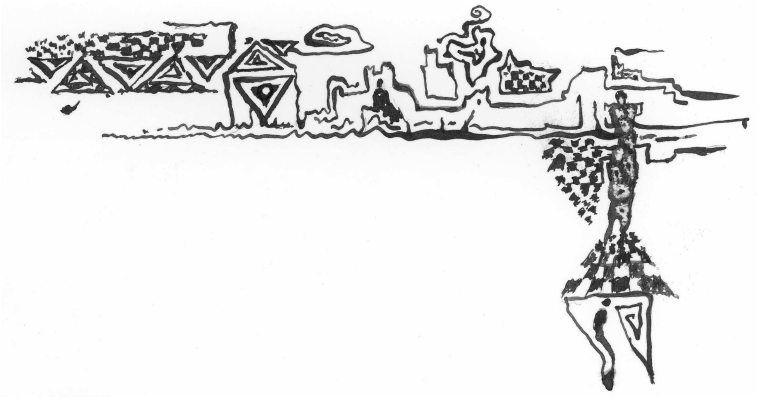

Perfection is based upon small things,
but perfection itself is no small thing at all.

Michelangelo Buonarotti

Chapter 2

Algorithmics, or What Have Programming and Baking in Common?

2.1 What Do We Find out Here?

The aim of this chapter is not to present any magic results or real miracles. One cannot read Shakespeare or Dostojevski in their original languages without taking the strenuous path of learning English and Russian. Similarly, one cannot understand computer science and marvel about its ideas and results if one did not master the fundamentals of its technical language.

As we already realized in the first chapter on computer science history, the algorithm is the central notion of computer science.

We do not want to take the whole strenuous path of learning all computer science terminology. We want to show that without using formal mathematical means, one can impart an intuitive meaning of the notion of an algorithm which is precise enough to imagine what algorithms are and what they are not. We start with cooking and then we discuss to what extent a recipe can be viewed as an algorithm.

After that, we directly switch to computers and view programming as a communication language between man and machine and imagine that programs are for computers understandable representations of algorithms. At the end of the chapter, you will be able to write simple programs in a machine language on your own and will understand to a fair extent what happens in a computer during the execution of computer instructions (commands).

By the way, we also learn what an algorithmic problem (task) is and that one is required to design algorithms in such a way that an algorithm works correctly for each of the infinitely many problems instances. To work correctly means to compute the correct result in a finite time. In this way, we build a bridge to Chapter 3, in which we show how important a deep understanding of the notion of infinity is for computer science.

2.2 Algorithmic Cooking

In the first chapter, we got a rough understanding of the meaning of the notion of algorithm or method. Following it, one can say:

> *An algorithm is an easily understood description of an activity leading to our goal.*

Hence, an algorithm (a method) provides simple and unambiguous advice on how to proceed step by step in order to reach our goal. This is very similar to a cooking recipe. A recipe tells us exactly what has to be done and in which order, and, correspondingly, we perform our activity step by step.

To what extent may one view a recipe as an algorithm?

To give a simple answer to this question is not easy. But, searching for an answer, we approach a better understanding of the meaning of this crucial term.

Let us consider a recipe for an apricot flan of diameter 26 cm.

Ingredients:

3	egg whites
1	pinch of salt
6	tablespoons of hot water
100g	cane sugar
3	egg yolks
1	teaspoon of lemon peel
150g	flour
1/2	teaspoon of baking powder
400g	peeled apricots

Recipe:

1. Put greaseproof paper into springform!
2. Heat the oven up to 180°C!
3. Heat up 6 tablespoons of water!
4. Mix three egg whites with the hot water and a pinch of salt, beat them until you get whipped egg white!
5. Beat 100g cane sugar and 3 egg yolks until a solid cream develops!
6. Add 1 teaspoon of lemon peel to the cream and mix them together!
7. Mix 150g flour and 1/2 teaspoon of baking powder and add it to the mixture! Then stir all contents carefully using a whisk!
8. Fill the baking tin with the cream mixture!
9. Place the skinned apricots on the mixture in a decorative way!

10. `Put the baking tin into the oven for 25-30 minutes until it gets a light brown color!`

11. `Take the flan out of the oven and let it cool!`

The recipe is available and the only question is whether we are able to bake the flan by following it. A possible answer may be that success can depend to some extent on the experience and the knowledge of the cook.

We are ready to formulate our first requirements for algorithms.

> *An algorithm has to be such an exact description of the forthcoming activity that one can successfully perform it even in the case where one does not have any idea why the execution of the algorithm leads to the given aim. Moreover, the description (algorithm) has to be absolutely unambiguous in the sense that different interpretations of the particular instructions are excluded. It does not matter who executes the algorithm, the resulting activity and so the resulting outcome must be the same, i.e., each application of the algorithm has to reach the same result.*

Now, one can start a long discussion about which of the 11 steps (instructions) of the recipe above can be viewed as unambiguous and easily understood by everybody. For instance:

- What does it mean **"to beat until one gets whipped egg whites"** (step 4)?
- What does it mean **"to stir carefully"** (step 7)?
- What does **"decorative"** mean (step 9)?
- What does **"light brown"** mean (step 10)?

An experienced cook would say: "Alright. Everything is clear, the description is going into unnecessary detail." Somebody trying to bake her/his first cake could require even more for help and may even fail to execute the whole procedure on her/his own. And this can happen in spite of the fact that our recipe is a more detailed and simpler description than the usual recipes described in cookery

books. What do you think about cookery book instructions such as:

- Put **quickly** a **little bit cooked** gelatin below the cheese and **stir them thoroughly**?

We are not allowed to accept situations in which an experienced person considers the recipe to be an algorithm and the rest of the world does not. One has to search for a way in which we can get general agreement. We already know that an algorithm is a sequence of instructions that are correctly executable by any person. This means that before defining the notion of a cooking algorithm

> *we have to agree on a list of instructions (elementary operations) such that each of these instructions can be mastered by anybody willing to cook or bake.*

For instance, such a list can contain the following instructions that are possibly correctly executable by a robot that does not have any understanding of cooking and no improvisation ability.

- `Put x spoons of water into a container!`
- `Separate an egg into an egg yolk and the egg white!`
- `Heat the oven up to x°C!`
- `Bake for y minutes at x°C!`
- `Weigh x g of substance A and put it into a container!`
- `Pour x l of liquid B into a pot!`
- `Stir the content of the container using a whisk for t minutes!`
- `Mix the content of the container using a fork for t minutes!`
- `Mix the content of the two containers!`
- `Pour the mixture into the baking tin!`

Certainly, you can find many further instructions that one can consider to be simple enough in the sense that we can expect that anybody is able to execute them. In what follows we try to rewrite the recipe in such a way that only simple instructions are applied.

Let us try to rewrite step 4 of our recipe into a sequence of simple instructions.

`4.1 Put the three egg yolks into the container G.`

`4.2 Put 1g of salt into G.`

`4.3 Put 6 tablespoons of water into the pot T.`

`4.4 Heat the water in T up to 60°C.`

`4.5 Pour the contents of T into G.`

Now, we get trouble. We do not know how to execute the instruction "`mix until the content of G becomes whipped egg white`". A solution may be to use some experimental values. Maybe it takes 2 minutes until the mass is stiff enough. Hence, one could write:

`4.6 Mix the content of G for 2 minutes.`

An instruction of this kind may also be risky. The time of mixing depends on the speed of mixing, and that may vary from person to person. Hence, we would prefer to stop mixing approximately at the moment when the mass became stiff. What do we need for that? We need the ability to execute tests in order to recognize the moment at which the whipped egg white is ready. Depending on the result of the test, we have to make a decision on how to continue. If the mass is not stiff, we have to continue to mix for a time. If the mass is stiff then the execution of step 4 is over and we have to start to execute step 5.

How can one write this as a sequence of instructions?

```
4.6 Mix the content of G for 10 seconds.
4.7 Test whether the content of G is stiff or not.
    If the answer is "YES", then continue with step 5.
```

```
If the answer is "NO", then continue with step 4.6.
```

In this way, one returns to step 4.6 until the required state of the mass is reached. In computer science terminology, one calls steps 4.6 and 4.7 a cycle that is executed until the condition formulated in 4.7 is satisfied. To make it transparent one uses a graphic representation such as in Fig. 2.1; this is called a **flowchart**.

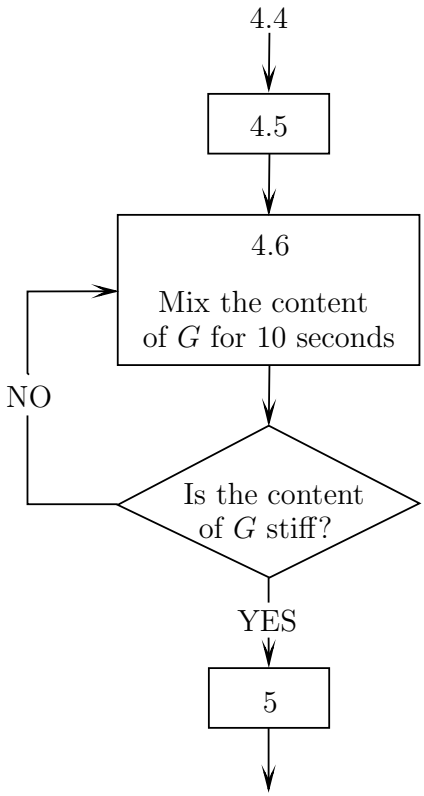

Fig. 2.1

Can one easily execute the test in step 4.6? Exactly as in the case of instructions, we have to agree on a list of simply executable tests. We do not want to discuss a possible execution of the test in step 4.6 in detail because the author is not an expert in cook-

ing. Originally, I considered putting a teaspoon into the mass and looking whether it stays up. But a female student explained to me that this does not work and that instead it is sufficient to make a cut in the mass using a knife and when the cut does not close (i.e., remains open), then the mass is stiff. Examples of other tests are:

- `Test whether the temperature of the liquid in a pot is at least x degrees.`
- `Test whether the weight of the content of a container is exactly x g.`

Exercise 2.1 Create a list of instructions and tests you consider anybody could execute. Then take your favorite recipe and rewrite it using the instructions and the tests from your list only.

Exercise 2.2 You want to heat 1 l of water up to 90°C. You are allowed to use only the following instructions:

- `Put the pot T on the hot plate for x seconds and then take it away.`
- `Pour x l of water into pot T.`

Moreover, you are allowed to use the following tests.

- `Test whether the water in pot T has at least x°C.`

Use this test and the two instructions above to write a cooking algorithm for heating 1 l of water up to 90°C that assures that the pot is not longer than 15 s on the hot plate after the water reached 90°C.

Whether you believe or not, after successfully solving these two exercises, you have already been working as a programer. The most important fact we learnt by baking is that we cannot speak about algorithms before the fundamental elements algorithms consist of are fixed. These elements are simple instructions and tests that everyone can execute without any problem.

2.3 What About Computer Algorithms?

Here, we want to discuss the similarities and the differences between algorithmic cooking and algorithmic computing in order to

realize exactly what computer algorithms and computer programs are and what they are not.

Analogous to cooking, one has to fix first a list of fundamental instructions (operations) that a computer can execute without any doubt. To get agreement here is essentially simpler than getting it by cooking. A computer does not have any intelligence and so any improvisation ability. Due to this, the language of the computer is very simple. Nobody doubts that a computer can add or multiply two integers or execute other arithmetic operations with numbers. Similarly, everyone accepts that a computer can compare two numbers as a test. These simple instructions and tests together with the ability to read the input data and to output the results are sufficient for describing any algorithm as a sequence of instructions.

It does not matter whether we consider cooking algorithms or computer algorithms. Both are nothing other than a sequence of simple instructions. But there is also an essential difference between cooking algorithms and algorithms in mathematics and in computer science. The input of a cooking algorithm is a set of ingredients and the result is a meal. The only task is to cook the aimed product from the given ingredients. Algorithmic tasks are essentially different. We know that a problem may have *infinitely many* **problem instances** as possible inputs. Consider, for instance, the problem of solving a quadratic equation.

$$ax^2 + bx + c = 0.$$

The input data are the numbers a, b, and c and the task is to find all x that satisfy this equation.

For instance, a concrete problem instance is to solve the following equation:

$$x^2 - 5x + 6 = 0.$$

Here, we have $a = 1, b = -5$, and $c = 6$. The solutions are $x_1 = 2$ and $x_2 = 3$. By substituting these values, one can easily verify that

$$2^2 - 5 \cdot 2 + 6 = 4 - 10 + 6 = 0$$

$$3^2 - 5 \cdot 3 + 6 = 9 - 15 + 6 = 0$$

and so that x_1 and x_2 are really the solutions of the quadratic equation $x^2 - 5x + 6 = 0$.

Because there are infinitely many numbers, one has infinitely many possibilities to choose the coefficients a, b, and c of the quadratic equation. Our clear requirements on an algorithm for solving quadratic equations is that the algorithm determines the correct solution for all possible input data a, b, and c, i.e., for each quadratic equation.

In this way, we get the second basic demand on the definition of the notion of an **algorithm**.

> *An algorithm for solving a problem (a task) has to ensure that it works correctly for each possible problem instance. To work correctly means that, for any input, it finishes its work in a finite time and produces the correct result.*

Let us consider an algorithm for solving quadratic equations. Mathematicians provided the following formulas for computing the solutions

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a},$$

if $b^2 - 4ac \geq 0$. If $b^2 - 4ac < 0$, there does not exist any real solution[1] to the equation. These formulas directly provide the following general method for solving quadratic equations.

[1]The reason for that is that one cannot take the root of a negative number.

Input: Numbers a, b, and c representing the quadratic equation $ax^2 + bx + c = 0$.

Step 1: Compute the value $b^2 - 4ac$.

Step 2: If $b^2 - 4ac \geq 0$, then compute

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Step 3: If $b^2 - 4ac < 0$, write "there is no real solution".

Now, we believe the mathematicians that this method really works and we do not need to know why in order to rewrite it as an algorithm.

However, we want to do more than to transform the description of this method into a program. The notion of a **program** is considered here as a *sequence of computer instructions* that is represented in a form that is understandable for a computer. There are essential differences between the notion of a program and the notion of an algorithm.

1. A program does not need to be a representation of an algorithm. A program may be a meaningless sequence of computer instructions.

2. An algorithm does not need to be necessarily written in the form of a program. An algorithm can also be described in a natural language or in the language of mathematics. For instance, the use of instructions such as "`multiply a and c`" or "`compute` $\sqrt{c}$" is acceptable for description of an algorithm while a program must be expressed in a special formalism of the given programming language.

We view **programming** as an *activity of rewriting algorithms* (methods descriptions) into programs. In what follows, we will

program a little bit in order to see how one can create a complex behavior by writing a sequence of very simple instructions.

In order to be able to read and understand the forthcoming chapters, it is not necessary to learn the rest of this chapter in detail. Hence, everybody not strongly interested in learning what programming is about and what happens in a computer during the execution of concrete instructions can jump this part.

We start by listing the simple operations and their representation in our programming language that we call "TRANSPARENT". In passing we show the high-level structure of a computer and see the main computer actions performed during the execution of some particular instructions.

We consider a rough, idealized model of a computer as depicted in Fig. 2.2.

This computer model consists of the following parts:

- A **memory** that consists of a large number of memory cells. These memory cells are called **registers**. The registers are numbered by positive integers and we call them **addresses** of the registers. For instance 112 is the address of `Register(112)`. This corresponds to the image in which the registers are houses on one side of a long street. Each register can save an arbitrarily large number[2].
- A special memory in which the whole program is saved. Each row of the program consists of exactly one instruction of the program. The rows are numbered starting at 1.
- There is a special register `Register(0)` that contains the number of the just executed instruction (row) of the program.

[2]In real computers, the registers consist of a fixed number of bits, 16 or 32. The large integers or real numbers with many positions after the decimal point that cannot be represented by 32 bits have to be handled in a special way by using several registers for saving one number. Hence, we have idealized the computer here in order to remain transparent and we assume that any register can save an arbitrarily large number.

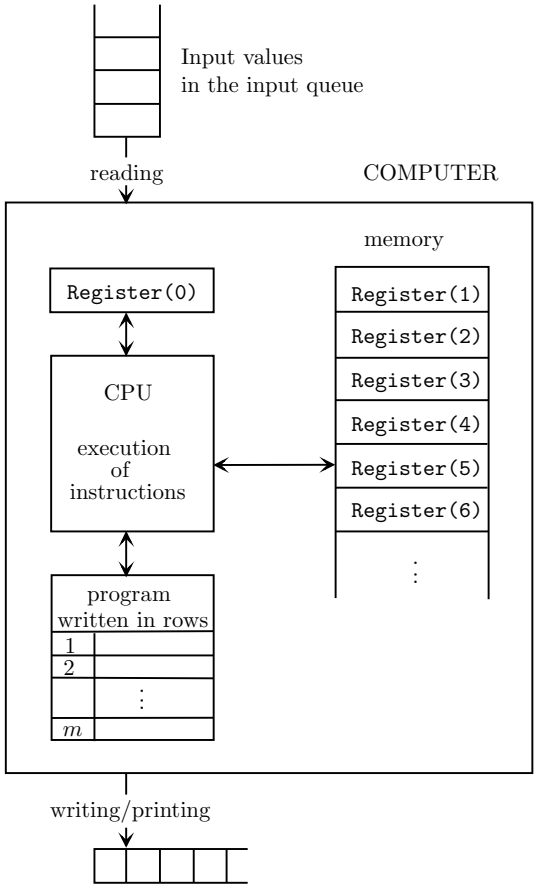

Fig. 2.2

- A CPU (central processor unit) that is connected to all other parts of the computer. In order to execute one instruction, the CPU starts by reading the content of `Register(0)` in order to fix which instruction has to be executed. Then looking at the corresponding instruction of the program, the CPU fetches the contents of the registers (the numbers saved in the registers) that are arguments of the executed instructions and executes the corresponding operation on these data. Finally, the CPU saves the result in the register determined by the instruction

and adjusts the contents of the register `Register(0)` to the number of the instruction to be executed next.

Additionally, the computer is connected to the world outside. The input data are waiting in a queue and the computer can read the first number in the queue and save it in a register. The computer has also a tape, where it can write the results computed.

Consider an analogy to baking or cooking. The computer is the kitchen. The registers of the memory are containers, bowls, jars, etc. Each container has an unambiguous name (exactly as each register has an address), and because of this one always knows which container is just considered. The memory containing the program is a sheet or a cookery book. The CPU is a person or a cookery robot together with all the other machines such as ovens, mixers, microwaves, etc. that are available in the kitchen. The content of `Register(0)` is the note telling where we are in this process of executing the recipe. The inputs are waiting in the refrigerator or in the pantry. We have to notice that they are not waiting in a queue, but one can take all the ingredients out and build a queue that respects the order in which they are needed. Certainly we do not write the output, instead we put it on the table.

As we have already learnt by baking, the first step and the crucial point for defining the notion of an algorithm is to agree on a list of **executable** instructions (operations). Everybody has to be convinced about their executability.

In what follows, we prefer to present the possible computer instructions in natural language instead of using the formal language of the computer called machine code. We start with the instructions for reading.

(1) `Read into Register(`n`).`

To execute this operation means to take the first number of the queue and save it in `Register(`n`)`. In this way this number is deleted from the queue and the second number of the queue takes over the first position of the queue.

Example 2.1 Consider the situation in which the three numbers 114, −67, and 1 are waiting to be picked up. All registers of the memory contain the value 0, except for `Register(0)` which contains 3. One has to execute the instruction

`Read into Register(3)`

in the third row of the program. After the execution of this instruction, `Register(3)` contains the number 114. Then, numbers −67 and 1 are still waiting in the queue. The content of `Register(0)` is increased by 1 and becomes 4, because after executing the instruction of the third row of the program one has to continue by executing the instruction of the next row.

The execution of this instruction is depicted in Fig. 2.3. We omit describing the whole computer state before and after executing this instruction and focus on the content of the registers only. □

The next instruction enables us to put a concrete number into a register without being forced to read it from the input queue

(2) `Register(`n`)` $\leftarrow k$

This instruction corresponds to the requirement to put the number k into the register `Register(`n`)`. Executing it means deleting the old content of `Register(`n`)`. After the execution of this instruction, this old content is not available anymore, it is definitely destroyed. There is no change in the queue related to this instruction.

Example 2.2 Consider that `Register(50)` contains the number 100. After executing the instruction

`Register(50)` $\leftarrow 22$

the register `Register(50)` contains the number 22. The old content 100 of `Register(50)` is not saved anywhere and so it is definitely lost.

If the next instruction is

`Read into Register(50)`

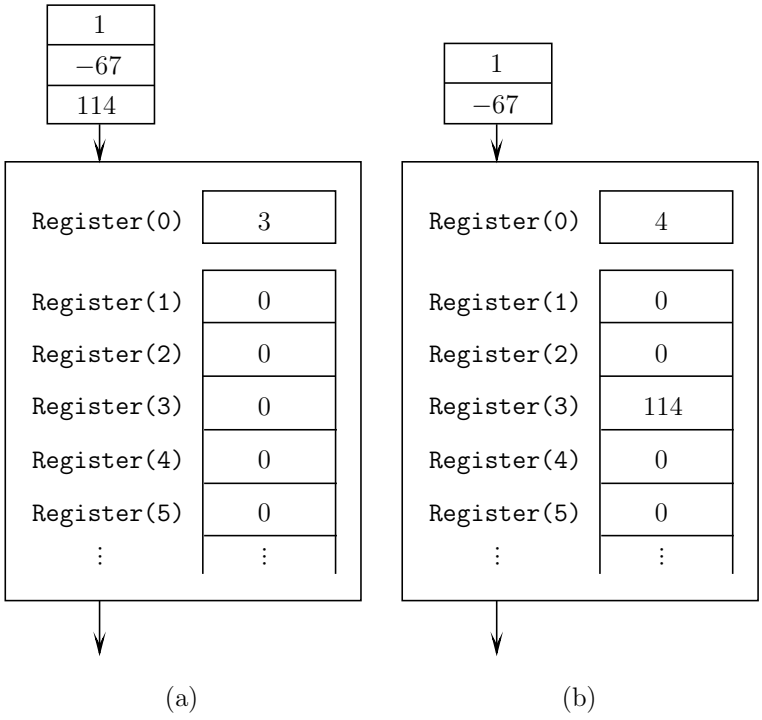

Fig. 2.3

and the number 7 is waiting in the queue, then after the execution of this instruction the number 7 in `Register(50)` is exchanged for the number 22. □

Exercise 2.3 The numbers 11, 12, and 13 are waiting in the input queue. The content of `Register(0)` is 1. `Register(2)` contains 1117 and `Register(3)` contains 21. All other registers contain 0.

a) Depict this situation analogously as in Fig. 2.3.
b) Execute the following program

```
1 Read into Register(1)
2 Register(2) ← 100
3 Read into Register(3)
4 Read into Register(2)
```

Determine and depict the content of all registers and the input queue after the execution of each particular instruction of the program.

Now we introduce some of the possible arithmetic instructions.

(3) `Register(`n`)` $\leftarrow$ `Register(`j`) + Register(`i`)`

The meaning of this instruction is as follows. One has to add the content of `Register(`j`)` to the content of `Register(`i`)` and to save the result in `Register(`n`)`. Executing this instruction, the original content of `Register(`n`)` is overwritten by the result of the addition. The contents of all other registers remain unchanged, except for the content of `Register(0)` that is increased by 1 (i.e., the execution of the program has to continue with the next instruction). There is also no change in the input queue.

Example 2.3 Consider the situation (the state of the computer), in which `Register(0)` contains 5 and each `Register(`i`)` contains the number i for $i = 1, 2, 3, 4$, and 5 (Fig. 2.4a). All other registers contain 0. The 5th row of the program contains the following instruction:

`Register(7)` $\leftarrow$ `Register(1) + Register(4).`

Figure 2.4b shows the situation reached after the execution of this instruction of addition.

The value 1 from `Register(1)` and the value 4 from `Register(4)` are summed $(1 + 4 = 5)$ and the result 5 is saved in `Register(7)`. The contents of `Register(1)` and `Register(4)` do not change during the execution of this instruction.

Assume that row 6 of the program contains the following instruction:

`Register(7)` $\leftarrow$ `Register(1) + Register(7).`

The content of `Register(1)` is 1 and the content of `Register(7)` is 5. Accordingly, the computer computes $1 + 5 = 6$ and saves 6 in `Register(7)`. In this way, the original content of `Register(7)` is deleted. Executing this instruction, we observe that one is also allowed to save the result of a computer operation in one of the

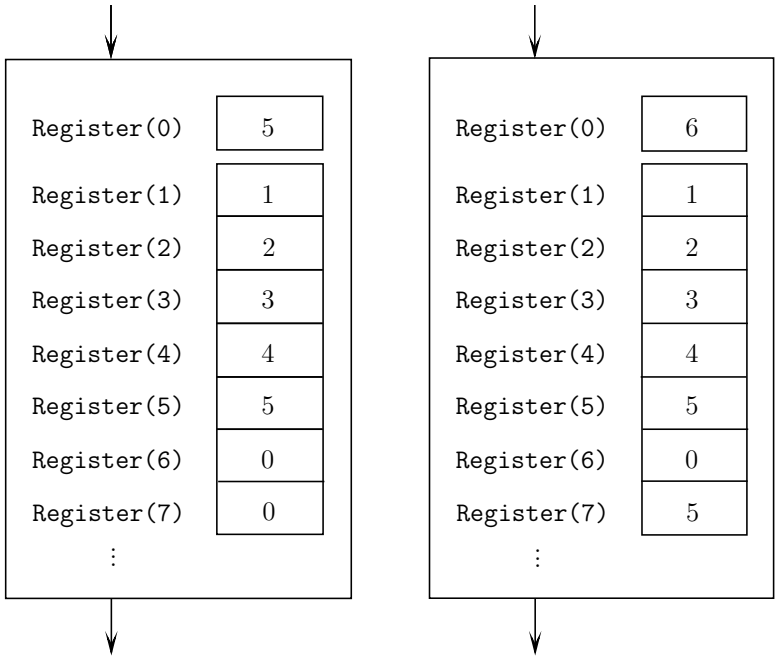

Fig. 2.4

two registers containing the operands (the incoming values for the operation). □

Exercise 2.4 Consider the computer state after executing the first addition (the instruction in row 5) in Example 2.3. This situation is depicted in Fig. 2.4b. Depict the memory state (analogously to Fig. 2.4b) after executing the second addition operation from row 6! After that perform the following three instructions

```
7 Register(3) ← 101
8 Register(3) ← Register(3) + Register(3)
9 Register(3) ← Register(7) + Register(3)
```

of the program and depict the final state of the memory.

Analogously to addition one can perform other arithmetic operations, too.

(4) `Register(`n`)` $\leftarrow$ `Register(`j`) - Register(`i`)`

To execute this operation means to subtract the content of `Register(`i`)` from the content of `Register(`j`)` and to save the result in `Register(`n`)`.

(5) `Register(`n`)` $\leftarrow$ `Register(`j`)` $*$ `Register(`i`)`

The computer has to multiply the contents of the registers `Register(`j`)` and `Register(`i`)` and to save the result in `Register(`n`)`.

(6) `Register(`n`)` $\leftarrow$ `Register(`j`) / Register(`i`)`

The computer has to divide the content of `Register(`j`)` by the content of `Register(`i`)` and to save the result in `Register(`n`)`.

(7) `Register(`n`)` $\leftarrow \sqrt{\texttt{Register}(m)}$

The computer has to compute[3] the root of the content of `Register(`n`)` and to save the result in `Register(`m`)`.

Exercise 2.5 Consider the following situation. All registers except[4] for `Register(0)` contain the value 0. `Register(0)` contains the value 1. The numbers a and b are waiting in the input queue. Explain what result is in `Register(3)` after the execution of the following program:

```
1 Read into Register(1)
2 Register(1) ← Register(1) * Register(1)
3 Read into Register(2)
4 Register(2) ← Register(2) * Register(2)
5 Register(3) ← Register(1) + Register(2)
```

Similarly to cooking, it is not sufficient to be able to execute some instructions only. We also need tests that decide about how to continue in the work. For this purpose, we present the following two simple basic operations:

[3] To compute a root of a number is not a basic instruction of a computer and we introduce it only because we need it for solving quadratic equations. On the other hand, there is no doubt that a computer can compute a root of a number, but to do so one has to write a program as a sequence of arithmetic instructions.

[4] Remember that `Register(0)` contains the order of the instruction executed.

(8) `If Register(`n`) = 0, then go to row` j

One has to test the content of `Register(`n`)`. If it is 0, the content of `Register(0)` is overwritten by the value j. This means that the execution of the program is going to continue by executing the instruction of row j. If the content of `Register(`n`)` is different from 0, then the computer adds 1 to the content of `Register(0)` and the work is going to continue by executing the instruction in the next row.

(9) `If Register(`n`)` $\leq$ `Register(`m`), then go to row` j

If the content of `Register(`n`)` is not larger than the content of `Register(`m`)`, then the next instruction to be executed is the instruction of row j. Else the computer is going to execute the instruction of the next row.

The instruction (operation)

(10) `Go to row` j

is an ultimatum to continue the execution of the program in row j.

Moreover, we still need operations for outputting (displaying) the results of the computation.

(11) `Output` $\leftarrow$ `Register(`j`)`

The content of `Register(`j`)` is written (displayed) as the output.

(12) `Output` $\leftarrow$ “`Text`”

The given text between “ ” will be displayed. For instance, the following instruction

`Output` $\leftarrow$ “`Hallo`”,

results in the word “Hallo” being written on the output tape.

The last instruction is

(13) `End`.

This instruction causes the end of the work of the computer on the given program.

Now we are ready to rewrite our algorithm for solving quadratic equations to a program. To make it transparent, we notice the current state of registers in parentheses.

Input: Integers a, b, c
Program:

1 `Read into Register(1)`
{`Register(1)` contains a}

2 `Read into Register(2)`
{`Register(2)` contains b}

3 `Read into Register(3)`
{`Register(3)` contains c}

4 `Register(4)` $\leftarrow$ `2`

5 `Register(5)` $\leftarrow$ `4`

6 `Register(6)` $\leftarrow$ `-1`
{The state of the memory is described in Fig. 2.5}

7 `Register(7)` $\leftarrow$ `Register(2) * Register(2)`
{`Register(7)` contains b^2}

8 `Register(8)` $\leftarrow$ `Register(5) * Register(1)`
{`Register(8)` contains $4a$}

9 `Register(8)` $\leftarrow$ `Register(8) * Register(3)`
{`Register(8)` contains $4ac$}

10 `Register(8)` $\leftarrow$ `Register(7) - Register(8)`
{`Register(8)` contains $b^2 - 4ac$ and so the first step of the method for solving quadratic equations is finished.}

11 `If Register(9)` $\leq$ `Register(8), then go to row 14`
{Since all registers unused up to now contain the value 0, the execution of the program continues in row 14 if $b^2 - 4ac \geq 0$,

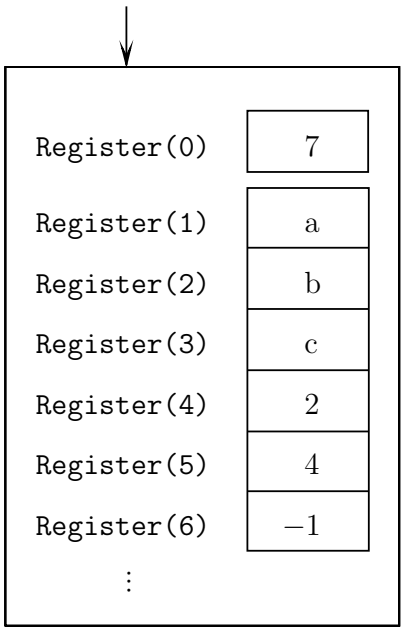

Fig. 2.5

i.e., if the quadratic equation has a real solution. If $b^2-4ac<0$, the computation continues in the next row.}

12 `Output` ← "`There is no solution.`"

13 `End`
{After displaying "There is no solution", the computer finishes the execution of the program.}

14 `Register(8)` ← $\sqrt{\texttt{Register(8)}}$
{`Register(8)` contains the value $\sqrt{b^2-4ac}$.}

15 `Register(7)` ← `Register(2) * Register(6)`
{`Register(7)` contains the value $-b$. The original content b^2 of `Register(7)` is deleted in this way.}

16 `Register(6)` ← `Register(1) * Register(4)`
{The situation is depicted in Fig. 2.6.}

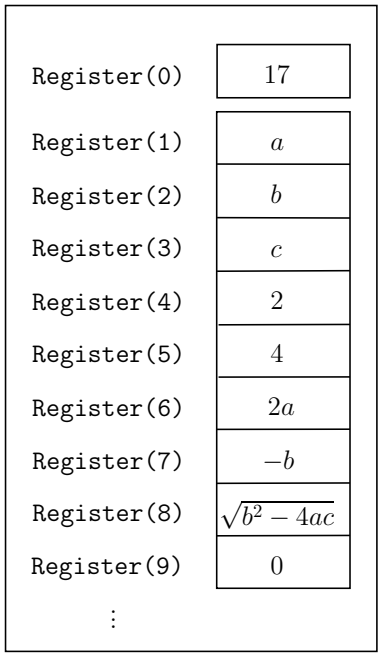

Fig. 2.6

17 Register(11) ← Register(7) + Register(8)

18 Register(11) ← Register(11) / Register(6)
{Register(11) contains the first solution $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$.}

19 Output ← Register(11)

20 Register(12) ← Register(7) - Register(8)

21 Register(12) ← Register(12) / Register(6)
{Register(12) contains the second solution $x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$.}

22 Output ← Register(12)

23 End.

A transparent presentation of this program is given in Fig. 2.7.

Exercise 2.6 Describe the content of all registers after the execution of the whole program!

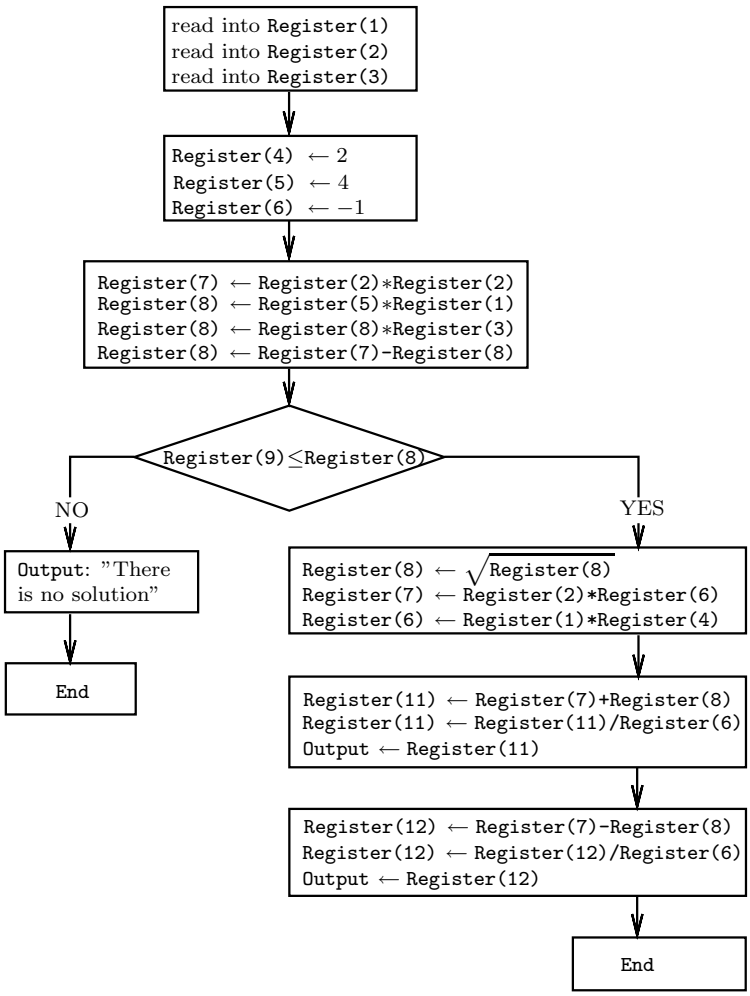

Fig. 2.7

Exercise 2.7 If $b^2 - 4ac = 0$, then there is only one solution $x_1 = x_2$ to this quadratic equation. Modify the presented program in such a way that in this case the program outputs first the text "There is only one solution and this is" and then

the value of x_1. Additionally, in the case $b^2 - 4ac > 0$ the program has to write the text "There are two solutions" before displaying x_1 and x_2.

Exercise 2.8 Explain what the following program computes!

```
1  Read into Register(1)
2  Read into Register(2)
3  Read into Register(3)
4  Register(4) ← Register(1) + Register(2)
5  Register(4) ← Register(3) + Register(4)
6  Register(5) ← 3
7  Register(6) ← Register(4) / Register(5)
8  Output ← Register(6)
9  End
```

Use a table to transparently depict the contents of the registers after the execution of the particular instructions of the program.

Exercise 2.9 Write a program that, for a given integer x, computes the following value

$$3x^2 - 7x + 11 \ .$$

Exercise 2.10 Write a program that, for four given numbers a, b, c, and x, computes the value

$$ax^2 + bx + c \ .$$

Exercise 2.11 Write a program that, for any four given integers a, b, c, and d determines and outputs the maximum of these four values.

It is always necessary to **implement** a method into a program to be able to see that the method is really an algorithm. For instance, if we see that the arithmetic operations and number comparisons are sufficient for performing the method for solving quadratic equations, then we are allowed to call this method an algorithm for solving quadratic equations. *Programming as rewriting of a method into a program is considered as a translation of an algorithm into the computer language.* From the formal point of view this transformation can be viewed as a proof of the automatic executability of the algorithm described in a natural language.

2.4 How Can the Execution of a Program Unintentionally Become a Never-Ending Story?

One of our most important demands on the definition of an algorithm for a computing task is that the algorithm finishes its work for any input and provides a result. In the formal language of computer science, we speak about **halting**. If an algorithm A finishes its work on an input (a problem instance) in a finite time, then we say that the **algorithm A halts on x**. In this terminology, we force to halt an algorithm on every possible input and in such a case we say that **A always halts**.

One could say now: "This is obvious. Who can be interested in developing programs for solving problems that work infinitely long and do not provide any output?". The problem is only that the algorithm designer or a programmer can unintentionally build a program that gets into an infinite repetition of a loop. How can one expect such a mistake from a professional? Unfortunately, this can happen very easily. The programmer can forget about a special situation that can appear only under very special circumstances. Let us return to our cooking algorithms to see how easily a mistake leading to an infinite work can happen.

We want to make tea by heating water and then putting the tea into the water. Our aim is to save energy and so to avoid letting the water boil for more than 20 seconds. Starting with these requirements, one can propose the cooking algorithm presented in Fig. 2.8.

At first glance, everything looks alright and works until a climber wants to apply this cooking recipe for preparing tea on the top of the Matterhorn on some afternoon. Do you already see the problem? Water boils at a lower temperature than 100°C at this altitude and so it may happen that it never reaches 100°C. Thus, the answer of our test will always be "NO". In reality the water won't boil forever, because eventually the fuel will run out or the water will completely vaporize.

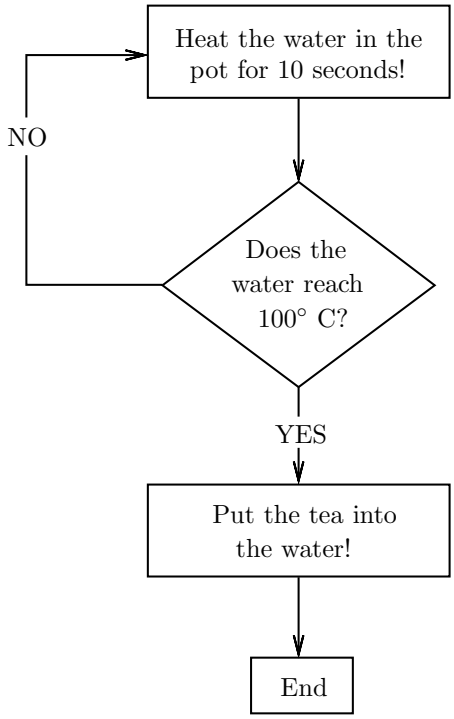

Fig. 2.8

We already see where the mistake happened. Writing the recipe, one forgot to think about this special situation, where the atmospheric pressure is so low that the water cannot reach 100°C. And the same can happen to anybody, if one does not think about all possible special problem instances of the given task and about all special situations that can occur during the computation. The following example shows such a case for a computer program.

Example 2.4 Assume, before starting the program, `Register(0)` contains 1 and all other registers contain 0. The integers a and b are waiting in the first input queue. We consider the following program.

```
1 Read into Register(1)
```

```
2 Read into Register(2)
3 Register(3) ← -1
4 If Register(1) = 0, then go to row 8
5 Register(1) ← Register(1) + Register(3)
6 Register(4) ← Register(4) + Register(2)
7 Go to row 4
8 Output ← Register(4)
9 End
```

The corresponding graphic representation of this program is presented in Fig. 2.9.

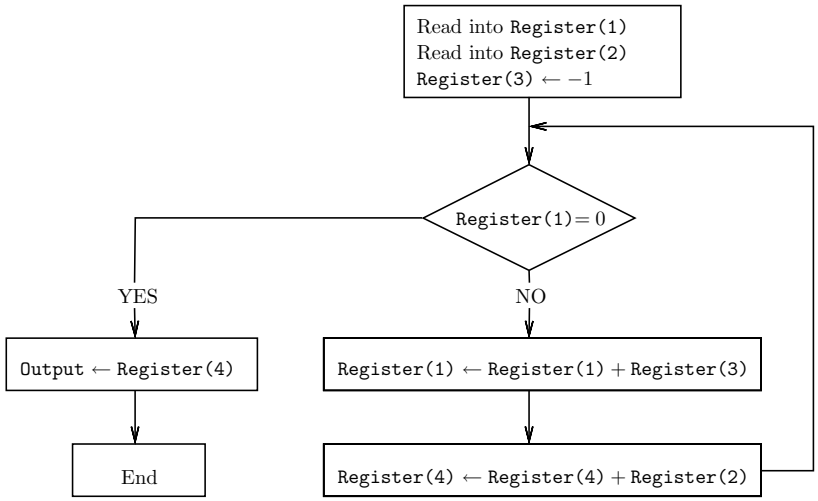

Fig. 2.9

The goal of this program is to compute $a * b$. The strategy is to compute $a * b$ in the following way

$$\underbrace{b + b + b + \ldots + b}_{a \text{ times}},$$

i.e., to sum a many values b.

Exercise 2.12 Assume $a = 3$ and $b = 7$. Execute the computation of the program on this input. Depict a table that shows the content of all registers after particular steps.

If $a = 0$, then the result has to be $a \cdot b = 0$. The programs work correctly for $a = 0$, because a is read into `Register(1)` and the test in row 4 leads directly to row 8, in which the value 0 as the content of `Register(4)` is displayed.
If $a \geq 0$, the execution of the instruction in row 6 causes that b is added to the content `Register(4)` that has to contain the final result at the end of the computation. The execution of the instruction in row 5 results in a decrease of the content of `Register(1)` by 1. At the beginning `Register(1)` contained a. After the i-th execution of the loop (see Fig. 2.9) for an $i < a$, `Register(1)` contains $a - i$ and `Register(4)` contains the number

$$\underbrace{b + b + \ldots + b}_{i \text{ times}} = i \cdot b.$$

If "`Register(1) = 0`", we know that the loop was executed exactly a times and so `Register(4)` contains the value

$$\underbrace{b + b + b + \ldots + b}_{a \text{ times}} = a \cdot b.$$

In this way we developed a program that can multiply two integers without using the operation of multiplication. This means that removing multiplication from the list of our basic instructions does not decrease the power of our algorithms and so does not affect our notion of "algorithm".

But the program in Fig. 2.9 has a drawback. At the beginning, we said that a and b are integers. What does the program do if a or b is negative? If b is negative and a is positive, then the program works orderly. But if a is negative, the content of `Register(1)` will never[5] be 0 and so the loop will be repeated infinitely many times. □

[5] At the beginning `Register (1)` gets the negative value a that is only going to be decreased during the run of the program.

Exercise 2.13 How do we proceed in order to modify the program in Fig. 2.9 to a program that correctly multiplies two integers a and b, also when a is negative?

Exercise 2.14 Try to write a program that computes the sum $a + b$ for two given natural numbers a and b and that uses only the new arithmetic instructions

```
Register(i) ← Register(i)+1
Register(j) ← Register(j)-1,
```

which increase respectively decrease the content of a register by 1. All other arithmetic operations are not allowed and the only allowed test operation is the question whether the content of a register is 0 or not.

Finally, one can see that all algorithms can be implemented as programs that use the test on 0, addition by 1, subtraction by 1, and some input/output instructions only. Therefore, there is no doubt about the automatic executability of algorithms.

Only for those who want to know the whole list of basic computer instructions we present more details. First of all, computing the root of a number is not a basic instruction. To compute the root of an integer, one has to write a program that uses the arithmetic operations $+, -, *$, and / only. Since this requires a non-negligible effort we forswear developing such a program here.

On the other hand, some indispensable basic instructions are still missing. To see this, consider the following task. The input is a sequence of integers. We do not know how many there are. We only recognize the end of the sequence by reading 0, which can be viewed as the endmaker of the sequence. The task is only to read all integers of the sequence and to save all of them in `Register(101)`, `Register(102)`, `Register(103)`, etc., one after another. This saving is finished when the last integer read is 0. One could start to design a program as follows:

```
1 Read into Register(1)
2 If Register(1) = 0, then go to row □
3 Register(101) ← Register(1)
4 Read into Register(1)
```

```
5 If Register(1) = 0, then go to row □
6 Register(102) ← Register(1)
7 Read into Register(1)
8 If Register(1) = 0, then go to row □
9 Register(103) ← Register(1)
  ⋮
```

We always read the next number into `Register(1)` and if the number is different from 0, then we save it in the first free register after `Register (101)`. The problem is that we do not know how to continue in writing the program. If the input queue contains $17, -6$, and 0, then we are already done. If the queue contains 1000 integers different from 0, then this program has to have 3000 rows. But we do not know when to stop writing the program and so we do not know where to put the row with the instruction `end`. We used the notation □ in the program, because we did not know where to put `end`. Certainly we are not allowed to write an infinite program. A common idea in similar situations is to use a loop. One could try to design a loop such as that in Fig. 2.10.

The only problem is that we do not know in which `Register(`$\triangle$`)` the actual integer has to be saved. Clearly, we cannot always use the same integer $\triangle$, because we want to save all integers. We know that we want to save the integer at the address $100 + i$ in the i-th run of the loop. But we are not able to do this, because our instructions allow us to use a fixed address for $\triangle$ in `Register(`$\triangle$`)`.

Therefore, we introduce new instructions that use so-called indirect addressing. The instruction

(14) `Register(Register(`i`))` ← `Register(`j`)`

for positive integers i and j means that the content of `Register(`j`)` has to be saved in the register, whose address is the content of `Register(`i`)`.

Is this confusing? Let us explain it transparently using an example. Assume that the content of `Register(3)` is 112 and that

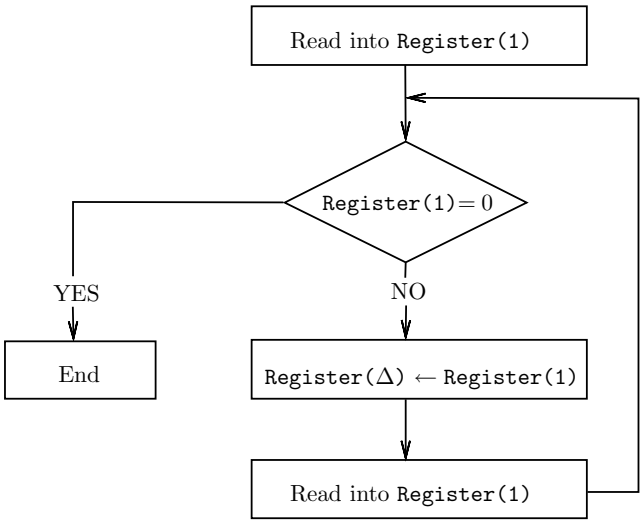

Fig. 2.10

Register(7) contains 24. The computer has to execute the instruction

Register(Register(3)) ← Register(7).

First, the computer takes the content of Register(3) and sees that it is 112. Then the computer executes the already known instruction

Register(112) ← Register(7).

In this way, the number 24 (the content of Register(7)) is saved in Register(112). The contents of all registers except for Register(112) remain unchanged.

Exercise 2.15 Most of the computer instructions introduced have a version with indirect addressing. Try to explain the meanings of the following instructions!

a) Register(k) ← Register(Register(m))
b) Register(Register(i)) ← Register(l)*Register(j)

Using indirect addressing one can solve our problem of saving data of unknown number as depicted in Fig. 2.11. We use Register(2)

for saving the address at which the next integer has to be saved. At the beginning, we put 101 into `Register(2)` and then after saving the next integer, we increase the content of `Register(2)` by 1. The number 1 lies in `Register(3)` during the whole computation.

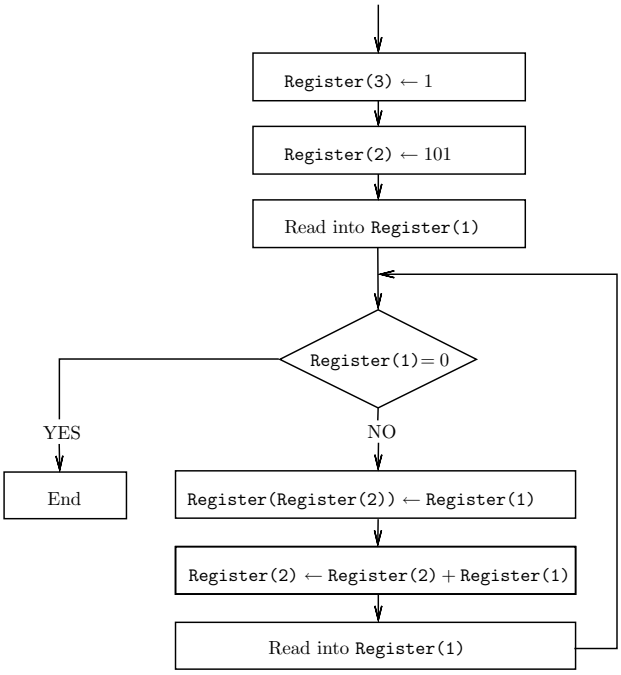

Fig. 2.11

Exercise 2.16 The input queue contains the integer sequence $113, -7, 20, 8, 0$. Simulate step by step the work of the program in Fig. 2.11 on this input! Determine the contents of the registers with the addresses $1, 2, 3, 100, 101, 102, 103, 104$, and 105 after each step! Assume that at the beginning all registers contain 0.

2.5 Summary or What We Have Learnt Here

It does not matter whether you believe it or not, if you were able to solve a few of the exercises formulated above then you have

already programmed and so you have learnt a little bit about what it means to work as a programmer. But this was not the main goal of this chapter.

Our aim was to explain the meaning of the notion of an algorithm. We understand that our expectation on the definition of the notion of an algorithm as a formalization of the notion of a method corresponds to the following requirements:

1. One has to be able to apply an algorithm (a method) even if one is not an expert in solving the considered problem. One does not need to understand why the algorithm provides the solution of the problem. It is sufficient to be able to execute the simple activities the algorithm consists of. Defining the notion of an algorithm, one has to list all such simple activities and everybody has to agree that all these activities are executable by a machine.

2. An algorithm is designed not only to solve a problem instance, but it must be applicable to solving all possible instances of a given problem. (Please be reminded that a problem is a general task such as sorting or solving quadratic equations. A problem instance corresponds to a concrete input such as "Sort the integer sequence $1, 7, 3, 2, 8$" or "Solve the quadratic equation $2x^2 - 3x + 5 = 0$".)

3. We require a guarantee that an algorithm for a problem successfully finds a solution for each problem instance. This means that the algorithm always finishes its work in a finite time and its output corresponds to a correct solution to the given input.

An algorithm can be implemented as a program in a programming language. A program is an algorithm representation that is understandable for the computer. But a program is not a synonym of the notion of an algorithm. A program is only a sequence of instructions that are understandable and executable by a computer. This instruction sequence does not necessarily lead to a reasonable activity. For instance, the execution of a program can lead to

some pointless computations that do not solve any problem or to an infinite work in a loop.

Solutions to Some Exercises

Exercise 2.2 A cooking algorithm for heating 1 l water to 90°C can be described as follows:

1. Pour 1 l water into the pot T.
2. Put the pot T on the hotplate for 15 seconds and then take it away.
3. If the temperature of the water is at least 90°C, finish the work! Else continue with step 2.

Exercise 2.3 The development of memory can be transparently represented using the following table:

	1	2	3	4	5
Input queue	11, 12, 13	12, 13	12, 13	13	
Register(0)	1	2	3	4	5
Register(1)	0	11	11	11	11
Register(2)	1117	1117	100	100	13
Register(3)	21	21	21	12	12
Register(4)	0	0	0	0	0

The first column represents the state of the memory before the execution of the program started. The $(i + 1)$-th column describes the solution immediately after the execution of the i-th instruction and so before the execution of the $(i + 1)$-th instruction.

Exercise 2.8 The given program reads the three integers from the input queue (rows 1, 2, and 3). Then it computes their sum and saves it in `Register(4)` (program rows 4 and 5). In the program rows 6 and 7 the average of the input integers is computed and saved in `Register(6)`. The instruction in row 8 displays the average value. The following table shows the development of the computer states after particular steps. In contrast to the table in Exercise 2.3, we write the values in the table only when the content of a register has changed in the previous step.

Input	a, b, c	b, c	c							
Register(0)	1	2	3	4	5	6	7	8	9	10
Register(1)	0	a								
Register(2)	0		b							
Register(3)	0			c						
Register(4)	0				$a+b$	$a+b+c$				
Register(5)	0						3			
Register(6)	0							$\frac{a+b+c}{3}$		
Output									$\frac{a+b+c}{3}$	