [fd 50 rt 90]
  lt 60 fd 50 rt 120 fd 50 lt 150
end
```

Listing 1.2: Drawing a row of houses.

```
to HOUSEROW
  repeat 5 [HOUSE rt 90 pu fd 50 lt 90 pd]
end
```

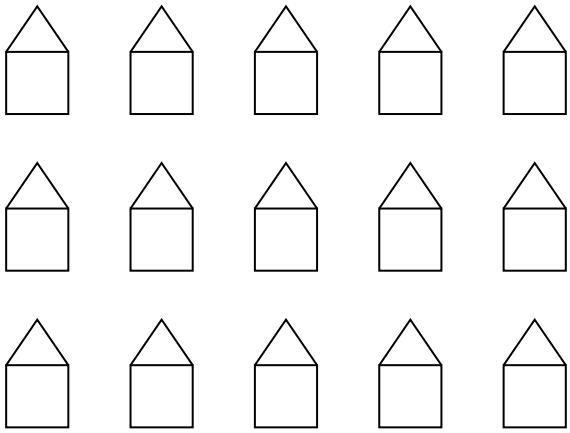

Fig. 2. A small town that consists of 15 houses.

Finally, the students are asked to use the modular approach in order to draw the town that consists of multiple streets. This way, the students learn how to extend the language of the computer step by step with more complex programs. The crucial observation is that the overall complexity is hidden in the smaller subprograms.

> ***Exercise.*** At this point, we would like to extend the complex of buildings by additional streets. Use the program `HOUSEROW` as a building block to draw the image shown in Fig. 2.
>
> *Hint:* After the completion of a row, the turtle has to be moved to the correct position to build the next street.

Modular development offers a didactically appealing platform for creative tasks. In the two following exercises, the students observe that even small changes such as adding a window, a door, or a chimney to their houses may have a considerable impact on the overall outcome of the streets and the town they are designing.

> ***Exercise.*** We decide to order the roof for the houses from another vendor. That is, we get two types of building blocks: One called `ROOF` and another one called `BASE`. Write two programs to draw the two building blocks. Combine those programs to form a new program `HOUSE1` that draws a house.

> ***Exercise.*** The houses in Listing 1.1 are very simple. Try to be creative and come up with a new design for a house. Use your house to build an entire complex of buildings.

The students learn that modular development is a systematic and efficient problem solving strategy. Moreover, they experience that subsequent changes in a basic module of a properly developed complex program require no or very limited additional programming effort.

3. Making Approximation Errors Visible with the Pac-Man

The turtle draws a circle by approximation, actually drawing a polygon with, say, 36 vertices (in practice, students often choose 360 vertices at first, building upon their knowledge that 360° stands for a complete circle). While the resulting figure is not discernible from a true circle on the screen, the approximation requires a couple of corrections when the circle is combined with other shapes. Most prominent is the question of finding the circle's center and the correct value of the radius. Both are slightly, but discernibly, off compared to a true mathematical circle.

A particularly illuminating example is drawing a Pac-Man shape. Students are asked to write a Python program that draws a Pac-Man and typically end up with a solution as shown in Listing 1.3 (note that the `repeat`-loop shown here has been added to Python in order to support our curriculum. A further discussion can be found in the aforementioned complementing paper (Hromkovič *et al.*, 2016)). However, their resulting pictures show that the shape is not closed as seen in Fig. 3: there is a small gap at the center of the shape. This discrepancy is subsequently discussed in a dedicated section and leads to a precise drawing of a pie chart.

Why does this gap in the center occur and how can we correct it? In a circle, any radius meets the circumference perpendicularly. This fact has been used twice in the program (Listing 1.3). For the approximation with a polygon, this does not hold anymore: the angle between the radius leading to a vertex and the circumference requires a small correction φ (Fig. 4). The correction φ is exactly half of the turtle's turning angle at each vertex. For our example with 36 vertices, this results in a 5°-correction (Listing 1.4). The correction of the angle can also be taken into the loop, resulting in a more symmetrical solution (Listing 1.5).

Listing 1.3: Drawing a Pac-Man.

```
from turtle import *
RADIUS = 100
right(45)
forward(RADIUS)
left(90)
repeat 27:
  forward(RADIUS * 3.1416 * 2 / 36)
  left(10)
left(90)
forward(RADIUS)
```

Listing 1.4: Correcting the angle (1).

```
left(90 + 5)
repeat 27:
  forward(RADIUS * 3.1416 * 2 / 36)
  left(10)
left(90 - 5)
```

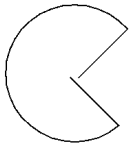

Fig. 3. An incomplete Pac-Man.

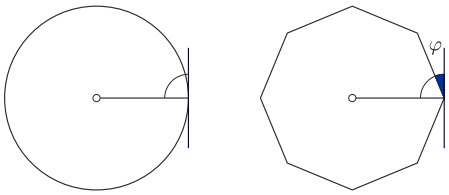

Fig. 4. As circles are approximated by polygons the radius does not meet the circumference in a right angle but is off by an angle φ.

Listing 1.5: Correcting the angle (2).

```
repeat 27:
  left(5)
  forward(RADIUS * 3.1416 * 2 / 36)
  left(5)
```

Listing 1.6: Starting the circumference not at a vertex but at the center of an edge instead.

```
repeat 27:
  forward(RADIUS * 3.15 / 36)
  left(10)
  forward(RADIUS * 3.15 / 36)
```

Finally, an alternative solution is to have the radius meet the circumference not at a vertex but at the center point of an edge. In this case, the radius does meet the circumference perpendicularly. The resulting code, again, has a very symmetrical form (Listing 1.6). Yet the ratio between radius and circumference now differs and requires to change the used approximation of the value of π.

4. Runtime Analysis Backed up by a Little Math

One of the authors is currently part of the team responsible at ETH Zurich for teaching computer science basics to non-computer science students (more specifically, students of biology, pharmacology, environmental sciences, health sciences and technology, agriculture, geology, and nutritional sciences).

We have been introducing students to Logo using the previously mentioned booklet (Gebauer *et al.*, 2016, see Section 2), whose main part is covered in roughly the first unit

Listing 1.7: A simple square.

```
to QUAD :WIDTH
  repeat 4 [fd :WIDTH rt 90]
end
```

Listing 1.8: Testing whether a given number is prime.

```
to PRIME :NUM
  ht
  make "IT 2
  make "ISPRIME 1
  while [:IT<:NUM] [
    make "RES mod :NUM :IT
    if :RES=0 [make "ISPRIME 0] []
    make "IT :IT+1
  ]
  if :ISPRIME=1 [setpc 1 QUAD 8] [setpc 0 QUAD 8]
end
```

of the lecture. After that, an advanced booklet is supplied that is specifically designed for this class (Böckenhauer *et al.*, 2015a). As part of this booklet, more involved concepts such as variables, conditional execution, and `while`-loops are introduced. The students are then given three *projects*, which consolidate what they have learned so far by designing small programs to solve specific tasks (Böckenhauer *et al.*, 2015b). The lecture is accompanied by exercise classes in which the students are asked to present and explain their solutions to a tutor.

As a first step towards the mathematical analysis of algorithms, we give the students the following project. The examples are taken from the project booklet (Böckenhauer *et al.*, 2015b) and the corresponding lecture notes. The goal is to show the students the idea of how to mathematically analyze how long a program will run depending on the input size. A typical example that does not need anything beyond high school mathematics is to test whether a given number is prime. Logo is especially suited to visualize the distribution of (small) prime numbers without much overhead.

The first component is a program called `QUAD` (Listing 1.7) that draws a square, and which essentially consists of a simple loop, which the students already know from previous lessons. The size of the square is determined by the value of the parameter `:WIDTH`.

Next, we can write a program that tests whether a given input is a prime number. This is done in the most straightforward fashion, i. e., by testing whether there is smaller number (except 1) that divides it. Depending on the result, either a red or black square is drawn on the screen using the instruction `setpencolor` (`setpc`). Before that, the turtle is hidden with the instruction `hideturtle` (`ht`). The corresponding algorithm `PRIME` is shown in Listing 1.8.

The students can easily follow the steps and try out different inputs. As a next step, we ask them to carefully check corner cases, and give them the following exercise.

Exercise. PRIME does not yet work correctly on all inputs as the value of :IT is initially set to 2. However, we know that 1 is by definition not a prime number. Thus, PRIME 1 creates an incorrect output. Extend the program such that a black square is drawn when the input is 1. Moreover, an error message should be output if 0 or a negative number is given.

Once the students familiarized themselves with the algorithm, we discuss its *running time*. It is obvious that the time grows with larger inputs, and this seems to be unavoidable on an intuitive level. Furthermore, it is easy to see that the running time directly depends on how often the body of the while-loop has been executed. We therefore agree on counting the number of these executions and neglect how many instructions are executed with each iteration. The above trivial attempt needs roughly 2^n iterations for inputs of length n (hence, n is the number of bits used to represent the input number). Now we show how to improve this running time using a little bit of math. The following exercise is well-suited to be presented to the students as part of the lecture.

Exercise. We can now make our algorithm PRIME "faster" (more efficient) by having it execute the while-loop less often. To this end, we make use of the following idea.

Suppose the input x is not a prime number. Then, by the definition of a prime number, there is a number a, which is neither 1 nor x, that divides x without remainder. But from this it also follows that there is a second number b, which is also neither 1 nor x, that also divides x without any remainder. An important point is that one of these two numbers is not larger than the square root $\sqrt{x}$ of x. If, e. g., a is larger than $\sqrt{x}$, then b has to be smaller, since otherwise $a \cdot b$ were larger than x.

We want to use this observation to improve PRIME. Write an algorithm PRIME2 which works exactly as PRIME, but in which the while-loop is modified such that the variable :IT does not take the values of all numbers smaller than :NUM, but only those that are smaller than or equal to $\sqrt{:\text{NUM}}$.

The students are asked to verify the speedup by trying different inputs. Good candidate inputs to observe the increase in speed of course depend on the computer used. Moreover, the students can try to make a simple running time analysis of PRIME2 themselves, which leads to the result that the loop is now executed at most (roughly) 1.41^n times for inputs of length n.

The above considerations give rise to another question, namely in which kind of analysis we are interested. To this end, the algorithm is modified such that the while-loop is left as soon as a divisor of the input is found. The resulting algorithm obviously still works correctly, but is it faster? Indeed, if the input is, say, an even number, the running time of the new algorithm is a lot better. However, if the input is prime, both algorithms take the same time. This is exactly the difference between a best case and a worst case analysis of the algorithm's running time.

Now we can follow the modular building of algorithms (see Section 2) and use PRIME2 as a building block to visualize the distribution of small prime numbers. More precisely, the algorithm PRIMES shown in Listing 1.9 uses PRIME2 as a subprogram to

Fig. 5. The distribution of prime numbers between 1 and 26. Instead of red and black squares, we use filled and unfilled ones.

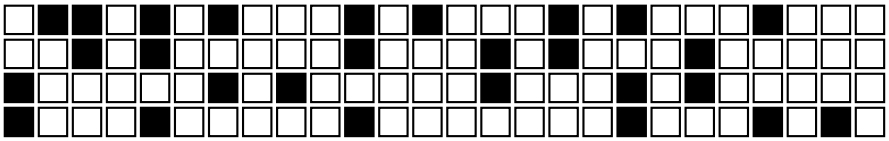

Fig. 6. The distribution of prime numbers between 1 and 104.

Listing 1.9: Visualizing primes.

```
to PRIMES :MAX
  pu lt 90 fd 300 rt 90 pd
  make "TEST 1
  repeat :MAX [
    pu rt 90 fd 10 lt 90 pd
    PRIME2 :TEST
    make "TEST :TEST+1
  ]
end
```

Listing 1.10: Visualizing primes more nicely.

```
to PRIMES2 :MAX :ROW
  pu lt 90 fd :ROW*10/2 rt 90 pd
  make "TEST 1
  repeat :MAX [
    pu rt 90 fd 10 lt 90 pd
    PRIME2 :TEST
    make "REST mod :TEST :ROW
    if :REST = 0 [
      pu lt 90 fd :ROW*10 lt 90 fd 10 rt 180 pd
    ] [ ]
    make "TEST :TEST+1
  ]
end
```

visualize the appearances of prime numbers among the first `:MAX` natural numbers.

The result of executing `PRIMES 26` is shown in Fig. 5. Next, we can improve the appearance by having the squares drawn in multiple rows (Listing 1.10). To this end, we write a new algorithm `PRIMES2` with an additional parameter `:ROW`. The turtle moves to the next row whenever it drew a number of squares that is divisible by the value of `:ROW`. With `PRIMES2 104 26` we obtain an output as shown in Fig. 6.

An advanced exercise then deals with prime powers. The students should solve this task at home, either alone or in small groups. The difficulty of this exercise is due to the usage of a return statement, which is implemented by the `output` instruction in Logo.

Exercise. A prime power is a natural number that has exactly one prime factor. For instance, 27 is a prime power since it has the prime factorization

$$27 = 3 \cdot 3 \cdot 3 = 3^3.$$

Clearly, every prime number is thus also a prime power.
In this project, you design a program PRIMEPOW, which checks whether a given number is a prime power. To this end, do the following steps:

1. Rewrite the program PRIME to obtain a program PRIMEOUT that uses the command output instead of drawing squares. If the value of :NUM is prime, the value 1 should be returned, otherwise 0.
2. PRIMEPOW has one parameter :TEST. We want to check whether the value assigned to :TEST is a prime power (possibly with the exponent being 1).
3. First, the program checks using PRIMEOUT whether :TEST is a prime number. If so, "Prime." is printed on the screen and the execution is ended using stopall.

```
make "ISPRIME PRIMEOUT :TEST
if :ISPRIME=1 [pr [Prime.] stopall] []
```

4. Otherwise, all numbers smaller than the value of :TEST are iterated, and again using PRIMEOUT it is checked whether the current number is a prime. If so, it is checked whether it divides the value of :TEST without remainder.
5. If such a prime number is found, PRIMEPOW takes note of this by setting the value of a variable :FOUND to 1. :FOUND is initialized with 0. If a second such prime number is found, this will be noted since the value of :FOUND is already 1. In this case, "More than one divisor." is printed on the screen and the execution is again ended with stopall.
6. Finally, if :FOUND is still 1 after all numbers were tested, "Prime Power. Base: " and the prime number that divides the value of :TEST without remainder are printed on the screen.
7. Check PRIMEPOW using small input values.

This introduction using Logo proved to be very valuable for the students in the succeeding lessons, where we implement more complex projects using Python. More precisely, they were able to learn important paradigms without having to worry too much about syntactical details.

5. Conclusion

Programming education is a great opportunity to teach important core concepts of computer science on various levels and to establish algorithmic thinking as part of a broad and general education. A necessary prerequisite is, of course, that we find ways to go beyond teaching the specifics of a programming language and rather put emphasis on those aspects of programming that lead to a deeper understanding of computer science.

In this article, we have provided three examples of how programming education can incorporate more general principles of algorithmic thinking. All three examples have been taken from our well-tested teaching materials for primary school, high school, and university level, respectively. Further details about our curricula are given in the complementing paper (Hromkovič *et al.*, 2016).

References

Bell, T., Rosamond, F., Casey, N. (2012). Computer science unplugged and related projects in math and computer science popularization. *The Multivariate Algorithmic Revolution and Beyond*, Springer-Verlag, 398–456.

Böckenhauer, H.-J., Hromkovič, J., Komm, D. (2015). *Programmieren mit LOGO – Projekte*. `http://abz.inf.ethz.ch/wp-content/uploads/unterrichtsmaterialien/primarschulen/logo_projekte.pdf`

Böckenhauer, H.-J., Hromkovič, J., Komm, D. (2015). *Programmieren mit LOGO für Fortgeschrittene*. `http://abz.inf.ethz.ch/wp-content/uploads/unterrichtsmaterialien/primarschulen/logo_heft_2_de.pdf`

Freiermuth, K., Hromkovič, J., Keller, L., Steffen, B. (2010). *Einführung in die Kryptologie*. Springer.

Gallenbacher, J. (2008). *Abenteuer Informatik. IT zum Anfassen – von Routenplaner bis Online-Banking*. Springer, 2 edition.

Gebauer, H., Hromkovič, J., Keller, L., Kosírová, I., Serafini, G., Steffen, B. (2016). *Programmieren mit LOGO*. `http://abz.inf.ethz.ch/wp-content/uploads/unterrichtsmaterialien/primarschulen/logo_heft_de.pdf`

Hromkovič, J. (2014). *Einführung in die Programmierung mit LOGO – Lehrbuch für Unterricht und Selbststudium*. Springer, 3 edition.

Hromkovič, J., Kohn, T., Komm, D., Serafini, G. (2016*). Combining the Power of Python with the Simplicity of Logo for a Sustainable Computer Science Education*. Unpublished Manuscript.

Kohn, T. (2016). *Python. Eine Einführung in die Computer-Programmierung*. `http://jython.tobiaskohn.ch/PythonScript.pdf`

Papert, S. (1993). *Mindstorms*. Basic Books, 2 edition.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2), 230–265.

J. Hromkovič is professor of informatics with a special added focus on computer science education at ETH Zurich. He is author of about 15 books published in 6 languages (English, German, Russian, Spanish, Japanese, and Slovak) and about 200 research articles. He is member of Academia Europaea and the Slovak Academic Society.

T. Kohn is writing his PhD thesis in computer science at ETH Zurich. The focus of his research is programming education, particularly in high schools. He holds an MSc in mathematics from ETH and has been teaching mathematics and computer science for 10 years.

D. Komm is lecturer at ETH Zurich and an external lecturer at University of Zurich. He studied computer science at RWTH Aachen University and Queensland University of Technology. He received his PhD from ETH Zurich in 2012. His research interests focus on algorithmics and advice complexity.

G. Serafini is lecturer in the Computer Science Teaching Diploma Program at ETH Zurich. He holds a MSc in computer science and a teaching diploma in computer science from ETH Zurich. His research interests focus on the contribution of computational thinking to school education. He is a member of the board of the Swiss Computer Science Teacher Association and a member of the Swiss Olympiad in Informatics.