Chapter 1

A First Script and Its Implications

While sending messages using direct interaction with a robot is a fun and powerful way of programming robots, it is rather limited as a technique for writing complex programs. To expand your programming horizons, I am going to teach you about the notion of a *script*, which is a sequence of *expressions*, together with all the fundamental concepts and vocabulary that you will need for the remainder of this book. It also serves as a map to subsequent chapters, which will introduce in depth the concepts briefly presented in this chapter.

First, I will show you how you to send multiple messages to the same robot by separating a sequence of messages with semicolons. Then you will learn how to write a script using a dedicated tool called a *workspace*. I will describe the different elements that compose a script and show some of the errors that one can make when writing a program.

1.1 Using a Cascade to Send Multiple Messages

Suppose you want to get your robot on the screen to draw a rectangle of height 200 pixels and width 100 pixels. To do so, you might click on your robot and then start to type the first message, `go: 100`, press the return key, then click on the robot and type the second expression, `turnLeft: 90`, and press the return key, then click on the robot and type the expression `go: 200`, and so on. You will quickly notice that this is truly a tedious way of interacting with your robot. It would be much more convenient if you could first type in all the instructions and then push a button to have the sequence of instructions executed.

In fact, you can send multiple messages to a robot by separating the messages with a semicolon character ;. To send a robot the messages `go: 100`, `turnLeft: 90`, and `go: 200`, simply separate them with a semicolon as follows: `go: 100 ; turnLeft: 90 ; go: 200` (see Figure 1.1). This way of sending multiple messages to the same robot is called a *cascade of messages* in Squeak jargon.

However, the technique of writing a cascade of messages (that is, sending a robot multiple messages separated by semicolons) does not work well for complex programs. Indeed, even for drawing a simple rectangle, the string of messages quickly grows too long, as shown by the second message in Figure 1.1). And there are other concerns as well. For example, programmers take into account issues such as whether they can store a sequence of messages and replay them later and whether they can reuse their messages and not have to type them in all the time. For all these reasons, we need other ways to program robots. The first way that you will learn is to write down a sequence of messages, called a *script*, in a text editor and ask the environment to execute your script.

1.2 A First Script

The BotInc environment provides a small text editor, called the Bot Workspace, which is dedi- cated to script execution (that is, executing the expressions that constitute a script). Click on the bottom flap, called Working. By default, it contains a Bot Workspace editor, as shown in Figure 1.2.

I will start off by writing a script that draws a rectangle, and then I will

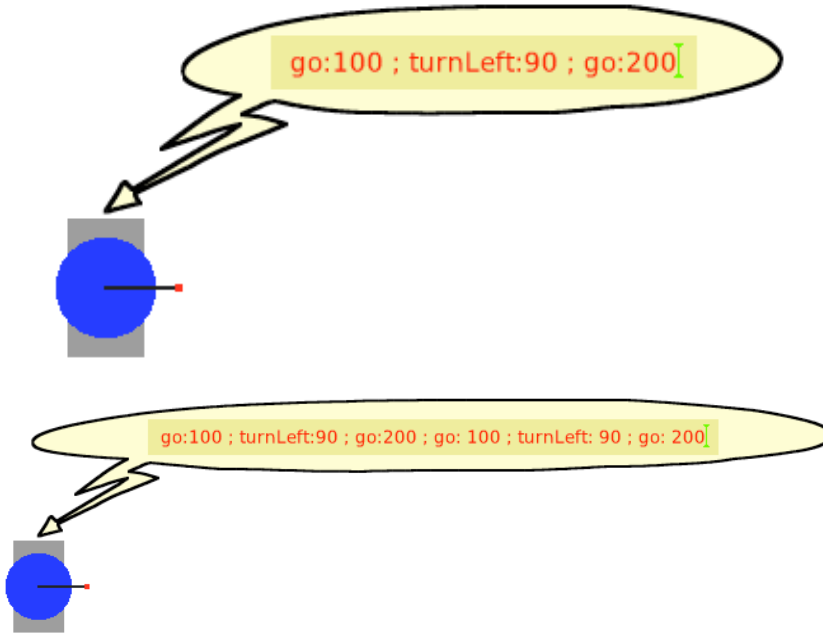

Figure 1.1: You can send several messages to a robot at once using the semicolon character (;).

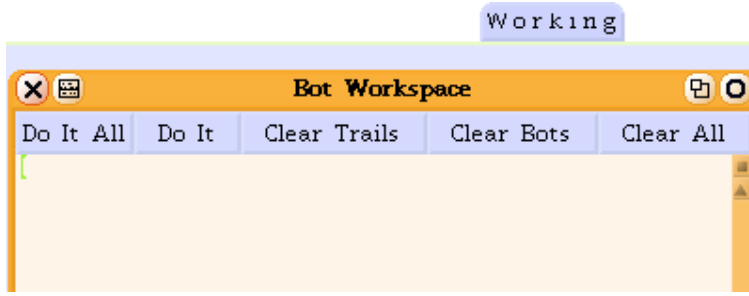

Figure 1.2: A Bot workspace is a small text editor dedicated to the execution of robot scripts.

explain it in detail (Script 1.1).

Script 1.1: *The robot pica is created and is made to move and turn.*

```
| pica |
pica := Bot new.
pica go: 100.
```

```
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90
```

Figure 1.3 shows the script in a Bot workspace and the result of its execution obtained by pressing the Do It All button. Try to get the same result: type the script and press the Do It All button. I have named the robot pica as short for Picasso, since our robots are drawing pictures, just like those of the great Spanish artist.

The Do It All button of the Bot workspace executes *all* the messages that the workspace contains. Therefore, before typing a script, make sure that no other text is already present in the Bot workspace. Moreover, computers and programming languages cannot deal with even the most obvious mistakes, so be careful to type the text exactly as it is presented in Script 1.1.

For example, you must type the uppercase "B" of Bot on the second line, and you must end each line with a period. (There is no need to put a period at the end of the last line, because periods separate messages in Squeak. There is also no need for a period after the first line, because it doesn't contain a message.) But more on that a bit later in the chapter. The script and its result are shown in Figure 1.3.

1.3 Squeak and Smalltalk

Script 1.1 is admittedly simple, but nonetheless, it constitutes a genuine computer program. A *program* is a list of *expressions* that a computer can execute. To define programs we need programming languages, that is, languages that allow programmers to write instructions that a computer can "understand" and execute.

Programming Languages

A well-designed programming language serves to support programmers in expressing solutions to their problems. By support, I mean that the language should, among other things, facilitate expression of the task to be performed, provide efficient execution of the program code and reliability

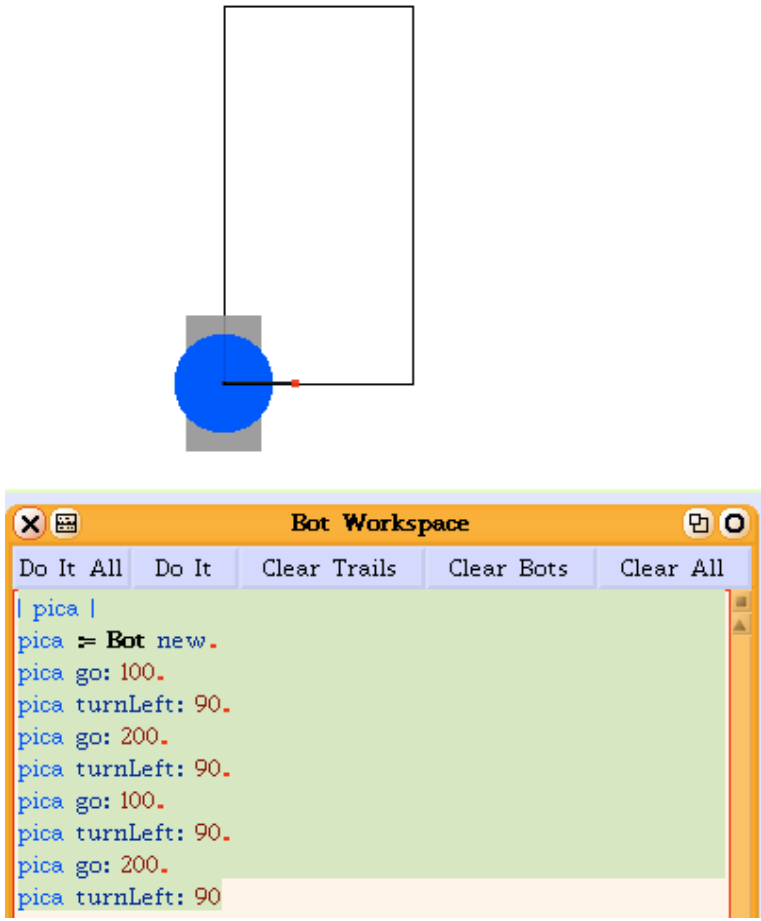

Figure 1.3: A script executed using the `Do It All` button of the Bot workspace and its result.

of the resulting application, give the programmer the ability to prove that programs are correct, encourage the production of readable code, and make it easy for programmers to make changes in their applications. There is no "best" or ideal programming language that satisfies all of these desirable properties, and different programming languages are best suited for different kinds of tasks.

Smalltalk and Squeak

This book will teach you how to program in the Smalltalk *programming language* within the Squeak *programming environment*. A programming environment is a set of tools that programmers use to develop applications. Squeak contains a large number of useful tools: text editors, code browsers, a debugger, an object inspector, a compiler, widgets, and many others. And that's not all! In the Squeak environment you can program music, animate flash files, access the Internet, display 3D objects, and much more. However, before you can start pro- gramming complex applications, you have to learn some basic principles, and that is the purpose of this book.

Squeak programmers develop their applications by writing programs using the programming language called Smalltalk. Smalltalk is an *object-oriented* programming language. Other object-oriented programming languages are Java and C++, but Smalltalk is the purest and simplest. As the term "object-oriented" suggests, such programming languages make use of objects. The objects that are created and used are, of course, not real objects, but logical structures, or "virtual" objects, within the computer. But they are called objects because it is useful to think of these structures as manufactured contraptions, such as a robot, for example, that are able to understand messages that are sent to them and to execute whatever instructions are contained in those messages. The point of the object analogy is that we can use a robot, or a radio, or a camera, without understanding its internal structure. We need only know how to use it by pushing its buttons or sending it messages via the remote control.

Where do manufactured objects come from? A factory, of course. The factories used to create objects are called classes in object-oriented programming languages. Defining classes is somewhat tricky, as is object-oriented programming in general, so in this introductory book I will not show you how to define classes. Instead, you will only define new types of behavior for your robot, and this will give you a good grounding in basic programming concepts.

I chose Smalltalk as the language for this book because it is simple, uniform, and pure. It is pure in that in Smalltalk, *everything* is an object that sends and receives messages to and from other objects. It is simple because in Smalltalk there are only a few basic rules, and it is uniform in that these rules are always applied consistently. In fact, Smalltalk was originally designed for teaching novices how to program. But that doesn't mean that Smalltalk can be used only for writing "baby" or "toy" applications. Indeed, large and complex applications have been written in Smalltalk,

such as the applications controlling the machines that produce the AMD corporation's microprocessors that may be running in your computer.

Another application written entirely in Smalltalk is the Squeak environment itself. Now isn't that interesting! This means that once you develop a good understanding of Smalltalk, you can modify the Squeak environment in order to adapt the system to your own purposes or simply to learn more about the system. With Smalltalk, then, you have quite a bit of power in your hands.

I hope that this discussion about programming languages in general, and Smalltalk in particular, has motivated you to learn how to program. But please be aware that learning to program is like learning to play the piano or to paint in oils. It is not simple, and so do not become discouraged if you have some difficulties. Just as a beginning piano player doesn't start off with Bach's Brandenburg concertos, and a novice painting student doesn't try to reproduce Michelangelo's Sistine Chapel ceiling, the beginning programmer starts off with simple tasks. I have designed this book so that topics are introduced in a logical order, so that what you learn in each chapter builds on your knowledge from previous chapters and prepares you for the material in the following chapters.

1.4 Programs, Expressions, and Messages

Now we are ready to take a closer look at your first script and explain just what is going on.

Typing and Executing Programs

When you wrote Script 1.1, you typed some text, constituting a sequence of expressions, and then you asked Squeak to execute it by pressing the Do it All button. Squeak executed the sequence of expressions; that is, it transformed the textual representation of your program into a form that is understandable by a computer, and then each expression was executed in sequence. In this first script, executing the sequence of expressions created a robot named pica, and then pica executed, one after another, the messages that were sent to it.

A program in Squeak consists of a sequence of *expressions* that are executed by the Squeak environment. In this book, such a sequence is called a *script*.

A script is a sequence of expressions.

A program is a bit like a recipe for a chocolate cake. A good cake recipe describes all the steps to be carried out in correct sequence: cream the butter and sugar; melt the chocolate; add the chocolate to the butter and sugar mixture; sift in the flour; and so on right through placing the filled cake pans in a 350° oven, cooling the baked cake on a rack, and spreading on the frosting. Enjoy! Similarly, a computer program describes all the steps in sequence needed to produce a certain effect: declare a name for a robot; create a robot with that name; tell the robot to move 100 pixels; tell the robot to turn; and so on.

The Anatomy of a Script

The time has arrived to analyze your first script, which is copied here as Script 1.2.

Script 1.2: *A simple script yet a program*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90
```

In a nutshell, Script 1.2 starts of by declaring that it will be using a *variable* named pica to refer to the robot it creates. Once the robot is created and associated with the variable pica, the script tells the robot to take a sequence of walks to different locations on the screen while turning 90 degrees to the left after each walk. Now let us analyze each line step by step. Don't worry if certain concepts such as the notion of a variable remain a bit fuzzy. Everything will be dealt with in due course, and if not in this chapter, then in a future chapter.

@@stef here@@

`| pica |` This first line declares a variable. It tells Squeak that we want to use the name pica to refer to an object. Think of it as saying to a

friend, from now on I am going to use the word pica in my sentences to refer to the robot that I am about to order from the robot factory. You will learn more about variables in Chapter 8.

`pica := Bot new.` This line creates a new robot by sending the message newto the robot factory (class) named Botand associates the robot with the name pica, the variable that was declared in the previous step. The word Botrequires an uppercase letter B because it is a class, in this case the class that is a factory for producing robots.

`pica go: 100.` In this expression, the message go: 100 is sent to the robot we named pica. This line can be understood as follows: "pica, move 100 units across the computer monitor." It is implicit in this expression that a robot receiving a `go:` message knows in what direction to travel. In fact, a robot is always pointing in some direction, and when it receives a go: message, it knows to move in the direction in which it happens to be point- ing. Note also that the message name go:terminates with a colon. This indicates that this message needs additional information, in this case a length. For example, go: 100 says that the robot should move 100 pixels. The message name is go:.

`pica turnLeft: 90.` This line tells pica to turn 90 degrees to its left (counter-clockwise). This line is again a message sent to the robot named pica. The message name turnLeft: ends with a colon, so additional information is required, this time an angle. The remaining lines of the script are similar.

> Important! Any message name that terminates with a colon indicates that the message needs addi- tional information,such as a length or an angle.For example,the message name turnLeft: requires a number representing the angle through which the robot is to turn counter-clockwise.

About Pixels On a computer screen, the unit of distance is called a pixel. This word was invented in about 1970 and is short for “picture element.” A pixel is the size of the smallest point that can be drawn on a computer screen. Depending on the type of computer monitor you are using, the actual size of a pixel can vary. You can see individual pixels by looking at the screen through a magnifying glass.

Expressions,Messages,and Methods

I have been using the terms expressionandmessage. And now it is time to define them. I will also define the important term method.

Expression Anexpressionis any meaningful element of a program. Here are some examples of expressions: • | pica |is an expression that declares a variable (more in Chapter 8). • pica := Bot newis an expression involving an operation, called assignment, that asso- ciates a value with a variable (see Chapter 8). Here, the newly created robot obtained by sending the message newto the class Botis associated with the variable pica. • pica go: 100is an expression that sends a message to an object. Such an expression is called a message send. The message go: 100is sent to the object named pica. • 100 + 200is also a message send. The message + 200is sent to the object 100. Message Amessageis a pair composed of a message name, also called a message selector, and possible messagearguments, which are the values that the object receiving the message needs for exe- cuting the message. These relationships are illustrated in Figure 2-4. The object receiving a message is called a message receiver. A message together with the message receiver is called a message send. Here are some examples of messages: • In the expression pica beInvisible, the message beInvisibleis sent to a receiver, a robot. This message has no arguments. • In the expression pica go: 100, the message go: 100is sent to a receiver, a robot named pica. It is composed of the method selector go:and a single argument, the number 100. Here, 100represents the distance in pixels through which the robot should move. Note that the colon character is part of the message selector. • In the expression 33 between: 30 and: 50, the message between: 30 and: 50is composed of the method selector between:and:and two arguments, 30and50. This message asks the receiver, here the number 33, whether it is between two values, here the numbers 30 and50. • In the expression 4 timesRepeat: [pica go: 100], the message timesRepeat: [pica go: 100], which is sent to the number 4, is composed of the message selector timesRepeat:and the argument [pica go: 100]. This argument is called a block, which is a sequence of expressions (in this case a single expression) inside square brackets (more on this in Chapter 7). • In the expression 100 + 200, the message + 200is composed of the method selector + and an argument, the number 200. The receiver is the number 100.

Message Separation

As mentioned earlier, each line of Script 2-1, except the first and last, is terminated by a period. The first line does not contain a message. Such

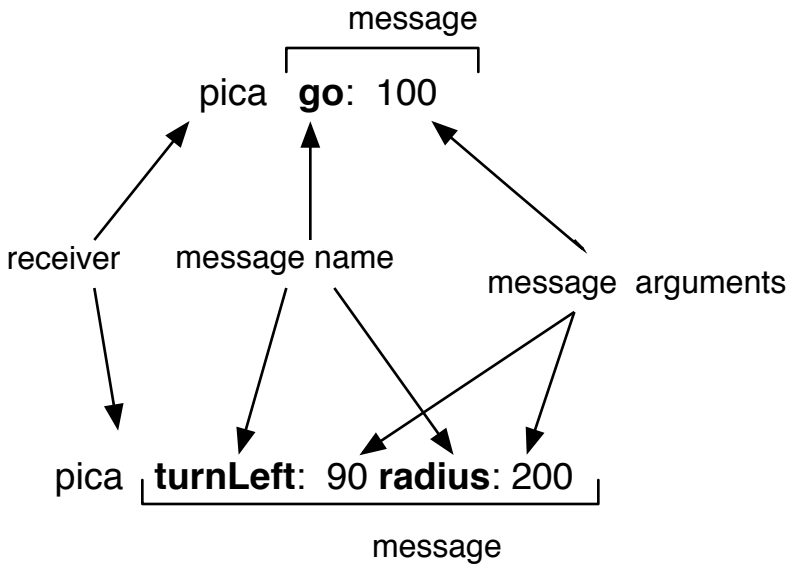

Figure 1.4: Two messages composed of a message name or method selector, and a set of arguments.

a line is called a variable declarationin computer jargon. Thus, we can make the following observation: each message send must be separated from the following one by a period. Note that putting a period after the last message is possible but not mandatory. Smalltalk accepts both.

> Important! Message sends should be separated by a period.The last statement does not require a terminal period.Here are four message sends separated by three periods.

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

> Important! A period character . is a message separator, so there is no need to place one after a message send if there is no following message send. Therefore,no period is necessary at the end of a script or of a block of messages.

Method

When a robot (or other object) receives a message, it executes a method, which is a kind of script that has a name. More formally, a method is a named sequence of expressions that a receiving object executes in response to the receipt of a message. A method is executed when an object receives a message of the same name as one of its methods. For example, a robot executes its methodgo:when it receives a messagewhose name is go:. Thus the expression pica go: 224causes the message receiver picato execute its method go:with argument 224, resulting in its moving 224pixels forward in its current direction. Later in the book, I will explain how you can define new methods for your robot, but for now, we do not need them to start programming.

Cascade

As I mentioned in the first section of this chapter, you can send multiple messages to a robot by separating them with semicolons. Such a sequence of messages is called a cascade. You can also use a cascade in a script to send multiple messages to a robot. Script 2-3 is equivalent to Script 2-2, except that now all the messages sent to the robot picaare separated by semicolons. Using cascades is handy when you want to avoid typing over and over the name of the receiver of the multiple messages. Cascades are useful because they shorten scripts. However, be careful! Shortcuts can lead to trouble if you don't watch your step, so be sure that you truly intend for all your messages to be sent to one and the same receiver.

```
| pica |
pica := Bot new.
pica
go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90 ;
go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90.
```

> Important! To send multiple messages to a robot,use a semicolon character ; to separate the messages,following the pattern aBot message1 ; message2.Here is an example:pica go: 100 ; turnLeft: 90 ;go: 200 ; turnLeft: 90

Creating New Robots

To obtain a new robot, you have to send an order to the robot factory to manufacture one for you. That is, you have to send the message newto the class Bot. There is nothing new here. It is exactly what you did in the previous chapter when you clicked on the blue and orange box namedBot, which represents the class of the same name, and typed newin the bubble. In Squeak, we always send messages to robots, other objects, or classes to interact with them. There is no difference in treatment, except that classes and objects understand different messages. It is the job of classes to create objects. An object does not know how to create other objects, so sending a robot the message newleads to an error. Classes, on the other hand, generally do not have colors and do not know how to move, and so sending the message coloror go: 135to a class does not make sense, and doing so leads to an error. Nonetheless, in both cases you are sending messages!

TheBotclass is not the only manufacturing company in the Squeak environment. There are other classes, and they understand different messages and employ different methods for creating different kinds of objects. For example, the class Colormanufactures color objects. It returns a blue or green color object in response to the message blueorgreen. Whenever in this book a new object must be obtained from a specific class, I will tell you how it is done.

> Important! To obtain a new object from a class,you generally send the message newto the class.Thus Bot newcreates a new robot.Other classes may offer different messages for obtaining new objects.For example,Color bluetells the class Colorto create a new bluecolor object.

1.5 Errors in Programs

Computers are very good at making highly complex calculations at incredible speed, but they lack the intelligence to correct small mistakes. If I accidentally wrote, "now turn on your computer," you might chuckle over my misspelling, but you would have no trouble under- standing what I meant. But computers have no such intelligence, which means that each expression given to a computer must be given precisely, without the least error. The smallest seemingly insignificant mistake in a program, even something as trivial as using a lowercase letter instead of an uppercase one, will almost certainly be misunderstood by the computer. If you have errors in your scripts, two things can go wrong: either an error message will appear on the screen, and this is likely to occur when you are doing your first experiments, or the pro- gram will be executed, but the result will not be what you intended. So when things go wrong, do not despair and try to find the error in your program.

Squeak has a helpful error-prevention and error-correction facility. It colors the letters while you are typing. When a word becomes red, this means that you are writing something that Squeak does not understand. An example is shown in Figure 2-5. When a word is blue, for a variable or a message, or black, for a class, this indicates that everything is structurally correct.

If you attempt to execute an expression containing an error, Squeak tries to help you by notifying you when it encounters the error in your code. The error messages that Squeak uses are actually menus. The top part of the menu window contains a short description of the error; then, depending on the type of error, some suggested corrections may be listed as options. If you don't like any of the options, you can always cancel execution by choosing "cancel" in the menu. Then you should locate the place in your script that Squeak did not understand, correct it, and try again to execute the script. I will now tell you about some of the most common errors.

Misspelling a Message Selector

Misspelling the name of a message leads to an error. In Figure 2-5, I misspelled the message selectorgo:, typing god:instead. The message god:does not exist in Squeak, and therefore Squeak turned the word red. Ignoring Squeak's friendly warning, I tried to execute the script. Squeak tried to guess what message selector I had in mind, and prompted me

with a menu of possibilities. At this point, I could choose the correct message selector (go:), and the message god:will be replaced by go:. Or I can simply choose "cancel." If I take the latter option, I will have to change god:togo:manually.

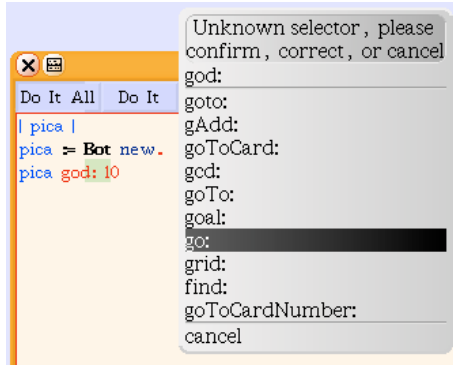

Figure 1.5: We mispell the message go: and type god: instead. The message god: does not exist therefore Squeak prompts us.

Misspelling a Variable Name

There are two ways to misspell the name of a variable: in the body of the script itself and when it is declared (between two vertical bars as in | pica |). Figure 2-6 shows the two cases: In the left-hand figure I declared a variable pica, but then I typed pica1 instead of pica in the script. Squeak noticed that I was trying to use an undeclared variable, so it turned the text red and prompted me, suggesting that I either declare the undeclared variable by declaring pica1 as a new variable, or replace pica1 by pica. Since pica is the variable name that I wanted, and pica1was just a typo, I chose the option pica, as shown in the figure. The right-hand figure shows that I accidentally typed a space between the candainpicawhen I attempted to declare the variable pica. Squeak did not consider this an error. It simply "thought" that I was trying to declare two variables, picanda. Then in the script I typed pica, thinking that I had declared that variable. But Squeak saw that in fact, picawas an undeclared variable, so it turned the text red and gave me some options, including declaring a new variable with the namepicaor else replacing what I had typed with the declared variable pic.

Unused Variables

It may happen that you accidentally declare too many variables. For example, you might declare the variables picaanddaly, thinking that you will need two robots, but then you never usedalyin your script. This is not really an error, and your program will run correctly even if it has declared variables that are ever used. It is analogous to buying two suitcases, just in case, but using only one of them. You simply have some extra baggage around that you are not using. But just in case you really did mean to use dalyand forgot, Squeak checks for unused declared variables and if it finds any, suggests that you might want to remove them. For exam- ple, in Figure 2-7, the script declares the variables picaanddalybut uses only pica. Squeak notices this and asks you whether you would like to remove the unused variable daly.

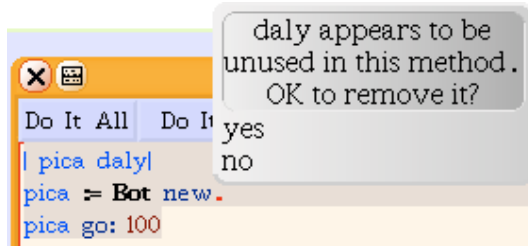

Figure 1.6: All the variables and messages are correct. However the variable daly is defined but not used so Squeak indicates it to us. Unused variables are not a problem so you can proceed.

Uppercase or Lowercase?

Another common mistake is to forget a required uppercase letter. Names of classes begin with an uppercase letter, so don't forget this when you want to send a message to an object factory. Figure 2-8 shows that I unthinkingly typed botinstead of Bot. Squeak tried to figure out what I meant, but it failed, and so none of the options that it offered for fixing the problem will do. In such a case you have to correct the error yourself. In the context of this book, the only classes you have to worry about are Bot, the robot factory, and Color, the color factory.

Forgetting a Period

Finally, one of the most common mistakes, one that even fluent programmers make, is to for- get a period between two message sends or a semicolon between two messages in a cascade. A period indicates that a new message send is about to begin, but without the period, Squeak thinks that the current message is being continued, and that the variable meant to be the message receiver of a new message is just another message selector. Since there is no message selector with the name of one of your variables, Squeak tells you that you have typed an unknown selector and offers you some possible corrections. For example, in Figure 2-9, a period is missing after the expression pica := Bot new, and Squeak tries to parse (that is, fig- ure out the structure of) the message pica := Bot new pica go: 120, and according to the rules of message syntax (structure), about which you will learn in Chapter 11, picashould be a message selector. But such a message selector does not exist, so Squeak protests and proposes some possible replacements. Since you know that picais your declared variable and not a message selector, you realize that you forgot a period and so you select "cancel" and type the period manually.

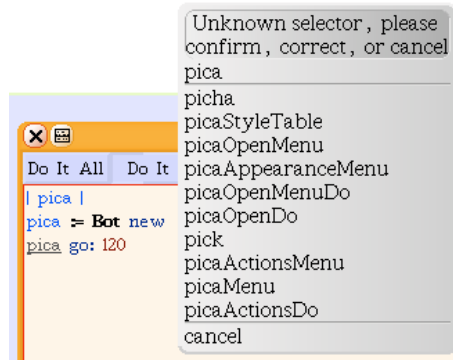

Figure 1.7: Two examples of error. We mistyped the names of the variable. First in the script pica1 has not been declared and second while defining the variable: car o defines two variables car and o but not a variable pica.

Words That Change Color

Squeak tries to identify mistakes while you are typing your scripts. If it detects something fishy, it changes the color of the text and provides

some visual cues that suggest what might be wrong. Figure 2-10 shows some typical situations. Unfortunately, the black-and-white figure does not show its true colors. But use your imagination!

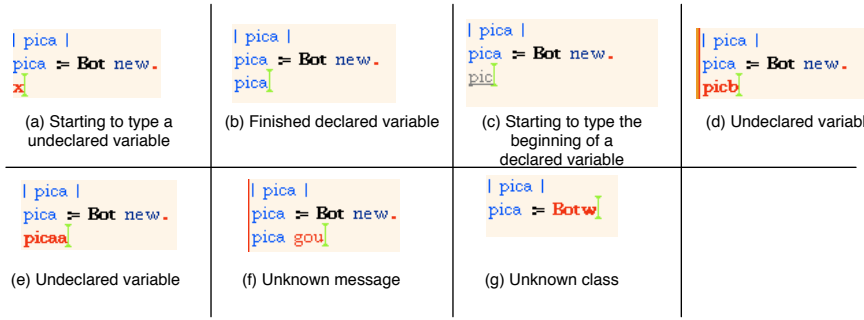

Figure 1.8: Squeak using colors to help finding mistakes.

Here is a key to the figure: (a) I started to type the first letter of an undeclared or unknown variable. Since no variable starting with the letter x has been declared, Squeak turns the x red, letting me know that something is amiss. (b) I finished typing a variable that has been declared. Squeak shows me that I have typed a declared variable correctly by turning the text blue. (c) I am in the process of typing the name of a variable. As long as what I have typed is the beginning of the name of a declared variable, Squeak underlines it to let me know that so far, everything is ok. (d) As soon as I type a character in a variable name that results in a sequence of letters that is not the beginning of the name of a declared variable, Squeak turns the word red. Note the difference with the previous case. In case (c), I could have typed the char- acteraand thereby completed the declared variable pica, as in (a). However, I typed the character b, and ended up with a sequence of letters (picb) that is not the begin- ning of the name of any declared variable. (e) After I typed the name of a declared variable (pica, as in case (b)), I accidentally added an extra character a, which leads to the sequence of letters (picaa), which is not the beginning of the name of a declared variable. (f) Squeak tries to do the same for message selectors as it does for variable names. Here I mistyped the message go:and typed instead gou. Squeak was looking for a message selector, and as soon as I typed the character u, it realized that there is no message selector that begins gou, so it turned the text red.

(g) Squeak tries to do the same for classes as it does for variables and

message selectors. Here I typed the character wafterBot, and Squeak, expecting a class name because of the uppercase B in Botw, indicates by turning the text red that there is no class in the system whose name begins Botw.

1.6 Summary

- To execute an expression. Press the Do It All buttonof the Workspace.
- A script is a sequence of expressions that performs a task.
- A message is composed of a message selector and possibly one or more arguments. Some message selectors do not take arguments, as in the message send pica beInvisible.
- Any message selector that ends with a colon requires additional information (one or more arguments), such as a length or an angle. For example, the message selector turnLeft:requires an argument whose value is a number representing the angle through which the robot should turn counterclockwise.
- To obtain a new object, you generally send the message newto a class. For example, Bot newcreates a new robot. Other classes may understand different messages for produc- ing new objects. For example, Color yellowasks the class Colorto create a new yellow color object.
- A class is a factory for producing objects. Class names always start with an uppercase letter. For example, Botis the factory for creating new robots, and Coloris the color factory. The message Bot new color: Color yellowasks the Bot class to create a new robot, and then the color factory is asked to create a yellow color object. Finally, the messagecolor:is sent to the new robot with the yellow color object as argument, resulting in the new robot having its color changed to yellow.
- Message sends should be separated by a period. A terminal period after the last mes- sage send is not required. Here is an example of four message sends separated by three periods:

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

- To send multiple messages to the same object use a semicolon to separate the mes- sages, as in aBot message1; message2. For example, pica go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90send the sequence of four messages (1) go: 100, (2)turnLeft: 90, (3)go: 200, (4)turnLeft: 90 to the robot named pica.