Session: Combinations

2005-08-31 PM

Explaining and Understanding LOGO Programs, a Discipline of Learning Computer Programming

Gerald Futschek

Institute of Software Technology and Interactive Systems
Vienna University of Technology

A 1040 Vienna, Favoritenstrasse 9, Austria

futschek@ifs.tuwien.ac.at

Abstract

Usually students and teachers have to speak about their computer programs while teaching or learning programming. Also software engineers have to explain their programs to their colleagues during the software development process. But "how to explain programs" is usually not taught and there exists no systematic methodology to do it. We show in this article how small LOGO programs can be explained systematically. We distinguish between the operational and the descriptive view on programs. It is argued that the question "why does the program the desired task?" is the most important part of explaining a program. Teachers and students should learn to use specific elements of explaining programs so that they can make better progress in learning programming.

Keywords

explaining programs, program understanding, learning programming

1. Introduction

Logo is a programming language that is often taught and learned as first programming language. Usually learning programming is seen as a very hard task and only a small group of people may succeed. It seems that learning programming with Logo is comparable easy especially when the turtle graphics is used. Many practice examples show that programming can be very easily learned with Logo not only by children from 8 years but also by adults of any age. Beginners in programming usually make rapidly progress in learning the concepts of programming while exploring some interesting problems in an application domain.

While programming and understanding the own programs is done relatively easily, it is very hard for nearly all people to explain their programs to other people so that these people understand the program. Also understanding of programs written by other people is often very hard, when they are not explained properly.

Explaining programs to other people is very important but nowhere taught systematically. It cannot be found in curricula and there is no literature about it. What we can find are good

teachers who know how to explain programs and good books that try to explain programs in a good way.

This article is an attempt to summarize some observations and principles in explaining computer programs from my own experience as teacher for programming. Especially when we try to explain programs to beginners in programming, we have to deal with so many aspects that it is wise to do it step by step. We show in this article a concept of explaining programs that is suitable for different levels of understanding.

2. Insight by speaking about programs

Computer programs are seldom self-evident. Most of the programs have to be commented or explained to be understood. Not only teachers have to explain computer programs to their students, also software engineers have to discuss their programs with their colleagues. The activity of explaining programs helps also the explaining person to understand his own program better.

Although the explanation of programs is very useful it seems to be very hard to do it. Explaining and understanding are closely related by didactics: good explanations ease understanding.

A beginner may think that programs are written for a machine and must be understood only by the compiler. No, programs are always written to be understood also by humans. It is a fact that programs are more often read by humans than by the computer (compiler). Especially during writing, debugging and modifying, programs are read by the programmer very often. As a consequence the programs themselves should be written in a style that they can easily be understood by the human reader. Donald E. Knuth developed a method called "Literate Programming" that allows to generate program documentation that explains programs, see Knuth D E (1984) and Thimbleby H (1998 and 2001).

3. What can be explained?

There are many different aspects that can be explained about computer programs. The following table gives an overview about different types of questions and their related topics.

Table 1. Categories of questions and related disciplines

1. What are the commands doing?	Programming Language
2. How to run a program on a computer?	Operating System
3. In which order are the statements executed?	Execution Model
4. What is the problem to be solved?	Specification
5. Why does the program compute the desired result?	Correctness Proof

It is impossible to explain all these aspects at once. Especially beginners need advice in all these aspects. The first three questions deal with the knowledge of the programming systems, the last two questions deal with the problem to be solved.

To become familiar with programming one needs

1. Insight in the programming language
2. Insight in the computer system

3. Insight in the control flow of a program

4. Insight in the problem to be solved

5. Insight in the algorithm to solve the problem

in this order.

It is essential that the learner has enough insight in the used statements of the program before one can discuss the algorithm that is performed by these statements. Of cause an algorithm can (better: should) be explained independent from a computer program, but for showing that the commands perform this algorithm we need the knowledge of the programming language.

Learning computer programming needs some time and

practice of many programming tasks

from simple to more complex tasks

understanding of the computer model

understanding of the underlying algorithms

Learning computer programming is often seen as a difficult task and takes a lot of time. Therefore good explanations may help to increase understandability.

4. Operational versus descriptive view of a program

The two views are described using an example. We want to draw the following flower with a Logo program:

Figure 1. Flower to be drawn by a Logo program

First we need to describe the parts of the flower (**descriptive view**):

the flower consists of a stem and a blossom

the stem has two leaves

the blossom consists of 8 leaves

In a program we have to define the order in which these parts are drawn (**operational view**):

```
to flower                       to flower
  blossom        or               stem
  stem                            blossom
end                             end
```

The operational view describes the order in which the program is executed: first draw the blossom, then draw the stem, or, first draw the stem and the blossom.

To draw the stem with its two leaves we can draw the stem from bottom to top and employ the following program:

```
to stem
  repeat 10 [fd 10 rt 1]
  leaf                        ; right leaf
  repeat 5 [fd 10 rt 1]
  lt 90 leaf rt 90            ; left leaf
  repeat 15 [fd 10 rt 1]
end
```

Using a turtle for drawing we need additional knowledge about the heading of the turtle in order to draw the parts of the stem in the right direction. In our example the left leaf needs a left turn of the turtle before drawing the leaf and a right turn afterwards. The right leaf needs no turning of the turtle at all. This can only be understood when we know how turtle geometry is defined and that a leaf is oriented 45 degrees to the right. So explanations need to cover also implementation dependent parts that are not part of the problem to be solved.

```
to leaf
  repeat 2 [repeat 90 [fd 1 rt 1] rt 90]
end

to blossom
  repeat 8 [leaf rt 45]
end
```

5. The need of a descriptive view

A binary tree may be drawn by the following Logo program:

```
to tree :l
; draws a tree of size l (actually l is the length of its trunk)
  if :l < 1 [stop]          ; don't draw too little trees
  fd :l                     ; draw trunk with length l
  lt 45 tree :l/2           ; draw left subtree of half size
  rt 90 tree :l/2           ; draw right subtree of half size
  lt 45 bk :l               ; go to the root of tree
end
```

Figure 2. result of call `tree 260`

An operational explanation is given in the comments of the program. For many of my students this explanation is not enough. They ask: "Why do we have to go to the root of the tree at the end of the program?" The answer often is: "After drawing the left subtree we expect the turtle back at the root of the left subtree so that we can just turn the turtle right 90 degrees to draw the right subtree. We have to fulfil this expectation otherwise the tree is not drawn properly."

If a student has problems to understand the two recursive calls of `tree`, an operational explanation would not help and the explaining person would get into troubles! One has to explain the order of all the recursive calls, which number can be high and gives not much insight in the problem.

A descriptive approach helps whenever we have recursive programs: We just need a specification and a description of the structure of the problem:

1. What is the desired effect of a call of the program? (Specification)

it draws a tree of size l (actually l is the length of its trunk)
at end of the program the turtle is at the root of the tree again

2. What is the structure of the problem?

A tree of length l consists of

- *a trunk of length l*
  and at top of trunk
  a left subtree of size l/2 turned 45 degrees to the left and
  a right subtree of size l/2 turned 45 degrees to the right

Usually the structure of the problem is used to construct the program. The programmer can decide in this example in which order the tree is drawn. Apparently the program `tree` implements the desired structure of the tree.

Both, the specification and the structure of the problem are descriptive and static, they have nothing to do with the execution of the program, which is operational and dynamic. The descriptive view eases explanations, but needs some experience in understanding it.

Usually the formulation of these static arguments is sufficient for understanding. In theory this static arguments are the basis of an induction proof that shows that the desired structure is achieved for all possible values of the parameters. Additionally the termination of the program must be shown.

We see that it is necessary to explain **why** the program fulfils the specification. It is not enough to show what the program is doing with some input data. The correctness of a program cannot be shown by examination of the program paths because there are a unlimited number of possible paths. Explaining why a recursive program fulfils the specification needs static arguments that must be used in a sort of induction proof to show that the specification is valid for all possible input values.

6. Example "x to the power of n"

Let's give a numeric example. We want to calculate x^n (x to the power of n).

The given program is:

```
to power :x :n
; yields x to the power of n
; precondition: n >= 0
  if :n=0 [output 1]
  output :x * (power :x :n-1)
end
```

If we want to explain the sequence of commands that are executed (operational view) we have to unfold the recursive calls and get a sequence of calls ending with a call x^0.

If we want to explain why `power` calculates x^n then it is wise to apply a static argument:

The mathematical definition is $x^n = \begin{cases} 1 & if \quad n = 0 \\ x * x^{n-1} & if \quad n > 0 \end{cases}$

It is easily seen that the program implements this formula properly, since it is a 1 to 1 implementation of this formula.

Now we want to understand the more efficient program that has two recursive calls:

```
to power :x :n
; yields x to the power of n
; precondition: n >= 0
  if :n = 0 [output 1]
  if even :n [output power :x*:x  :n/2]
  output :x*power :x :n-1
end
```

Unfolding the recursive calls gives no insight at all. In this example we have to describe mathematically the desired result. We just add a third alternative for the special case of even n:

$$x^n = \begin{cases} 1 & if \quad n = 0 \\ \left(x^2\right)^{n/2} & if \quad even(n) \\ x * x^{n-1} & otherwise \end{cases}$$

Again it is easily seen that a static description of the desired result helps to understand the program, since it is just a straightforward implementation of the formula.

7. Summary

In this article is argued that the question "why a program yields the desired results" cannot be argued with operational (dynamic) arguments. To know the order of commands does not give insight to an algorithm. The only way to understand algorithms is to apply descriptive (static) arguments.

This article can also be seen as a sequel in memoriam to Elisabeth Hopfenwieser (†2005) to our article at Eurologo 2003, Hopfenwieser E and Futschek G (2003), where we presented a metaphor for explaining and understanding the execution of a program (operational view). The content of that article is now complemented by the descriptive view.

References

Hopfenwieser E and Futschek G (2003), *Train to program - A Metaphor for Learning Basic Concepts of Computer Programming*, Proceedings of the 9th European Logo Conference, ed. by CNOTINFOR, Coimbra, 210 - 218

Knuth D E (1984), *Literate Programming*, Computer Journal, **27**(2), 97-111

Thimbleby H. and Ladkin P (1998), *A proper explanation when you need one*, Technical Report

Thimbleby H (2001), *Explaining programs reliably*, Middlesex University School of Computing Science Technical Report