
Brainstorming Draft

Draft

Stéphane Ducasse
stephane.ducasse@free.fr

©-2006

To

April 8, 2006

Contents

| | | |
|-----------|---|-----------|
| I | Joe the Miner | 7 |
| 1 | Bots, Inc. | 11 |
| 1 | Getting Started | 11 |
| 2 | The Robot's World | 12 |
| 3 | Robot Behavior | 13 |
| 4 | First Scripts and Methods | 16 |
| 5 | Creating and Editing Bot's Areas | 18 |
| 2 | Learning Conditions with Bot | 21 |
| 1 | A Simple Example | 21 |
| 2 | The Need for ifTrue:ifFalse: | 24 |
| 3 | Nested Conditions | 27 |
| 4 | About Method Returned Values | 28 |
| 5 | Final Experiments | 29 |
| 6 | Summary | 30 |
| 3 | Conditional Loops | 31 |
| 1 | Conditional Loops | 31 |
| 2 | About the Use of [] | 33 |
| 3 | Learning from Errors | 33 |
| 4 | Practicing | 35 |
| 5 | Experiments: Finding a Yellow Spot | 35 |
| 6 | Summary | 38 |
| II | Advanced Concepts and Fun Projects | 41 |
| 4 | Paths and Mazes | 43 |
| 1 | Defining a Path | 43 |
| 2 | Defining a Strategy | 43 |
| 3 | Escaping a Maze: Following a Wall | 47 |
| 5 | Advanced Path Finder | 53 |
| 1 | Keep Track and Escaping Traps | 53 |
| 2 | Towards a Solution | 53 |
| 3 | Implementing the Solution: Managing Signs | 55 |
| 4 | Some Work Left | 61 |

| | | |
|-----------|--|------------|
| 6 | A Quick Look at Recursion | 63 |
| 1 | Picking Diamonds | 63 |
| 2 | Learning from Errors | 64 |
| 3 | Fun with Recursion | 65 |
| 4 | About Keeping Context | 70 |
| 5 | A More Traditional View on Recursion | 73 |
| 6 | The Subtle Difference of Accumulating Intermediate Results | 75 |
| 7 | Collections: Grouping Objects | 79 |
| 1 | A First Look at Collection | 79 |
| 2 | Grouping Arguments | 83 |
| 3 | Iterating over a Collection | 84 |
| 4 | Collection as Method Arguments | 86 |
| 5 | Translating and Tiling | 86 |
| 6 | Other Important Operations on Collections | 90 |
| 8 | Block: Messages with Delayed Execution | 95 |
| 1 | An Example | 95 |
| 2 | Block Definition | 95 |
| 3 | Invoking Block Execution | 97 |
| 4 | About Returned Value of Blocks | 98 |
| 5 | Other Loops with Variables | 100 |
| 9 | Fun with Collections | 103 |
| 1 | Programming Fractal Drawings | 103 |
| 2 | Dot to Dot | 105 |
| 3 | Follow | 112 |
| 10 | Simple L-Systems and Fractal Production | 117 |
| 1 | L-Systems | 117 |
| 2 | The Graphical Interpretation of L-Systems | 119 |
| 3 | Implementing the L-System | 121 |
| 4 | L-Systems with a Single Loops | 123 |
| 5 | A Gallery of 1-rule based L-Systems | 124 |
| 6 | Experimentation | 127 |
| 7 | Analysis of the Solution | 127 |

Warning

This is a very draft versions of some chapters that were removed from the book. I plan to proofread it, ask a professional to copy-edit them and give them for free. Pay attention that the Turtle class corresponds to the Bot class of the book and that the Bot class in this chapter are called Miner in the associated image. I changed the name of these small guys to please my previous editor.

Do not hesitate to contact me if you have problems and find bugs Stef (stephane.ducasse@free.fr)

Part I

Joe the Miner

For that purpose we introduce a new environment in which you will be able to program robots to execute some tasks such as finding diamonds or escaping mazes. Playing with bots cover again all the previous concepts we explained but in a different context. Therefore this is really an opportunity to evaluate whether you understand well the concepts explained previously. Moreover with robots you will learn how to program with conditions.

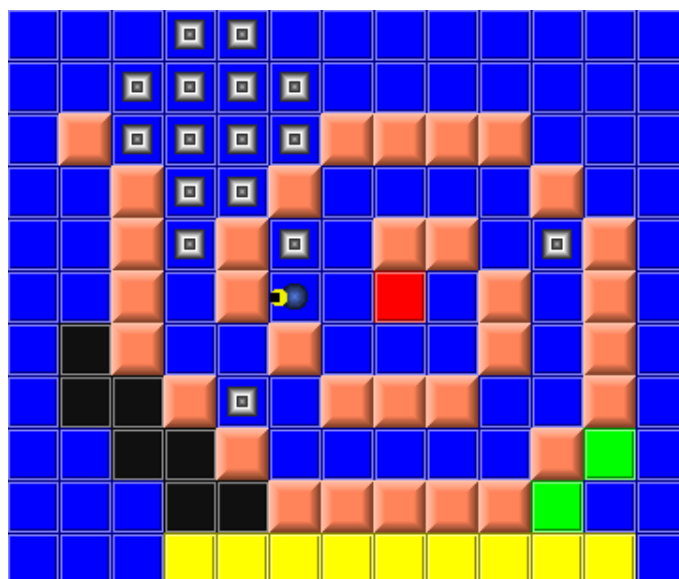
Helper's Hints

The idea of using a robot is not new and has been used since decades with Joseph and Karel the robot. Both were and are still used to teach procedural thinking and in its latest incarnation used to teach object-oriented programming. We think that kind of exercises that can be proposed by such robots are limited that's why we did not based this book entirely on it. However, the metaphor of the robot with sensors and a limited environment supports extremely well the presentation of loops, conditional statement and conditional loops. This is the reason why we designed and introduced a complete world for steering robots. Note that we designed also some chapters on conditional programming with the turtles if you do not want to use a new environment.

Helper's Hints

Note that for readers not willing to use the robot this part also contains two chapters explaining those concepts using turtles.

Bots, Inc.



In this chapter we describe the environment of the robots, their behavior and capacities. We show how simple scripts can be defined using the environment and how they can be turned into methods as shown in Chapter ??.

1 Getting Started

To create a robot world grab and place on the Squeak desktop the first thumbnail of the orange flap named Bot World as shown by Figure 1.4) or execute the script 1.1 in a workspace.

Script 1.1 (*Opening the World of Bot*)

```
BotWorldBoard newStandAlone openInWorld
```

You should obtain an environment similar to the one displayed in the first figure of the chapter. The bottom button bar allows you to access the functionality of this environment as shown by Figure 1.1.

The Bot World window has the following buttons from left to right:

- **Add Bot.** Add a new robot in the current area. Once this button pressed the user is asked to give a name for the newly created bot. This name is also used as a variable to send messages to the robot using a robot Controller (see Figure 1.8). A robot appears on the starting place (the red tile) of the board.



Figure 1.1: The functionality of the bot environment.

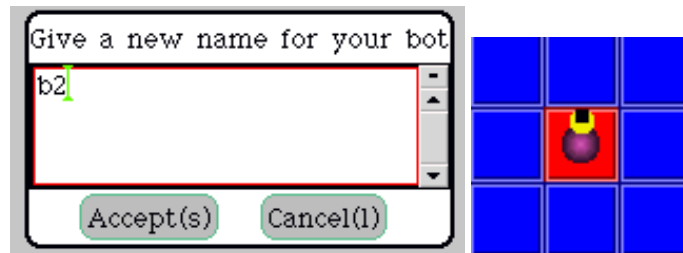


Figure 1.2: Giving a name to a new robot and getting a new robot on the starting place.

- **Controller.** Open a robot Controller, *i.e.*, a dedicated workspace in which we can send messages to the robots by using their names (see Figures 1.3 and 1.8).
- **Restart.** To reset the area but without changing the bots that are already created.
- **Pick Area.** To select a new area among the list of all the areas defined.
- **Quit.** To quit and close the playing board.

2 The Robot's World

The world of our robots is an area composed of different kind of tiles as shown by the first figure of this chapter.

- Blue tiles represent the ground. Robots can freely walk on them.
- Red tiles represent the *starting place*. This is on this tile that new robots appear. There is one and only one red tile per area. When you will be defining new areas you should always specify a starting place.
- Black, yellow, magenta, and green tiles are just painted ground tiles. However the robots have sensors to know if they are on of such tiles.

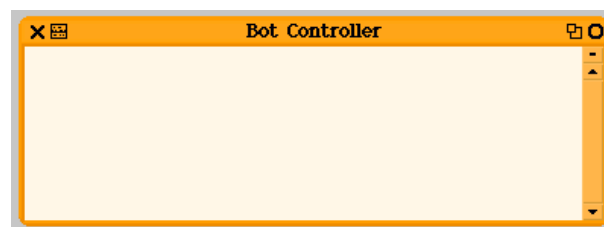


Figure 1.3: Opening a bot controller by pressing the button 'Controller'.



Figure 1.4: The flap containing the environment and the tools to program the robots.

- Painted tiles (blue, red, black, yellow, and green) can contain diamonds and robots can drop diamonds on them.
- Light brown tiles represent bricks. Robots are blocked by bricks and therefore cannot walk on them. A robot has sensors that indicates it whether it is facing a wall.

A robot can walk and drop diamonds on all the tiles except the bricks. It cannot walk outside the world limits.

3 Robot Behavior

A robot is always on one tile and can only move from one tile to another one. Note that each tile has a location and that you can see the bot location by moving the mouse on the bot and waiting for a balloon to show up as shown in Figure 1.6. A bot can only move forward and turn in four directions as shown by Figure 1.5 but we will show how to define more advanced operations such as moving back in the future.

Moves and Directions

A robot understand the following messages:

- **go**. In response to this message, the receiver moves by one tile in the direction in which it is pointing at. What is *really* important is that we do not want to damage our robot by making it

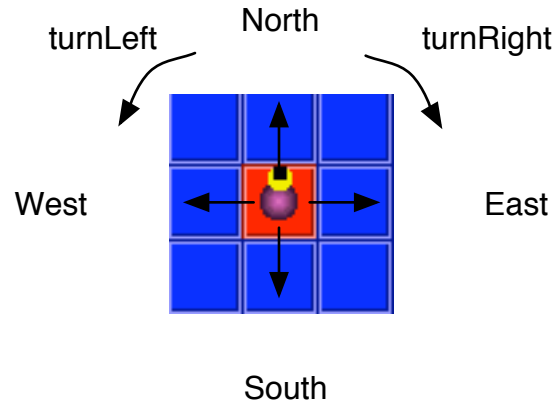


Figure 1.5: Possible robot movements.

crashing into a wall. Therefore you have check using the method `canMoveForward` if the move is possible as we will show later.

- `turnLeft` and `turnRight`. In response the receiver changes its direction to the left or right relative of the current direction.
- `north`, `south`, `east`, and `west`. In respond the receiver changes its direction to point to the corresponding directions.



Figure 1.6: Displaying some information relative to a bot.

As we mentioned it, a robot should not bumped into a wall, else an error occurs as shown by Figure 1.7. A correct program should never such a kind of errors. Similarly, when a robot picks a diamond or drop one, it should check if it can do it. The following chapter explain how this can be done using conditional. When such an error occurs just close the window by clicking on the cross at the left top corner of the window.

Helper's Hints

It can be annoying to let the robots making noise while moving. You can indicate that all the robots have to

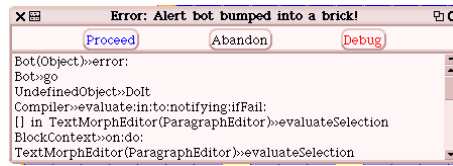


Figure 1.7: When a robot bumped into a wall.

move silently by executing the expression `Bot silent`. Read the Chapter ?? to have more information.

Helper's Hints

Sensors

A robot has several sensors. The following messages to check their status:

- `canMoveForward`, `canMoveLeft`, and `canMoveRight` return whether the bot can move forward, left, or right.
- `isOnBlack`, `isOnMagenta`, `isOnGreen`, `isOnYellow`, and `isOnRed` return whether the bot is on a colored tile of the given color.
- `canPick` returns whether the robot is above a diamond and `canDrop` returns whether the the robot can drop a diamond.
- `isAtHome` returns whether the bot at the starting place.

Figure 1.6 shows that you can get some information on a bot by letting the mouse over it to get a balloon.

Managing Diamonds

A robot can pick or drop diamonds executing the following methods:

- `pick` picks up a diamond. Again you will have to pay attention using the method `canPick` that before picking a diamond that there is effectively one. The method `pick` raises an error when there is no diamond.
- `drop` drops a diamond on the current location if the robot has still one diamond, else it raises an error. The method `canDrop` indicates wether you can drop a diamond.
- `diamNumber` returns the number of diamonds that a robot is carrying. Note that you can also load a robot with a number of diamonds using the method `loadWith: anInteger`.

Note that a robot can walk on tiles containing diamonds without problems.

Finally a robot can paint tiles. Ask a robot to paint a tile of a given color using the messages `paintGreen`, `paintBlue`, `paintYellow`, `paintMagenta`, and `paintBlack`. Note that the starting place and bricks cannot be painted.

Helper's Hints

The design of the robot behavior forces robot programmers to make tests for picking and dropping a diamond or before moving forward. We designed it especially to have this behavior. However, if you do not want to have the students testing for example before dropping a diamond. As explained in Chapter ?? you just have to define the method `safeDrop` defined in the class `Bot` as follows:

Method 1.1

```
Bot>>safeDrop

self canDrop
  ifTrue: [self drop]
```

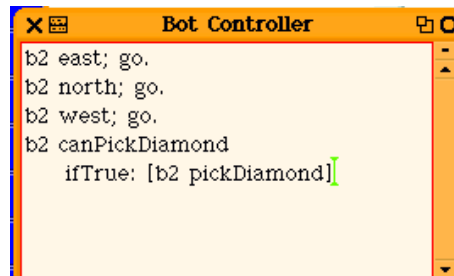
Helper's Hints

Figure 1.8: Steering a robot by sending it messages.

4 First Scripts and Methods

Using a Bot Controller we can send messages to a bot using its name. There is no need to declare a variable to refer to the bot and to initialize it as we were used to do with the turtle. This is because a Bot Controller is a special tool that automatically declares as variables all the robots defined in a given playing area.

Select the area named `area02` to have more space, create a robot, name it `b2`, and try the following script (script 1.2):

Script 1.2 (A Simple Script)

```
b2 west.
b2 go
```

Note that the script 1.2 can be expressed using cascade ; to avoid to repeat unnecessary the receiver as shown by the script 1.3.

Script 1.3 (A Simple Script using a Cascade)

```
b2 west ; go
```

Hints... A Bot Controller allows you to execute only one single line if you position the cursor on it and select the menu item **do it** or command `d`. If you want to execute a script composed of multiple lines select them first.

For example Figure 1.8 defines some scripts to steer the bot `b2`. Note that the variable `b2` in the script refers to the robot we created and named `b2`.

Some Steering Exercises

Here is a list of exercises you can try to get used to bot programming. Pick the area named `area2` to have more place to play and define the following scripts.

- Define a script that makes a robot walk 5 tiles in its current direction without taking care of bricks.
- Define a script that makes a robot turn east and walk 5 tiles in its current direction.
- Define a script that makes a robot walk a square of 6 tiles.

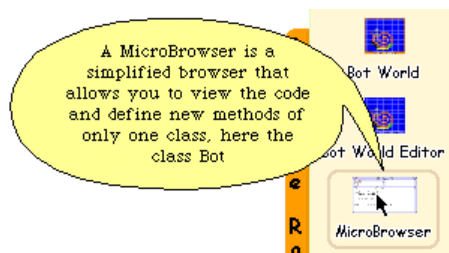


Figure 1.9: Drag and drop the thumbnail of the micro browser to open it.

Defining Methods

In a similar manner than you defined methods for the turtles (see Chapter ??, you can define new methods for the robots using a dedicated code browser. To open such a code browser, drag its thumbnail from the orange flaps as shown in Figure 1.9 or execute the following expression `MicroBrowser browseBot`. You should obtain a micro browser as shown by Figure 1.10. Note that the window title contains the word 'Bot' to indicate you that you will be defining methods for the robots.

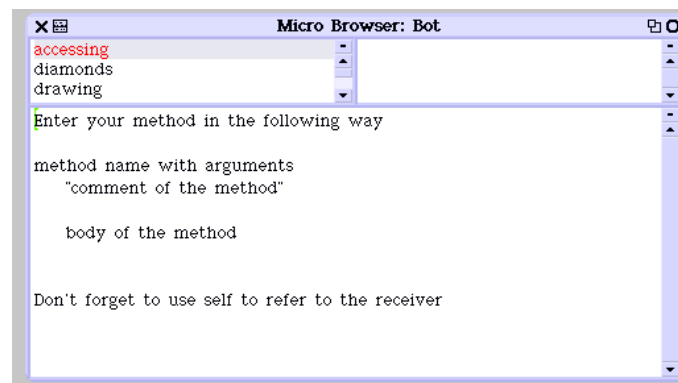


Figure 1.10: A micro browser to define bot behavior.

You can create a category named for example `simple methods` for your methods. Define the method `fiveSteps` that is defined as follows:

Method 1.2

In category simple methods
fiveSteps

5 timesRepeat: [self go]

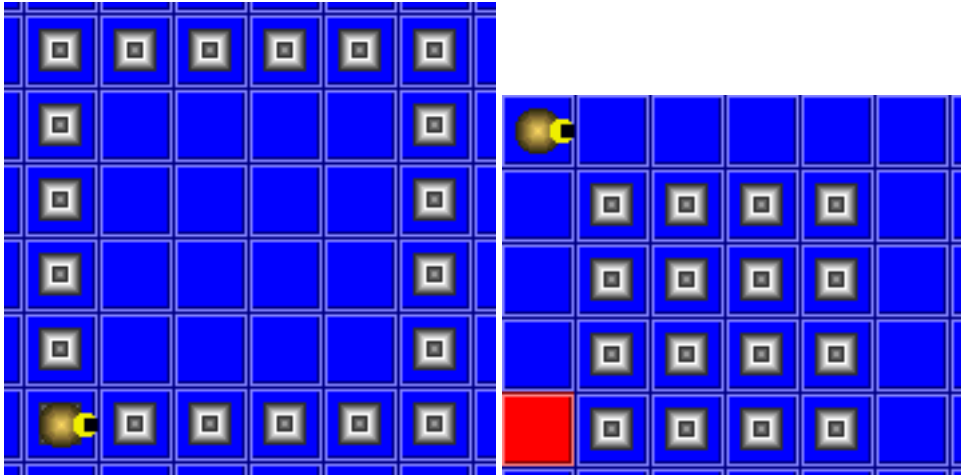


Figure 1.11: Two shapes.

With the assumptions that there is no brick, that the robot is in the middle of the world, and that it has enough diamonds to drop on the floor (Use the method `loadDiams:` to charge it with enough diamonds), define the following:

- Define a script that makes the bot draw the left square of diamonds shown in Figure 1.11.
- Define a script that makes the bot draw the right square of diamonds shown in Figure 1.11.
- Define a script that makes a triangle of diamonds as shown by Figure 1.12.

5 Creating and Editing Bot's Areas

The bot environment allows you to define your own areas using a dedicated editor. To get the editor drag and drop the thumbnail named **Bot World Editor** from the orange flap or execute the following expression `BotWorldBoardEditor newStandAlone openInWorld`. Once executed this script opens a blue window having the bar of buttons shown in Figure 1.13.

The Bot World Editor has from the left to right the buttons.

- **Open Tile Pane.** To open a pane with all the tiles that can be placed on an area.
- **Empty.** To empty the current area.
- **Pick Area.** To select one of the areas already defined.
- **Area Named.** To select one of the areas using its name.

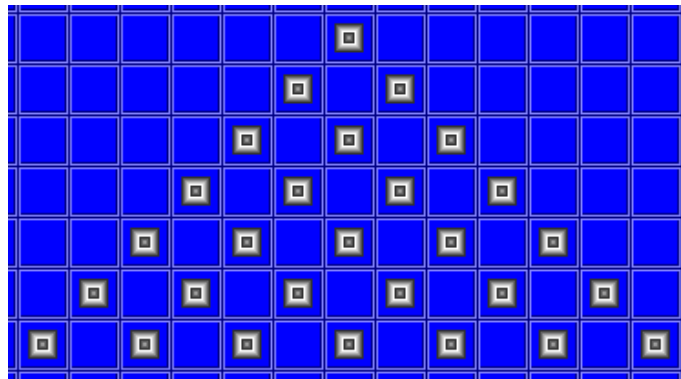


Figure 1.12: A triangle of diamonds.

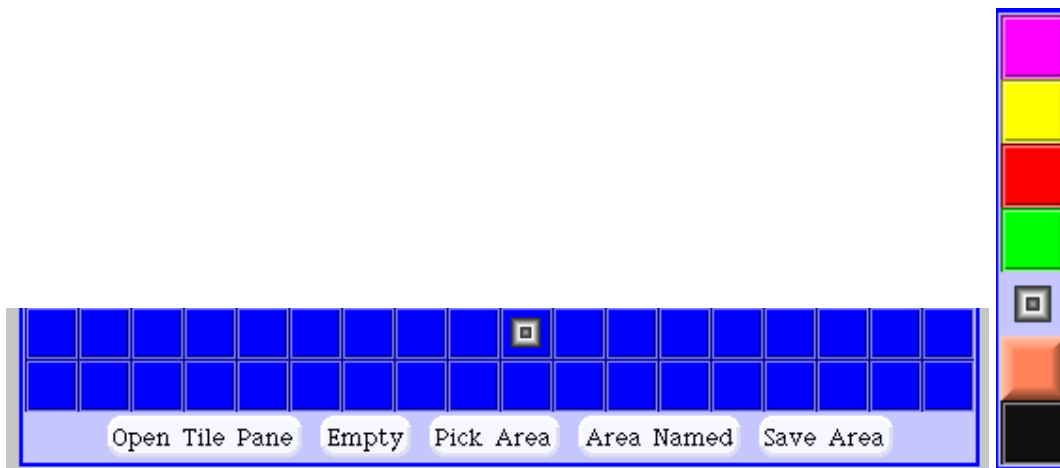


Figure 1.13: The actions proposed by the Bot World Editor and the tile pane.

- **Save Area.** To save the current area.

To add a tile, click on the wished tile in the pane containing the tiles or on a tile already present in the area and drop the tile at the wished location. If you want to delete a tile just pressed shift while clicking on the tile. Now we get ready to try further topics such as how to program our robots.

Summary

You should now be able to open an robot environment, create some robots, steer them using the Bot Controller, and define new methods. The basic set of operations that a robot can execute is: go, turnLeft, turnRight, north, south, east, west, pick, drop, loadDiams: anInteger, diamNumber, paintBlue, paintYellow, paintGreen, paintBlack, and paintMagenta.

In addition to this a robot has several sensors that we will use in the following chapter: canMoveForward, canMoveLeft, canMoveRight isOnBlack, isOnGreen, isOnYellow, isAtHome, isOnMagenta, canPick, and isAtHome.

Fill up the table

| Message | Description | Example |
|-----------------|---|-------------------------|
| $x @ y$ | Creates a point of given coordinates | 300 @ 600 |
| goAt: aPoint | Ask a turtle to move to a given point | caro goAt: 300 @ 600 |
| jumpAt: aPoint | Position a turtle to a given point | caro jumpAt: 300 @ 600 |
| point1 + point2 | Create a point whose components are the sum of the components of two given points | 50 @ 200 + 300 @ 600 |
| point1 negated | Construct a point whose components are the opposite of the original point | (50 @ 200) negated |
| center | Returns the current position of a turtle as a point | barPoint := caro center |

Learning Conditions with Bot

Up until now all the programs we defined were executing *all* the messages they contained one after the other. There was no way to describe that certain messages have to only be executed when certain conditions were true. In this chapter and the following one we will introduce an important programming concept: the notion of *conditional* execution, that is, the fact a certain piece of code is executed when a given condition holds.

We start by defining a simple example that shows the need of conditional execution, in short conditional, then we present in detail the conditional expressions available in Smalltalk.

1 A Simple Example

For this exercise we suggest you to pick the second area as shown in Figure 2.1. This will help you to understand and compare the results of your script with the one we describe.

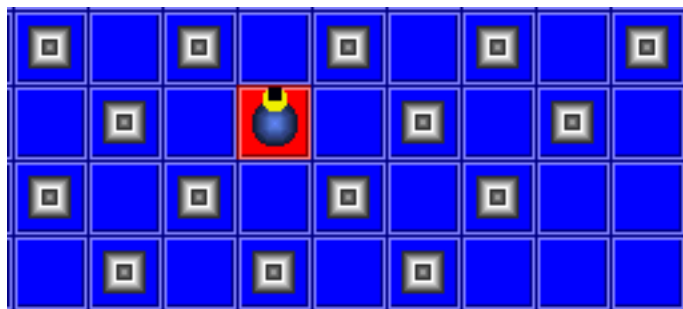


Figure 2.1: Part of the area named area02

A Small Problem. We would like that while a robot is moving it picks a diamond *if* possible, *i.e.*, if there is a diamond on the tile where the robot stands. This problem requires a *conditional* execution, that is when there is a diamond, and only then, the robot should pick it, when there is no diamond it should continue its way and do not try to pick the diamond which would lead to an error as we mentioned in the previous chapter.

The script 2.1 shows a solution to our small problem and Figure 2.2 shows the effect of executing this script 3 times in a row.

Script 2.1 (*Picking diamonds*)

```

b2 east.
b2 canPick
  ifTrue: [b2 pick].
b2 go

```

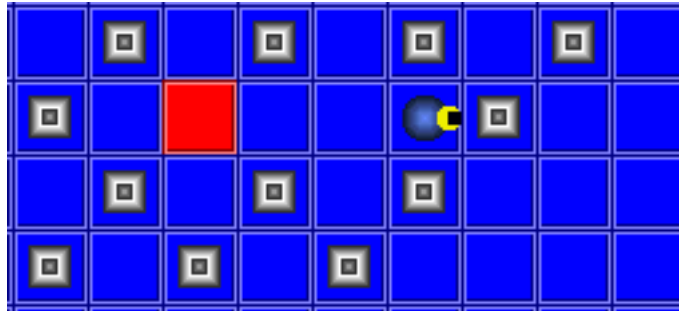


Figure 2.2: Result of applying 3 times the script 2.1

Let us analyze now what happened.

1. We asked the robot to face the east.
2. Then with the expression `b2 canPick ifTrue: [p2 pick]` we asked the robot to check whether it could pick a diamond and *if* this was the case to pick it. This expression is a conditional expression.

A conditional expression is composed of two parts: a *condition* and *conditional messages* as shown by Figure 2.3. The expression `b2 canPick` is a condition and the expression `[p2 pick]` is a *conditional message*. The message `ifTrue:` defines the meaning of the condition, it says that the conditional messages are only executed when the condition is true. Here `b2 pick` will only get executed when the expression `b2 canPick` is true which gets only executed when the condition is true.

3. Finally the expression `b2 go` is executed.

During the first two executions of this script, there was no diamond on the starting place and the one on its right (as shown by Figure 2.1) so the expression `b2 canPick` was false and therefore the conditional expression `b2 pick` was *not* executed (because it is only executed when the condition is true as the `ifTrue:` message indicates it). During the third execution of the script, the bot was on a tile with a diamond, so the condition `b2 canPick` was true and therefore the conditional expression `b2 pick` was executed as the message `ifTrue:` defines it.

What you see is that there are different kinds of expressions: some that are always executed while others are executed only when their associated condition hold. When using the conditional message `ifTrue:` this is when the condition is true that the conditional messages are executed. Note that a conditional expression is not limited to one single message but can be an extremely complex sequence of messages. Similarly the condition can be a complex expression as we will present it in Chapter ??.

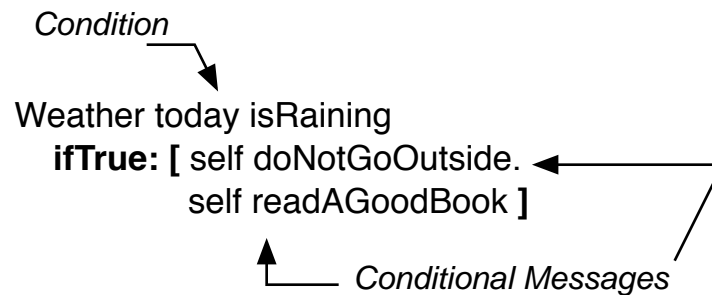


Figure 2.3: A conditional expression composed of a condition and conditional messages.

A conditional expression is composed of a condition and conditional messages.

1.1 Condition and Debugging

If you want to know how many diamonds a robot is carrying just ask it using the expression `b2 diamNumber` and print the result. After executing four times the previous script starting from the robot home it should carry one diamond.

However if you want to understand what your robot is doing, you can write in the Transcript as we shown in Chapter ???. For example modify the script 2.1 to obtain the one shown in script 2.2 and open a Transcript (first thumbnail of the Advanced flap) to follow its execution. Remember that you have to convert number into string using the message `printString` and that you concatenate two strings into one sending the message `,` to the first one.

Script 2.2 (*Picking diamonds*)

```
b2 east.
b2 canPick
  ifTrue: [b2 pick.
    Transcript show: 'Got a new diamond, now I''m carrying ',
      b2 diamNumber printString, ' diamonds'; cr].
b2 go
```

1.2 Two Conditional Messages: `ifTrue:` and `ifFalse:`

In addition to the method `ifTrue:`, Smalltalk offers another method to express conditional execution the method `ifFalse:`. Contrary to `ifTrue:`, the method `ifFalse:` executes its conditional messages when its condition is false. We can always use an `ifFalse:` method instead of a `ifTrue:` method by *negating* the condition. The script 2.3 using `ifTrue:` is equivalent to the script 2.1 using `ifFalse:` because we negated the condition by sending the message `not`.

Script 2.3 (Picking diamonds using *ifFalse:*)

```
b2 east.
b2 canPick not
  ifFalse: [b2 pick].
b2 go
```

The methods `ifFalse:` and `ifFalse:` follow the template shown below. Both execute a condition and depending on the value returned by the condition execute or not the conditional messages.

Important Messages 2.1

aCondition

ifTrue: [*messagesIfConditionIsTrue*]

aCondition

ifFalse: [*messagesIfConditionIsFalse*]

Helper's Hints

When a message is executed the receiver of the message and the arguments are *always* evaluated. There is no exception to this rule. However, for conditional statements or loops, it is necessary to be able to control when a sequence of messages will be executed, if any or the number of times it will be repeated. This is for this purpose that blocks are used delimited by [and]. A block is evaluated, but its semantics is to delay the execution of the messages it encloses. This is the reason why the methods such as `timesRepeat:` and `ifTrue:` require blocks as arguments.

Helper's Hints

2 The Need for `ifTrue:ifFalse:`

Now imagine that we want a robot to turn north when it cannot pick a diamond. The idea is that the robot should walk in zigzag going to the north each time it cannot pick a diamond. The result of such a script is illustrated by Figure 2.4.

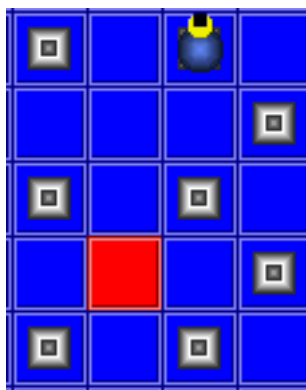


Figure 2.4: Result of applying 3 times the script 2.5 or the script 2.6.

We could write a script as the one shown in script 2.4, however this script does not do what we want as shown by Figure 2.5. Read the script, execute it step by step slowly, and try to understand why the robot is only going straight to the north and not in zigzag as it should do.

Script 2.4 (*Picking diamonds or walking (Wrong Solution)*)

```
b2 east.
b2 canPick
  ifTrue: [b2 pick].
b2 canPick
  ifFalse: [b2 north].
b2 go
```

The idea was to use the `ifTrue:` and `ifFalse:` methods on the same condition thinking that when one will be executed the other won't. But in fact when the first conditional expression is picking a diamond, it is modifying the context of the execution and when the second test is executed it is not the same context as when the first one was executed. Indeed the diamond is not there anymore! In fact when the robot can pick a diamond it does it then when the second condition `b2 canPick` is executed the condition is false. And it will *always* be false, so the robot will always walk in direction of the north.

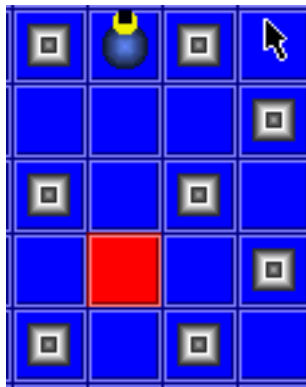


Figure 2.5: Result of applying 3 times the script 2.4.

What we need is to avoid to execute twice the method `canPick` and keep the result of the method for the second conditional expression. We should use the value of the condition when it was *first* executed. This is easy using a variable (see Chapter ??). For example, the script 2.5 introduces a variable named `resOfCanPick` and the conditional expressions use this variable instead of invoking the message `canPick` two times.

Script 2.5 (Picking diamonds or walking (First Correct Solution))

```
| resOfCanPick |
b2 east.
resOfCanPick := b2 canPick.
resOfCanPick
  ifTrue: [b2 pick].
resOfCanPick
  ifFalse: [b2 north].
b2 go
```

The solution is working but it is not really elegant to have to add an extra variable. Imagine if we would have a lot of conditional expressions, we would end up having a lot of extra variables. Smalltalk offers the method `ifTrue:ifFalse:` to solve this problem. The script 2.6 presents the final solution. What we see is that we do not need to use an extra variable and that we do not need to have two conditions. Note that the method `ifTrue:ifFalse:` is a *single* method with two arguments, one for the true case and one for the false case, in a similar way that `color:withSize:` requires two arguments. Therefore you should not put a period after the `]` following the `ifTrue:`.

Script 2.6 (Picking diamonds or walking (Correct Solution using ifTrue:ifFalse:))

```
b2 east.
b2 canPick
  ifTrue: [b2 pick]
  ifFalse: [b2 north].
b2 go
```

The method `ifTrue:ifFalse:` executes one condition, here `b2 canPick`, and depending on its value executes the messages corresponding to the true case or to the false case as explained by Figure 2.6 and the following template. We say that the method `ifTrue:ifFalse:` has two branches corresponding to two different possible executions. Here the branches are limited to a message send (`b2 pick` for the true branch and `b2 north` for the false branch) but a branch can contain a complex sequence of messages as we will see later.

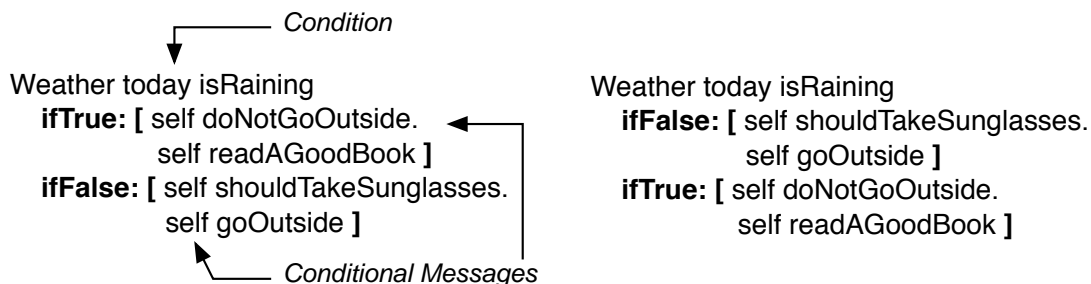


Figure 2.6: Conditional with two branches: one for true and one for false

Note that the method `ifFalse:ifTrue:` also exists and that it works the same way the method `ifTrue:ifFalse:`, that is, it will execute the false case when the condition is false and the true one when the condition is true, exactly as the method `ifTrue:ifFalse:`. This method is just there to help

you writing more readable code if you want to start reading the messages executed when the condition is false as shown by Figure 2.6.

Important Messages 2.2

aCondition

ifTrue: [*messagesIfConditionIsTrue*]
ifFalse: [*messagesIfConditionIsFalse*]

aCondition

ifFalse: [*messagesIfConditionIsFalse*]
ifTrue: [*messagesIfConditionIsTrue*]

Now you are ready to solve a lot of problems based on conditional. The final variation of the small problem with work on will show you that conditions can be nested.

3 Nested Conditions

A conditional expression can contain any other messages and in particular other conditional expressions. This is what we will present now. There is nothing spectacular but it is common that's why we want to show it to you.

Another Small Problem. We would like that a robot picks a diamond *if there is one or* drops a diamond *if there is no diamond and if it has enough diamond to drop.*

We load the robot using the script 2.7 because we can be in a situation where the robot has to drop diamonds without been able to pick some and it should still be able to do it.

Script 2.7 (*Initializing the robot*)

```
b2 east.  
b2 loadWith: 10.
```

Then to solve our problem we write the script 2.8. If you execute four times in a row this script you should obtain the situation shown by Figure 2.7.

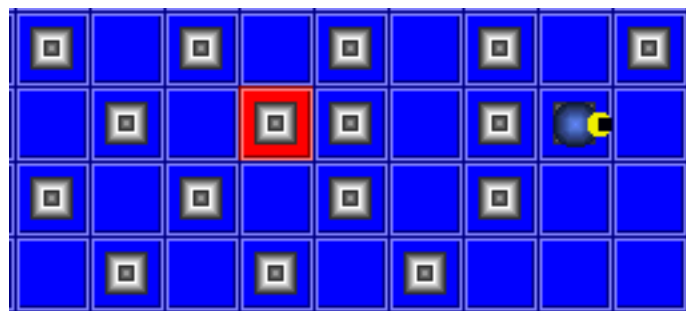


Figure 2.7: Situation once the script pick or drop has been executed four times.

Script 2.8 (Pick or drop)

```

b2 canPick
  ifTrue: [b2 pick]
  ifFalse: [b2 canDrop
    ifTrue: [b2 drop].
    Beeper beep ]
b2 go.

```

As you see this script has the same general structure than the script 2.6. Let us analyze what can happen and how the script works by evaluating different scenario.

1. First we start to test whether the robot can pick a diamond.
2. If this is possible, the robot does it and the conditional expression is over, it then walks on the next tile.
3. If it was not able to pick a diamond then the messages composing the false case are executed. These messages contains a sequence of messages: the first one `b2 canDrop ifTrue: [b2 drop]` is again a conditional expression and the second one is a message to make a little noise (We added the expression `Beeper beep` to show you that you can have multiple condition messages).

The expression `b2 canDrop` is executed

- When it is true the conditional message `[b2 drop]` is executed.
- When the robot cannot drop diamond, the conditional expression is over.

The second message `Beeper beep` is executed independently that the robot can or not drop a diamond.

This example shows that we can nest conditional expressions inside other conditional expressions. It allows one to express more complex situations. Note that when a conditional branch is executed all the expressions that compose it are executed. For example the execution of the message `Smalltalk beep` does not depend on the fact that the robot can or not drop a diamond. The beep will be always produced when the robot cannot pick a diamond because it is contained in a sequence of messages that depend on the `canPick` condition.

4 About Method Returned Values

An important point that we want to stress now is that when we have a conditional expression such as the one shown in script 2.1, the conditional expression `b2 canPick` has to *return* a boolean that is *true* or *false*. This means that we are not only interested by the execution of the message *but also* by the result it computes and returns.

In Smalltalk per default a method always returns the *receiver* of the message, `self`. If we want that the method returns another object such as a boolean, we have to explicitly mention it using the caret `^` construct followed by the value to return.

The method `example1` shown in 2.1 returns the number 1, the method `example2` returns the value of the expression `1 + 2` after the `^`. As the value of the expression `1 + 2` is 3, it returns 3.

Method 2.1

```
example1
  "returns 1"

  ^ 1
```

```
example2
  "returns the result of the expression 1+2"

  ^ 1 + 2
```

Methods are not limited to return only boolean objects. They can return all kind of objects. Look at what we already saw without paying attention. A method can return:

- *numbers*. The method `+` in `10 + 2` returns the sum of the two numbers, `10 max: 20` returns the maximum, `20 atRandom` returns a number between 1 and 20, `caro direction` is returning the current direction to which a turtle is pointing at.
- *booleans* as we just saw.
- *colors* as we were used to do with the turtle. For example the expression `Color blue` is asking the class `Color` to create a new color blue.
- *points*. For example, `b2 botWorldPosition` returns the position of a robot in its environment.
- *string of characters*. For example `b2 name` returns the name of a robot.
- *turtles*. In the expression `Turtle new` the class `Turtle` returns a newly created instance.

The chapter ?? will explain in detail the concept of boolean and boolean expressions. Here we want to stress the point that the *condition* of a conditional expression *must* return a boolean value that is `true` or `false`.

Note that this boolean value can be computed based on other result. For the example, in the following script 2.9 the expression `b2 diamNumber` returns a number, but the expression `b2 diamNumber = 3` returns a boolean, so this is correct. The conditional message, `Beeper beep` will only be executed when the robot named `b2` carries 3 diamonds.

Script 2.9 (A conditional expression comparing numbers.)

```
b2 diamNumber = 3
ifTrue: [Smalltalk beep].
```

5 Final Experiments

- Pay attention to the slightly different problem. Imagine that we want the bot to pick a diamond if there is one *and* dropping one if it can.
- It is quite annoying to get an error when a bot bumps into a wall or in the limits of its world. To fix this problem, we can define the method `sureGO` that checks first whether the move is possible. Define such a method (note that the problem is that we will never know if the robot is indeed bumping into a wall when building more complex behavior).

- In a similar fashion, define the method `safePick` that checks that there is a diamond before picking one.
- Define a method `goBackIfBrick` that makes the robot turning in the opposite direction when it cannot walk anymore.

6 Summary

- A conditional expression is composed of a condition and conditional messages. The conditional messages are executed depending the value of the condition.

| Method | Description |
|---|---|
| <pre>aCondition ifTrue: [messagesIfConditionIsTrue]</pre> | <p>Execute <code>messagesIfConditionIsTrue</code> only if <code>aCondition</code> is true. The robot will only pick the diamond if he can.</p> <pre>b2 canPick ifTrue: [b2 pick]</pre> |
| <pre>aCondition ifFalse: [messagesIfConditionIsFalse]</pre> | <p>Execute <code>messagesIfConditionIsFalse</code> only if <code>aCondition</code> is false. The system beeps only when the robot cannot pick a diamond</p> <pre>b2 canPick ifFalse: [Smalltalk beep]</pre> |
| <pre>aCondition ifTrue: [messagesIfConditionIsTrue] ifFalse: [messagesIfConditionIsFalse]</pre> | <p>Execute <code>messagesIfConditionIsTrue</code> whether <code>aCondition</code> is true otherwise execute <code>messagesIfConditionIsFalse</code>. The robot pick a diamond when it can otherwise the system beeps.</p> <pre>b2 canPick ifTrue: [b2 pick] ifFalse: [Smalltalk beep]</pre> |
| <pre>aCondition ifFalse: [messagesIfConditionIsFalse] ifTrue: [messagesIfConditionIsTrue]</pre> | <p>Execute <code>messagesIfConditionIsTrue</code> whether <code>aCondition</code> is true otherwise execute <code>messagesIfConditionIsFalse</code>. The robot pick a diamond when it can otherwise the system beeps.</p> <pre>b2 canPick ifFalse: [Smalltalk beep] ifTrue: [b2 pick]</pre> |

Conditional Loops

Having conditions is a crucial tool for expressing complex programs. However, conditions are not enough. Sometimes we would like to combine loops and conditions. In fact we would like *conditional loops*, that is loops that repeat sequence of messages while a certain condition holds. Note that recursion that we will briefly present in chapter 6 allows one to express conditional loops without the need of dedicated messages. However, conditional loops not based on recursion is an important concept that deserves an in depth presentation. We will use heavily conditional loops to simulate animal behavior in Chapter 9.4 and other strategies such as escaping mazes or following paths.

1 Conditional Loops

Contrary to simple loops that repeat a sequence of messages a given number of times, conditional loops repeat a sequence of messages while a certain condition holds. Smalltalk defines two messages `whileTrue:` and `whileFalse:` that allows one to define conditional loops as shown by the templates below. In a similar manner than a conditional expression, a conditional loop is composed of a *condition* and *conditional messages* as shown by Figure 3.1.

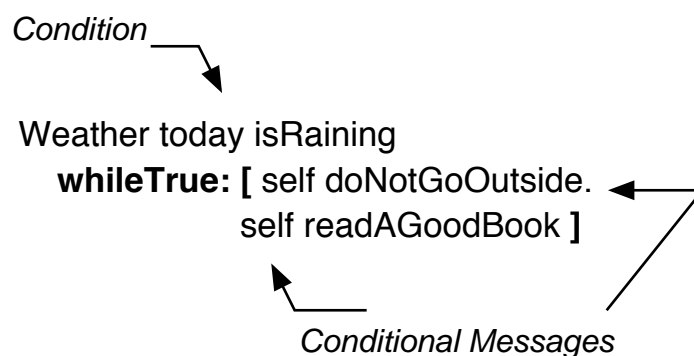


Figure 3.1: The `whileTrue:` conditional loops is composed of a condition and a sequence of conditional messages.

Important Messages 3.1

```
[ condition ] whileFalse:
  [ conditional messages ]
```

```
[ condition ] whileTrue:
  [ conditional messages ]
```

An example. Let us take a simple example to illustrate their use. Imagine that we want that a robot takes all the diamonds available on a line. Let us call this method `pickCompleteLine`. To simplify the example, we first define the method `safePick` (method 3.1) that makes a robot pick a diamond if there is one and do not raise error otherwise.

Method 3.1

```
safePick
  "pick a diamond if there is one"

  self canPick
    ifTrue: [self pick]
```

Now we can define the method `pickCompleteLine` as follows:

Method 3.2

```
pickCompleteLine

  [ self canMoveForward]
    whileTrue: [ self safePick.
                self go]
```

How does it work? First it is important to see that we use the message `whileTrue:` which states that a sequence of messages is repeated while to condition is true. Using the message `whileFalse:` the sequence is repeated while the condition is false.

1. First a condition is executed, here `self canMoveForward`.
2.
 - As the conditional loops message is `whileTrue:`, if the condition is true then the conditional messages are executed. Here, `self safePick. self go` are executed. Then the process restarts as described in the point 1. This is why we have a conditional loop.
 - If the condition is false then the conditional loop stops and nothing more is executed in this method as there is no message following the conditional loop.

In exactly the same way that conditional expression, the condition in a conditional loops must return a boolean as explained in the previous chapter in section 4.

As for `ifTrue:` and `ifFalse:`, `whileTrue:` can be substituted by `whileFalse:` by negating the condition. For example, the method `pickCompleteLine` can easily be defined using the method `whileFalse:` by negating the condition as follow (see method 3.3):

Method 3.3

pickCompleteLine

```
[self canMoveForward not]
  whileFalse: [self safePick.
              self go]
```

As you see the new version is less understandable. Our advice is that you should use the method that helps you to understand the program you are defining.

2 About the Use of []

You may have difficulties to remember when to put squared brackets [] and not. There are basically two rules in Smalltalk. You surround an expression by squared brackets [and] when:

- you need to execute several times the same expression. For example,
 - 4 timesRepeat: [caro go: 10; turnLeft:90] repeats 4 times the messages caro go: 10; turnLeft:90,
 - 1 to: 10 do: [:i | Transcript show: i printString ; cr] repeats ten times the [:i | Transcript show: i printString ; cr] which prints the number to the transcript.
- the expression is not always executed. For example,
 - b2 canPick ifTrue: [b2 pick] only executes b2 pick under certain circumstances,
 - [self canMoveForward] whileTrue: [self safePick; go] repeats multiple times conditionally both self canMoveForward and self safePick; go, therefore the receiver and the argument are blocks.

3 Learning from Errors

As we always learn from errors and that we are always doing errors, looking at errors is an excellent way to understand deeper a topics.

Picking all Diamonds

You may note that the method pickCompleteLine method 3.2 was not really good. Can you understand what is the problem? If you need a small hint can you imagine what's happen for the diamonds that are just in front of a brick? A robot will not pick them. Indeed using the method pickCompleteLine a robot does not pick a diamond that is just near a brick or near the limit of the world. First explain why by showing step by step why this situation occurs. Second propose a solution to this problem.

A possible solution about this problem is shown by the method 3.4. The idea is that a robot should pick a diamond then try to move forward.

Method 3.4

```
pickCompleteLine
```

```
[self safePick.
 self canMoveForward]
 whileTrue: [self go]
```

This example shows that a condition can be composed by a sequence of expressions, here `self safePick` and `self canMoveForward`. The only constraint is that as for all the conditional expressions, the last expression of the sequence should return a boolean (see section 4). Here the method `canMoveForward` returns boolean so this constraint is verified.

Stopping an Endless Loop.

As you may guess or already experienced, it may happens that a conditional loop does not terminate. In fact it is not exceptional to write endless loop even confirmed professional programmers do that. This happens because the boolean expression never returns a value that contradicts the conditional message: true for `whileFalse:` and false for `whileTrue:`.

You can stop a loop by pressing Apple-. on Mac or Alt or Control-C on other platform. Then to understand why the loop does terminate you can use the debugger that pops up and by clicking its Debug button.

What you should understand is that there should be something in the sequences of expressions that leads to change the value of the condition. For example, the message `go` in the method 3.2 linked to the fact that our environment is bounded makes sure that the robot will arrive at one point in time in a situation where the condition `self canMoveForward` will not be true anymore, hence the loop will stop. As you see this is not simple but you should try to always ask yourself whether your loops expressions contain at least expressions that would change the condition value.

To avoid infinite loop, ask yourself if the repeated messages have will eventually invalidate the conditional expression

To help to understand what is going wrong with your loops you can use simple debugging techniques such as introducing a `self halt` expression or the conditional messages in the condition which opens a debugger (see method 3.5).

Method 3.5

```
pickCompleteLine
```

```
[self halt.
 self safePick.
 self canMoveForward]
 whileTrue:
 [self go]
```

Another approach is to generate a trace in the Transcript as explained in Chapter ?? as shown in Figure 3.6. For example in the following method we write in the transcript the position of the robot.

Note that the position are points and that we transformed them into strings because the transcript only knows to print strings.

Method 3.6

```
pickCompleteLine
```

```
[self safePick.
 self canMoveForward]
 whileTrue:
   [Transcript show: self botWorldPosition printString ; cr
    self go]
```

4 Practicing

Now we are ready to write some other simple conditional loops. Keeping in mind that a robot can drop multiple diamond on the same tile, define the following methods:

- `fullyDrop` that makes a robot dropping all the diamonds it is carrying on the current tile.
- `fullyPick` that makes a robot picking all the diamonds on the current tile.
- `dropLineOfFive` that makes a robot dropping all its diamond per packets of five in a line.
- `fullPickLine` that makes the robot picking all the diamonds available on each tiles for a complete line.
- `reverseLine` that makes the robot picking a diamond if there is one or dropping one if there is none and this on a complete line.
- `straightUntilYellow` that makes the robot moving straight and stopping if it is passing over a yellow tile.

Note that you may have to nest conditional and conditional loops.

5 Experiments: Finding a Yellow Spot

Now we would like to build different strategies to let a robot walk until it finds a yellow tile.

Preparation

First define an environment that looks like the one shown by Figure 3.2 to have a bit more fun.

We need some methods to manage the direction of our robot. For example, we can define the method `pointBack` which makes a robot pointing in the opposite direction, `pointLeftOrRight` which makes it points randomly on the left or the right. Imagine how such methods could be implemented to train yourself before reading our definitions.

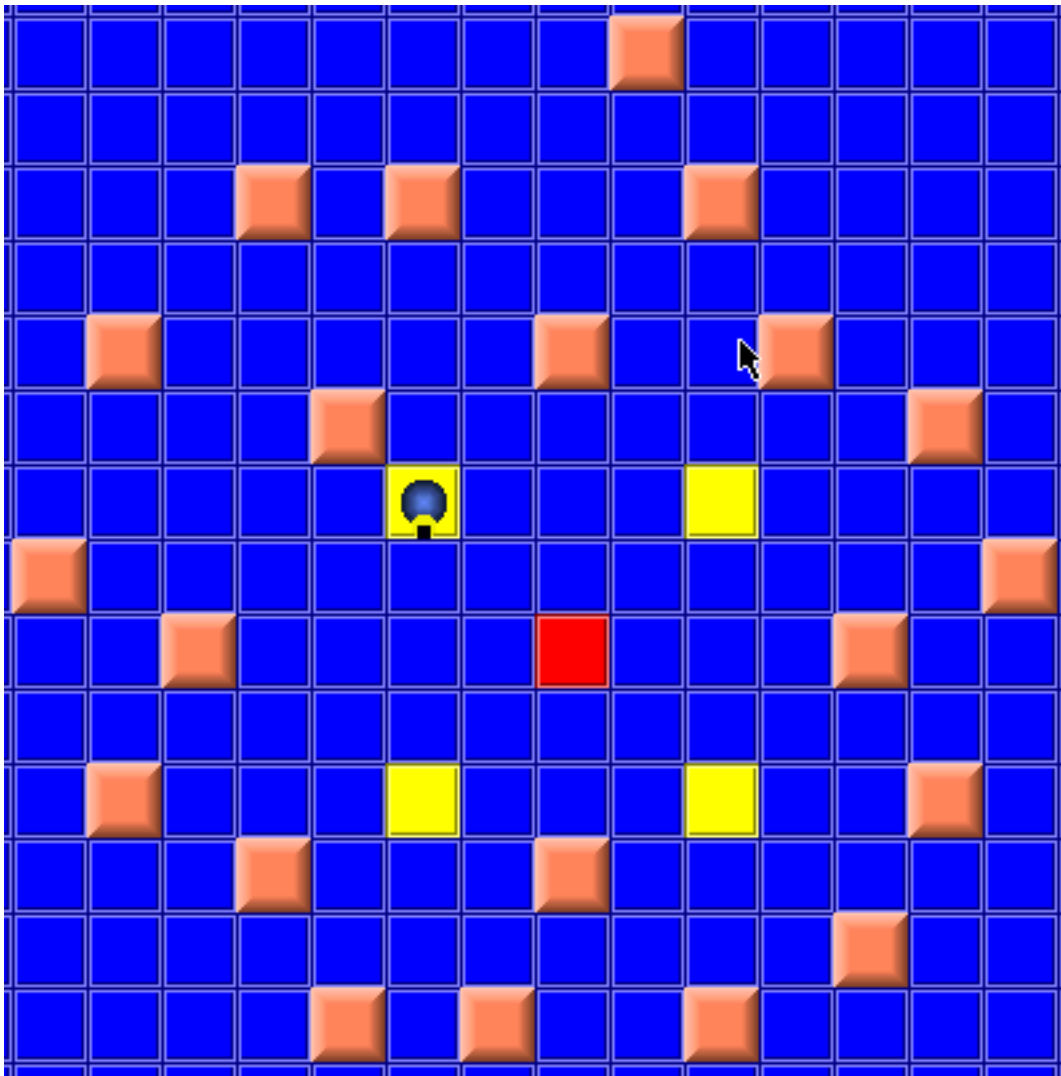


Figure 3.2: An environment for fooling around looking for yellow tiles.

Method 3.7

pointBack

"Make the receiver pointing in the opposite direction"

self direction: self direction negated

Method 3.8

```

pointLeftOrRight
  "Make the receiver pointing randomly to its left or right direction"
  2 atRandom = 2
  ifTrue: [self turnLeft]
  ifFalse: [self turnRight]

```

The method `pointRandomly` can be implemented in different ways. We propose you to define the method `randomDirection` that will return a random direction. This implementation is explained in Chapter 6. Note that a direction is a point composed by -1,0, or 1 with the constraint that $x * y$ should always be = to 0 as the receiver should always point in one of the four directions. This method is provided by default.

Method 3.9

```

pointRandomly
  "Make the receiver pointing in a random direction"

  self direction: self randomDirection

randomDirection
  "a direction is a point composed by -1,0, or 1 with the
  constraint that  $x * y$  should always be = to 0"

  | x y |
  x := 3 atRandom - 2.
  y := 3 atRandom - 2.
  ^ (x * y) isZero
    ifTrue: [x @ y]
    ifFalse: [self randomDirection]

```

Strategies

Imagine different strategies for a robot to find a yellow tile.

Bumping and Turning. A simple strategy is to make a robot walking until it is on top of a yellow and when the robot bumps into a wall make it turns randomly on the right or the left.

Method 3.10

```

rightOrLeft

[self isOnYellow]
  whileFalse: [self canMoveForward
    ifFalse: [self leftOrRightDirection].
  self go]

```

Note that there is problem with the method `rightOrLeft`. Can you see it in Figure 3.3? What's happen if there is just a brick close to another one and that the robot choose to go exactly in that direction. Propose a solution.



Figure 3.3: Brick configuration potentially blocking a robot using `leftOrRightDirection` strategy.

Fooling around. Another strategy is to let the robot randomly browse and check whether it is walking on a yellow tiles.

Method 3.11

browse

```
[self isOnYellow]
whileFalse:
    [self pointRandomly.
     self canMoveForward
     ifTrue: [self go]]
```

A fun extension is to make the robot painting the tiles were it passed over. This way you will be able to see where it moved. You can imagine a solution where you use simply green (left in Figure 3.4). Then a variation is to use multiple colors that represent the number of times the bot passed on the same tile. For example in the right picture of Figure 3.4 a robot painted first in green, then magenta when it was on a green tile and finally black when it was on a magenta tile. We suggest you to define a method `paintLevel` (see 3.12) that you call from the method `browse`.

Method 3.12

paintLevel

```
self isOnBlue
ifTrue: [self paintGreen]
ifFalse: [self isOnGreen
ifTrue: [self paintMagenta]
ifFalse: [self isOnMagenta
ifTrue: [self paintBlack]]]
```

6 Summary

- To avoid infinite loop, ask yourself if the repeated messages have will eventually invalidate the conditional expression.

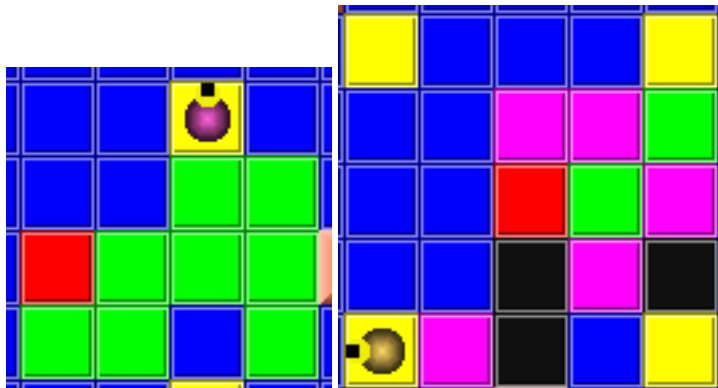


Figure 3.4: Left: A robot browsing and letting a simple trace. Left: A robot browsing and letting a complex trace.

- The last message of a condition should return a boolean

| Method | Description |
|--|--|
| <code>[aCondition] whileFalse: [SequenceOfMessages]</code> | Execute SequenceOfMessages only if <i>aCondition</i> is false. |
| <code>[aCondition] whileTrue: [SequenceOfMessages]</code> | Execute SequenceOfMessages only if <i>aCondition</i> is true. |

Part II

Advanced Concepts and Fun Projects

Paths and Mazes

In this chapter we propose you to define programs so that a robot follows automatically a simple path. You will practice conditional loop and conditional. The next problem we will study is how to program a robot so that it can escape a maze.

1 Defining a Path

The first thing we have to agree upon is the definition of a path. We define a path as connected yellow tiles starting from the red tile and ending with a black tile. The bot should stop on the black tile. We also put the extra constraint that a path should not have branches and there should be only one way to reach the black tile. In a following chapter we will propose an algorithm that makes a bot finding the black tile even in presence of branches and dead ends. Figure 4.1 shows an example of a simple path.

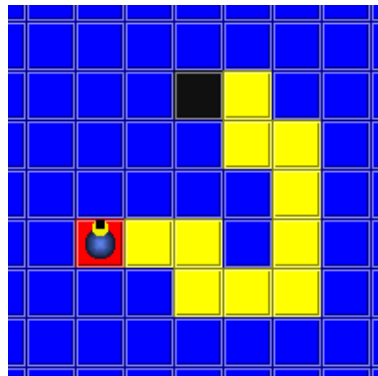


Figure 4.1: A path: yellow tiles starting from the red tile until a black tile without branch.

2 Defining a Strategy

We have to define a strategy for our bot. Such a strategy depends on the sensor capabilities. Right now a bot can only detect whether a tile is of a given color when it is standing on that tile. Therefore the simplest idea is to move the bot in a direction, ask it to check whether the tile is yellow, ask it to come back to the previous position when the tile was not yellow, then to check another direction and so on.

We have to choose an order between the four positions a bot can move. We chose to test first the tile forward, then the left, right, and finally the rear position. Figure 4.2 right shows the steps.

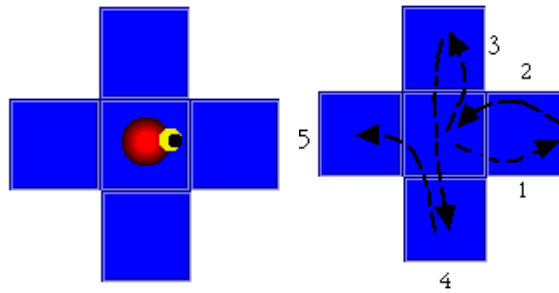


Figure 4.2: A simple strategy checking forward (1), then left (2 and 3), right (4), and rear (5).

To ease the definition of the method `followYellowPath` which will define the algorithm to follow a path, we propose you to define the following helper methods:

- `goBack` that places the receiver one tile rear conserving its original direction (see 4.1),
- `goLeft` and `goRight` that move respectively the receiver one tile left or right from the current position (see 4.2 and 4.4), and
- `goOpposite` that moves the bot from the place marked 3 on Figure 4.2 to the place marked 4 (see 4.3).

Again it is much simpler to compose a complex method based on methods that hide certain details.

Method 4.1

`goBack`

"Move the receiver one tile rear conserving the original direction"

```
self turnLeft; turnLeft; go; turnLeft; turnLeft
```

Method 4.2

`goLeft`

```
self turnLeft; go
```

Method 4.3

`goOpposite`

```
self turnLeft; turnLeft; go; go
```

Method 4.4**goRight**

```
self turnRight; go
```

Finding the Next Yellow Tile

Once these simple methods defined we can define the method `nextPlace` which looks for the next yellow tile and ask the robot to move there (see 4.5). This is this method that defines the order to to check the neighboring tiles.

Method 4.5**nextPlace**

"Place the receiver on the next yellow tile starting from the front,
left, right and rear place"

```
self go.
self isOnYellow
  ifTrue: [^ self]
  ifFalse: [self goBack; goLeft.
    self isOnYellow
      ifTrue: [^ self]
      ifFalse: [self goOpposite.
        self isOnYellow
          ifTrue: [^ self]
          ifFalse: [self goBack; goRight.
            self isOnYellow
              ifTrue: [^ self]
              ifFalse: [self error: 'no yellow around']]]]]]
```

Note that this method uses the fact that the method execution terminates when a return statement is encountered (^). The method `meth:returnTwo` shows this behavior. The expression `0 isZero` is always true, therefore the expression `^` is executed. The method returns 2. No other messages is then executed. The last message `Beeper beep` outside of the conditional is not executed. The method `nextPlace` uses this behavior to stop its execution as soon as the robot is on a yellow tile.

Method 4.6

```
Bot>>returnTwo
(0 isZero)
  ifTrue: [^ 2]
  ifFalse: [^ 3].
Beeper beep
```

Now we define the method `followYellowPath` that moves the bot while it is not on the black tile (see 4.7).

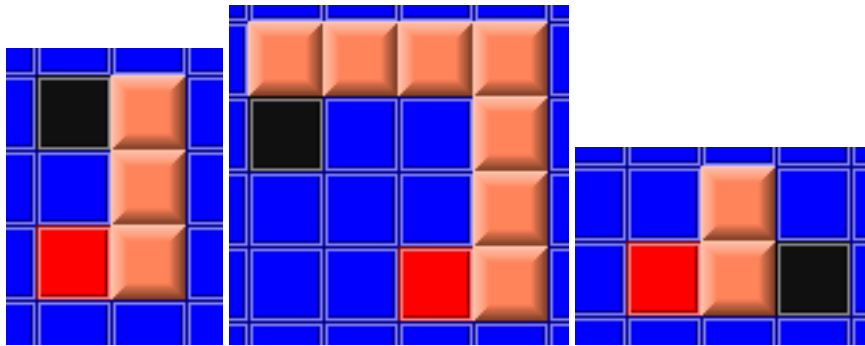


Figure 4.4: Some basic but already interesting mission.

3 Escaping a Maze: Following a Wall

Different strategies exist to escape mazes. We will develop one of them that consists of following a wall until the exit. We will present step by step how to develop an algorithm so that a robot follows a wall located on its right. We decide that the bot has finished his mission when it arrive on a black tile. Figure 4.4 shows some extremely simple missions. We assume that the bot starting place should have a brick on the right.

We need two sensors `canMoveRight`, `canMoveForward` in addition to the sensor checking the color `isOnBlack`. The simplest version of an algorithm that could steer the robot to follow walls is shown by the method 4.9. This method makes the bot moving forward if it is not on a black tile and if there is a no brick in front of the bot but one on its right.

Method 4.9

In category wall follower
`followRightWallOne`

```
| wallOnRight forward |
[self isOnBlack]
  whileFalse: [wallOnRight := self canMoveRight not.
               forward := self canMoveForward.
               wallOnRight & forward
               ifTrue: [self go]]
```

Reproduce the areas described by Figure 4.4 and try this first method. Note that you can open several worlds. The first version of our algorithm works for the first straight mission but should loop endlessly for the other ones. You can stop the loop by pressing `Apple-` on Mac or `Alt.` on windows or Linux. This will open a debugger that you can simply close. If you want to understand how the method works or debug it we suggest you to insert the expression `self halt` before the expression `wallOnRight & forward`. This way you can see the process of the algorithm.

You have now all the pieces to define your own solution. Try to define one, use the different situations shown in the following Figures 4.6, 4.7 to see if your approach really works. The following shows our own solution, step by step but this is not necessary the best one.

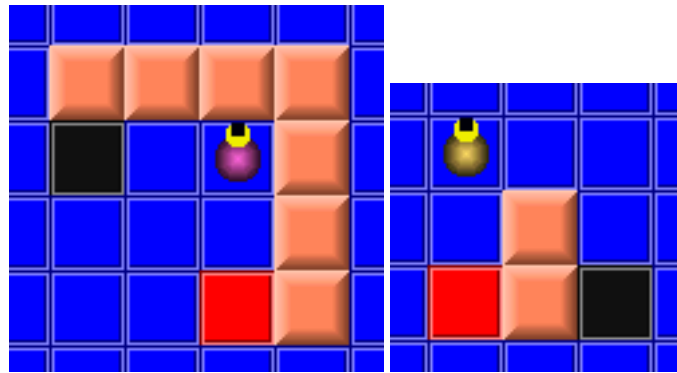


Figure 4.5: Limits of the first algorithm.

A Possible Solution

By analyzing the middle mission of Figure 4.4 we see that the bot should do something when it is facing a wall. As the expression `wallOnRight & forward` can be false when the bot is facing a wall or when there is no wall on its right, we have to test first before doing something.

In the method 4.10 we added the expression `forward ifFalse: [self turnLeft]` making the bot turns left when it was facing a wall. Indeed to test if the bot is facing a wall, we simply test in the second branch that it can move forward. We can do that because in the first `ifFalse:` branch, we know that the expression `wallOnRight & forward` is false which means that the bot cannot move forward, does not have a wall on its right or both at the same time.

Method 4.10

followRightWallTwoIn category wall follower

```
| wallOnRight forward |
[self isOnBlack]
whileFalse: [wallOnRight := self canMoveRight not.
  forward := self canMoveForward.
  wallOnRight & forward
  ifTrue: [self go]
  ifFalse: [forward
    ifFalse: [self turnLeft]]]
```

This method works well for the left and middle missions but failed for the right one. Because it still does nothing when there is no wall on its right. To fix this problem we introduce a true branch to the previous test as shown in the method 4.11. Note that the bot does not only turn right but also moves. Understand why this is not sufficient to only turn right.

Method 4.11

followRightWallThree *In category wall follower*

```

| wallOnRight forward |
[self isOnBlack]
whileFalse: [wallOnRight := self canMoveRight not.
  forward := self canMoveForward.
  wallOnRight & forward
  ifTrue: [self go]
  ifFalse: [forward
    ifTrue: [self turnRight ; go]
    ifFalse: [self turnLeft]]]

```

Experiment with the method 4.11 various missions such as the ones shown in Figure 4.6. However the method 4.11 is not perfect. Indeed, a bot is not able to finish the missions presented by Figure 4.7.

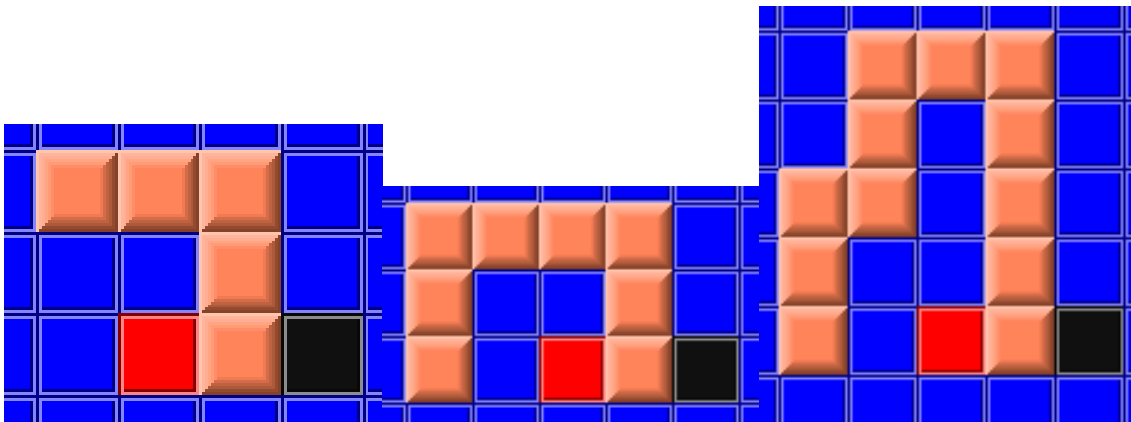


Figure 4.6: Some other missions that the method 4.11 easily handle.

Insert a breakpoint (`self halt`) in the method 4.11 to understand what is happening. Press the proceed button of the debugger to let the method runs until the next time the halt expression will be executed (as shown by Figure 4.8).

The problem is that when there is no wall on the right but a wall in front of the bot, it does not turn right. The method 4.12 fixes this problem.

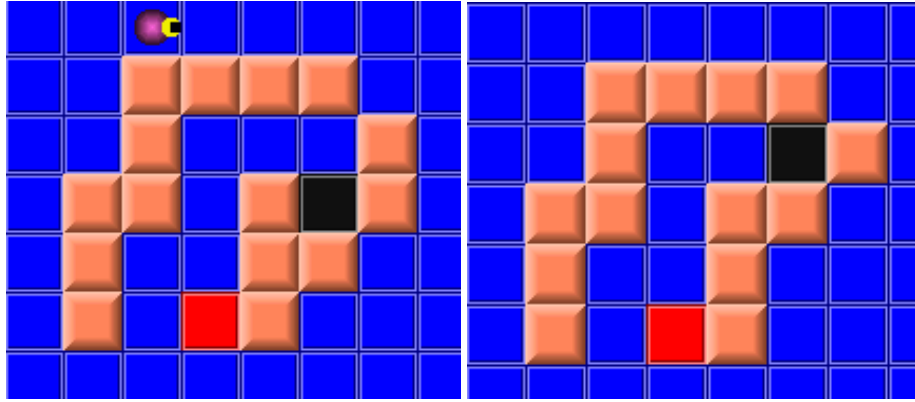


Figure 4.7: Missions that are not handled well by the method 4.11.

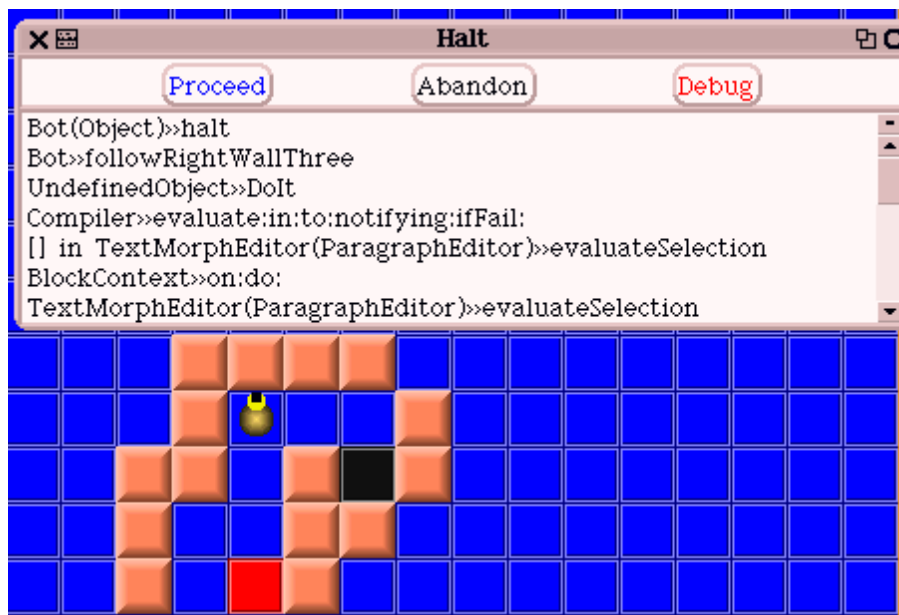


Figure 4.8: Identifying the problems of the method 4.11 using a debugger.

Method 4.12

In category follower

followRightWallFour

```

| wallOnRight forward |
[self isOnBlack]
whileFalse: [wallOnRight := self canMoveRight not.
forward := self canMoveForward.
wallOnRight & forward
ifTrue: [self go]
ifFalse: [forward
ifTrue: ["no brick right and can forward"
self turnRight; go]
ifFalse: [wallOnRight
ifTrue: ["brick right and in front"
self turnLeft]
ifFalse: ["no brick right but brick in front"
self turnRight; go]]]]]

```

The method 4.12 is not the only solution. Another solution is given by the method 4.13. We let you understand and test them on the final mission illustrated in 4.9.

Method 4.13

In category follower

followRightWallFive

```

| wallOnRight forward |
[self isOnBlack]
whileFalse: [wallOnRight := self canMoveRight not.
forward := self canMoveForward.
wallOnRight & forward
ifTrue: [self go]
ifFalse: [wallOnRight
ifTrue: [forward
ifFalse: ["brick right and in front"
self turnLeft]]
ifFalse: [self turnRight; go]]]

```

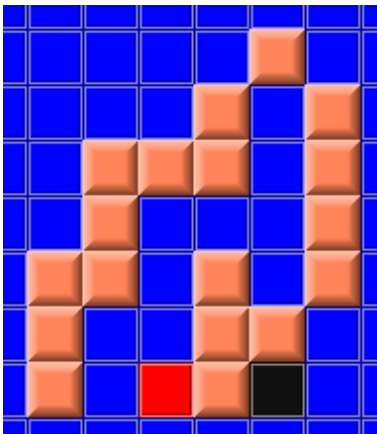


Figure 4.9: A final mission for our super bot.

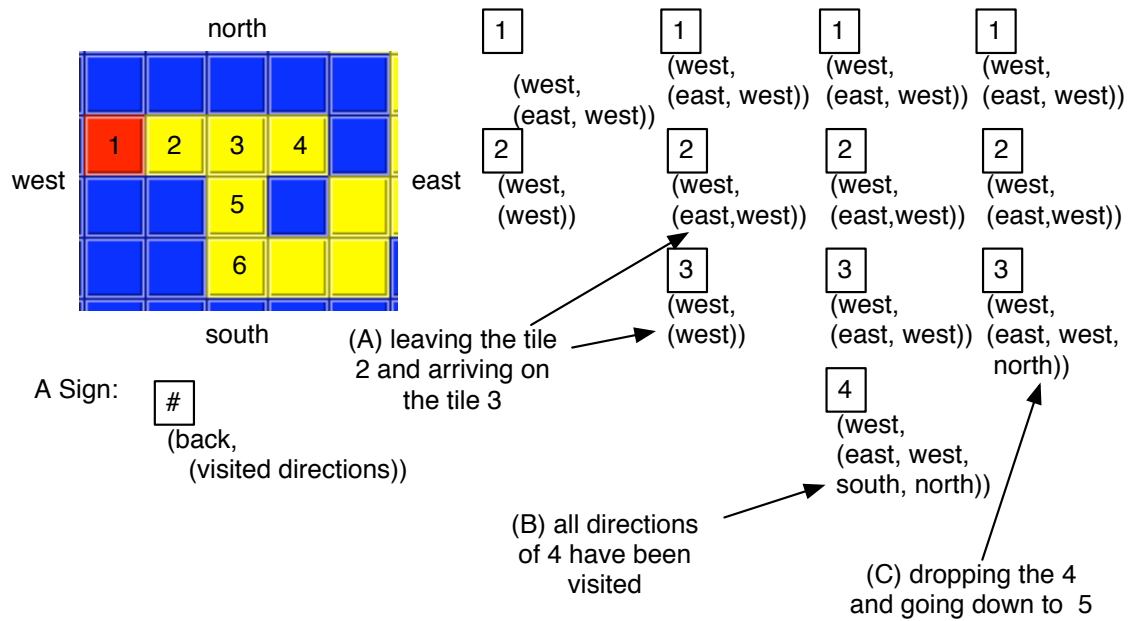


Figure 5.2: Keeping enough information to come back on our steps. Three situations (A) leaving the tile 2 and arriving on 3, (B) all the directions of 4 have been visited, and (C) dropping 4 and going to 5.

Figure 5.2 shows the information a robot keeps. A sign that is referring to a tile is composed of a *back* direction and a set of *already visited* directions. Each time the robot explores the surrounding tiles it adds information to the current tile. Figure 5.2 shows how a list of signs evolves while the robot is exploring and following the path. Let's look at some key situations.

- The leftmost column represents the sign situation when the robot is about to go on the tile marked 3. For the tile 2 we have the sign (west, (west)) which indicates that the robot came from the west and that for the tile 2 it already visited its west which is obvious as it is coming from the west.
- The second column from the left shows the situation when the robot left the tile 2 and arrives on the tile 3. First a new sign is added to the list. This sign (west, (west)) indicates that the robot came from the west. Then the sign relative to the second tile is updated. As the tile 3 is to the east of the tile 2, the sign now mentions that the east direction for the tile 2 has been also visited.
- The situation depicted by the third column is when the robot arrived on the tile marked 4 and after it finished to explore the surrounding directions. The sign (west, (east, west, north, south)) reflects this fact. The tile 4 is a dead-end, all the directions have been tried so the robot should follow the back direction of the sign, here west and try another direction except the east.
- The rightmost column shows the situation after the robot came back from the tile marked 4 and tried other directions. Here, the sign show that it tried to go north. Now the only direction left is the south.

3 Implementing the Solution: Managing Signs

First we need a way to represent a sign and associate a sign to a tile. We represent a sign by an array with two elements: a direction and a set of directions¹. We do not associate a sign to each tile but a robot keeps a list of the signs with the constraint that the sign at the end of the list is the one associated with the tile where the robot is.

The method 5.1 declare two variables `signList` and `allPossibleDirections`. The variable `signList` will be used to represent the list of signs. It is initialized with an `OrderedCollection`, that is a collection which maintains the order in which elements are added. The variable `allPossibleDirections` represents all the directions that a robot can follow. It is initialized with an ordered collection containing all the directions. Note that we could have defined a new method but it would have created a collection at each execution².

Method 5.1

In category path finder initialize

```
Boot>>initializeRobot
  "Initialize the sign list and some other variables"
```

```
| dirs |
  self set: #signList to: OrderedCollection new.
  dirs := OrderedCollection new.
  dirs add: self eastEncoding.
  dirs add: self westEncoding.
  dirs add: self northEncoding.
  dirs add: self southEncoding.
  self set: #allPossibleDirections to: dirs
```

We define two methods `allPossibleDirections` and `signList` to access the robot variables.

Method 5.2

```
Boot>>allPossibleDirections

  ^ self valueOf: #allPossibleDirections
```

Method 5.3

```
Bot>>signList

  ^ self valueOf: #signList
```

As we mentioned earlier, we define the sign for the current tile as the last one of the sign list. We define the method `currentSign` to return this sign as shown in method 5.4.

¹In object-oriented programming using array this way is a sign that we should have created a new class to represent signs. However in the context of this book classes will not be presented.

²Again using object-oriented programming we could have solved this problem easily but this is not the topic of this book.

Method 5.4

```
Bot>>currentSign
  "returns the current sign"

  ^ self signList last
```

We define the method `newSignGoingBack` which returns a new sign with the information to move back in the direction from which the robot is currently coming. So if a robot is heading at the east, the sign will be `(west, (west))` to represent that to move back it should go west and that the west tile has already been visited.

Method 5.5

```
Bot>>newSignGoingBack
  "a sign is a structure whose first element tells from where the
  bot comes from and second element tells the directions that
  have been tested. This method creates a new sign to come back
  to the current position"

  | oppositeDirection |
  oppositeDirection := self oppositeDirection.
  ^ Array
    with: oppositeDirection
    with: (Set new add: oppositeDirection;
          yourself)
```

Method 5.6

```
Bot>>oppositeDirection

  ^ self direction negated
```

Now we can define the method `addNewSignForCurrentPlace` that adds a new sign to the sign list. This method simply adds at the end of the sign list a new one, as we always consider that the signs representing the current place are the last ones of the list of sign we keep.

Method 5.7

```
Bot>>addNewSignForCurrentPlace
  "add to the signList a new sign from the current place"

  self signList addLast: self newSignGoingBack
```

The method `findAndPointFirstYellowDirection` 5.8 points a robot in a direction of a yellow tile around its current position and returns the direction in which the robot is pointing to. Invoking this method ensures that the robot will start its journey in the right direction. This method does not work if there is no yellow tile around.

Method 5.8

```

Bot>>findAndPointFirstYellowDirection
"Return the first direction having a yellow tile from the current
tile. Does not work if there is no yellow around"

| allDirections currentDirection |
allDirections := self allPossibleDirections copy.
[currentDirection := allDirections removeFirst.
self direction: currentDirection.
self isTileInFrontYellow] whileFalse.
^ currentDirection

```

The Strategy

Now we are ready to implement the strategy that a robot will follow. The method `find` shown below (5.9) initializes the robot, picks up a first direction (see method 5.8), adds the sign corresponding to the starting tile, and invokes the main loop of the algorithm via the method `nextMove`.

Method 5.9

```

Bot>>find

self initializeRobot.
self findAndPointFirstYellowDirection.
self addNewSignForCurrentPlace.
[self isOnBlack]
  whileFalse: [self nextMove]

```

The method `nextMove` is the main method of the algorithm: the robot first picks a new direction, then moves on the new tile, then it checks whether this is a dead-end that is no surrounding yellow tiles other than the one it comes from. In such a case the robot steps back following the signs and drops the sign related to the dead-end from the list of signs.

Method 5.10

```

Bot>>nextMove

| dir |
dir := self pickAndMarkDirectionOnCurrent.
dir isNil
  ifFalse: [self moveToNewDirection: dir].

```

Now we will look at each of the methods invoked during the execution of `nextMove`.

The method `pickAndMarkDirectionOnCurrent` is responsible for determining where a robot should go. First the method `pickNonVisitedDirection` (see 5.13) is used to find a direction that does not have been previously visited. When it returns `nil`, it means that there are no other directions to try, the robot steps back on the previous tile using the expression `goBackFollowingSign: self dropAndReturn-`

CurrentSign. Then when the tile in the new direction is neither yellow nor black, a sign is added to the signs of the current tile to mention that the direction has been tested. This is done until a yellow or black tile is found.

Method 5.11

```

Bot>>pickAndMarkDirectionOnCurrent
  "Pick a direction, mark the current path. Return nil if this a dead end."

  | dir res |
  [dir := self pickNonVisitedDirection.
  dir isNil
  ifTrue: [self goBackFollowingSign: self dropAndReturnCurrentSign.
  ^ nil].
  res := (self isTileBlackInDirection: dir) | (self isTileYellowInDirection: dir).
  res ifFalse: [self markTestedDirection: dir].
  res] whileFalse.
  ^ dir

```

With the method `moveToNewDirection:` (see 5.12), the robot points in the new direction, adds to the current signs the fact that he will visit the new direction, creates a new sign for the new direction and moves there.

Method 5.12

```

Bot>>moveToNewDirection: dir
  "define a new sign for the new tile, mark that it will be
  visited and move there"

  self direction: dir.
  self markTestedDirection: self direction.
  self addNewSignForCurrentPlace.
  self go

```

Getting a new direction for a given place is simple, it suffices to subtract the set of the already visited places to the one of all the possible directions. If the result is empty then it means that all the directions have been visited and we return nil in that case.

Method 5.13

```

Bot>>pickNonVisitedDirection
  "Return the direction of a tile surrounding the current one
  that does not have been already visited. Return nil if all the
  tiles have been visited."

  | possible visitedDir |
  visitedDir := self currentSign at: 2.
  possible := self allPossibleDirections difference: visitedDir.
  ^ possible isEmpty
  ifFalse: [possible first]
  ifTrue: [nil]

```

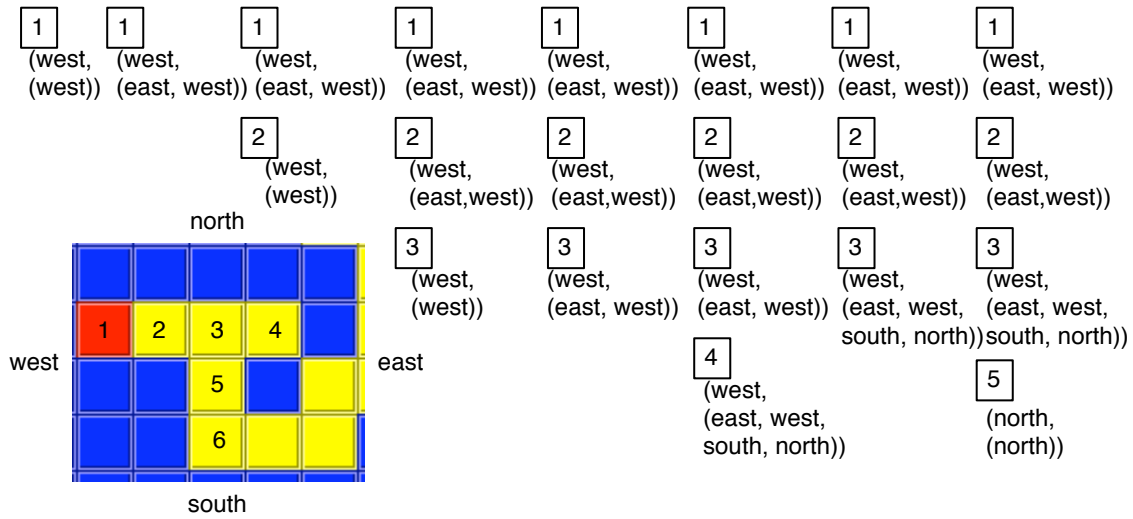


Figure 5.3: Steps by steps keeping and improving signs.

Note that the expression `ifTrue: [nil]` may not be shown when using the pretty printing as when the expression `ifFalse:` executes a true condition it returns `nil`.

The method `markTestedDirection:` is a key method as it adds that a given direction has been visited to the last element of the sign list. A sign is a structure whose first element tells where the bot should step back and second element tells the direction tested. Adding a tested direction implies to add it in the second element of the current sign.

Method 5.14

```
Bot>>markTestedDirection: aDirection
"Add the given direction to the list of visited directions for
the current sign"

(self signList last at: 2) add: aDirection.
```

Figure 5.3 illustrates in detail the process of adding signs to the current tile and to the next one.

Some Utilities Methods

We still need to define some simple methods. When all the directions have been visited and the tile is not black the robot need to step back on the tile it comes from and also drop from its list of sign the one representing the dead-end. The method `dropAndReturnCurrentSign` removes the last sign of the list of signs.

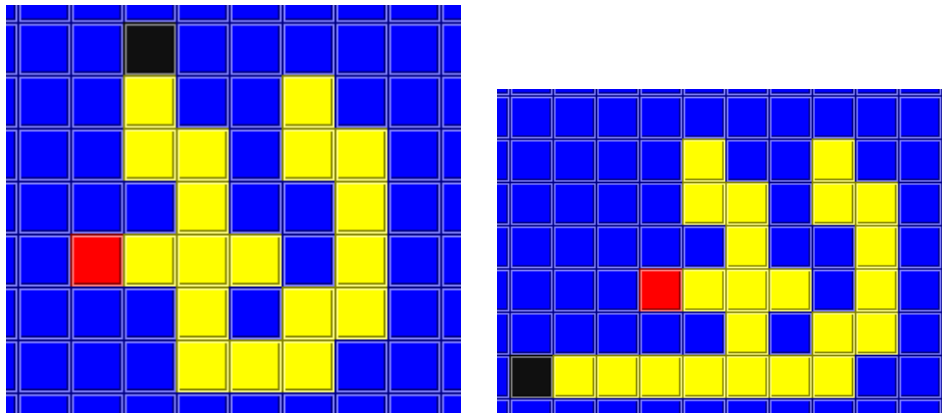


Figure 5.4: Some other interesting paths.

Method 5.15

```

Bot>>dropAndReturnCurrentSign
    "remove the last sign from the list of visited tiles
    and return it"

self signList isEmpty
    if True: [self error: 'should backtrack but cannot'].
^ self signList signList removeLast

```

To step back to the tile it comes from the robot uses the information written in the sign, in particular the information mentioning where it should move back. With the method `goBackFollowingSign:`, a robot steps back to the previous tile.

Method 5.16

```

Bot>>goBackFollowingSign: aSign

self direction: (aSign at: 1).
self go.
self turnLeft; turnLeft.

```

Finally you need to define some advanced sensors as follow and explain in chapter ??.

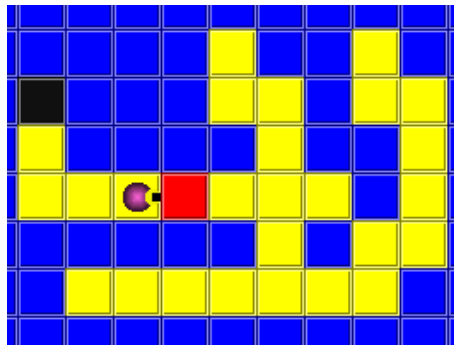


Figure 5.5: When the robot is forced to pass over the starting tile our implementation break.

Method 5.17

In category advanced sensors

Bot>>isTileBlackInDirection: aDirection

```
| tileInFront |
tileInFront := aDirection + self botWorldPosition.
^ (self isNextMoveInWorld: tileInFront)
and: [(botWorld tileAt: tileInFront) isBlackTile]
```

Method 5.18

In category advanced sensors

isTileYellowInDirection: aDirection

```
| tileInFront |
tileInFront := aDirection + self botWorldPosition.
^ (self isNextMoveInWorld: tileInFront)
and: [(botWorld tileAt: tileInFront)
isYellowTile]
```

4 Some Work Left

The strategy we implemented works well except in one case: when the robot has to cross its starting point. For example Figure 5.5 breaks our implementation. This is due to the way we initialize the list of signs, in fact for the starting point we said that we were going from the left while in fact we should have distinguish that the first sign was a bit special. The problem happens because when the robot passes over the starting place it makes and drops it from the list (because we said that the robot already visited the tile which is false) while in fact it should not. Propose a solution to this problem.

To help you find the solution we can tell you that you need to have a method that add a sign for the starting place and that this method should be invoked instead of the method `addNewSignForCurrent` in the method `find`. Our solution is one line so we are sure that you will find it.

A Quick Look at Recursion

*Recursive – Adj. Quality of something that is partly defined in terms of itself.
See recursive*

Once we have conditionals we can express *recursive* methods, that is methods that are partly expressed in terms of themselves. Recursive methods allows one to write extremely powerful algorithms in an elegant way. In this chapter we simply show you the main principle without going into the details.

1 Picking Diamonds

Let us start with the following problem. Imagine we want to pick all the diamonds in the direction to which the robot is pointing at. Using recursion we can define the method `pickLine` (See 6.1).

Method 6.1

In category recursion
Bot>>pickLine

```
self canMoveForward
  ifTrue:
    [self safePick.
     self go.
     self pickLine ]
```

How does it work? In fact writing recursive method is natural and simple. Writing recursive methods that do not loop endlessly is more complex. Let us have a look at what's happen when the method is executed

1. the method is invoked,
2. the condition `canMoveForward` is executed,
3. when the condition value is false the method terminates.
4. when the condition is true, the conditional messages are executed one by one. When the method `pickLine` is invoked the process starts as mentioned to the point 1.

When defining a recursive method, the three following points are important to consider.

- The first key point is to always check that the method contains at least one case that is not calling the method again. Here the non recursive part is implicit as we used the method `ifTrue::` when the condition is false the conditional messages are not executed anymore and the method `pickLine` is not recursively invoked.
- The second point is that the recursive part of the method should somehow tend towards stopping the recursive calls that is the condition should be false. Here the fact that the area is bounded and that the bot moves always one step in the same direction ensure that eventually the bot will not be able to move further that is the expression `self canMoveForward` will be false.
- The third point that will be shown in the section 5 is that a typical recursive method builds its result by composing the partial results obtained by recursive calls.

Another Example of Recursive Method

In Chapter 3 we defined and used a method named `randomDirection` which returned a direction at random for the robot. This method tries to find a new direction, that is a point composed by -1, 0, or 1 with the extra constraint that $x*y$ should always be equal to 0. When it did not find a point satisfying the constraint it just retries invoking itself.

Method 6.2

```

Bot>>randomDirection
"return a random direction i.e. a point composed by -1,0, or 1
with the constraint that x * y should always be = to 0"

| x y |
x := 3 atRandom - 2.
y := 3 atRandom - 2.
^ (x * y) isZero
  ifTrue: [x @ y]
  ifFalse: [self randomDirection]

```

Note that the method `randomDirection` could also be implemented using a conditional loops. Try to do it.

A terminating recursive method should have at least a non recursive branch and the recursive branch should tend toward invalidating the condition to exit the recursive branch.

2 Learning from Errors

While writing for the first time the method `pickLine`, we did a mistake, we forgot the expression `self go`. Can you imagine what was the result?

The robot could move so the method `canMoveForward` was true. Therefore the conditional messages were executed. A diamond was picked up if possible and then the method `pickLine` was called again. So we ended up in an endless loop.

What can we learned from this bug? An essential point, a recursive method have to always tend towards its non recursive part. This means that if we want that the method stops as it should, we should at least give it a chance to stop. When you are writing a recursive method always have in mind that you should (1) have a non recursive part, here the fact that we have an `ifTrue:` means that when the robot cannot move we will not be in the recursive part anymore, and (2) you should tend toward this situation, here we expect that by moving the bot forward it will end up bumping into the limits of its world and this will lead to executing the non recursive part, here doing nothing and ending the execution.

In fact the method `pickLine` is not working completely well. It will not pick a diamond that is just in front of a brick or near the limit of the world. First explain why by showing step by step why this situation occurs. Second propose a solution to this problem.

In fact picking a diamond can be done all the time therefore it does not require to be only executed within the conditional expression. The method 6.3 fixes this problem, as you see a recursive invocation will then at least invoke the method `safePick`.

Method 6.3

```
Bot>>pickLine

self safePick.
self canMoveForward
  ifTrue:
    [self go.
     self pickLine]
```

A recursive method is a method partly defined in terms of itself.

3 Fun with Recursion

In this section we want to show you how to define intriguing curves, fractal curves whose definition is recursive. The idea of a fractal curve is to define – part of it – using its own definition. Let us start to try to find the definition of the method that can draw the pictures shown in Figure 6.1.

The idea is that we take a single line, cut it in three equal parts and replace the middle segment by a shape, here simply two lines. Then we apply the same principle on the all the segments we create. We obtain the drawings shown in Figure 6.1.

Let us start with a non-recursive method. The method `pic:` (see 6.4) draws the second picture from the left in Figure 6.1: it replaces the middle segment of a line based on three segments by a simple shape, here a triangle.

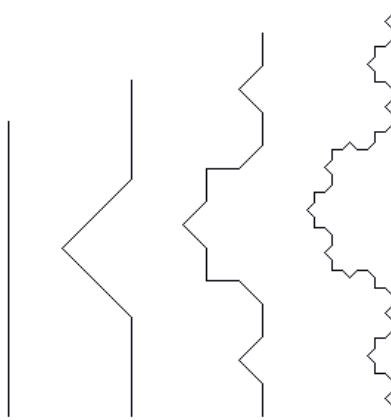


Figure 6.1: The `pic:n` with $n=1, 2, 3,$ and 4 .

Method 6.4

```
Turtle>>pic: size
"self new north; pic: 200"

| segSize |
segSize := size / 3.
self go: segSize.
self turnLeft: 45.
self go: segSize.
self turnRight: 90.
self go: segSize.
self turnLeft: 45.
self go: segSize
```

Note that we could have paid attention that the basis of the triangle would measure exactly a third of the original line but this does not change anything to what we want to show you and we suggest you to do it. Now we want to operate the same on each segments therefore we replace all the original `go:` calls by a call to the recursive method we are defining, we name it now `picRec:` to make it explicit and we obtain the method – not working yet – described in method 6.5. Can you explain why this method is not working?

Method 6.5

```
Turtle>>picRec: size
  "Turtle clearWorld. self new north; picRec: 200 ; beInvisible"

  | segSize |
  segSize := size / 3.
  self picRec: segSize.
  self turnLeft: 45.
  self picRec: segSize.
  self turnRight: 90.
  self picRec: segSize.
  self turnLeft: 45.
  self picRec: segSize
```

After reading the beginning of this chapter, you should ask yourself the following questions:

- How can it stop? Indeed there is no condition to stop the infinite recursive, each time `picRec:` is executed it has no alternative except to invoke itself infinitely.
- Where the job is done? Even if the method would contain a non-recursive call, there should be a place where the line should be drawn.

Even if these two questions are quite rudimentary, every recursive method should provide an answer to have a chance to work. The final version of the method `picRec:` (see 6.6) first contains a non recursive branch, second it defines a reasonable condition as the length of the drawn segment is regularly divided, and a place where the curve is drawn.

Method 6.6

```
Turtle>>picRec: size
  "Turtle clearWorld. self new north; picRec: 200 ; beInvisible"

  | segSize |
  size < 10
  ifTrue: [self go: size]
  ifFalse:
    [segSize := size / 3.
    self picRec: segSize.
    self turnLeft: 45.
    self picRec: segSize.
    self turnRight: 90.
    self picRec: segSize.
    self turnLeft: 45.
    self picRec: segSize]
```

You should realize that this method is only drawing from a single place, the expression `self go: size` while all the other invocations only modify the parameters and define a different execution contexts such as changing the direction of the turtle.

Do the following experimentations:

- Replace some recursive calls to `picRec:` by `go:` and understand the result.

- Put a `self halt` in the middle to see that the all the recursive calls before the breakpoint are drawn but non after.

- Change the condition.

To let you better experiment with the concept of recursive methods, we define a method method named `picRec:n:` which uses an explicit counter as the number of subsegments we want to have (See 6.7). We replace the condition `size < 10`. The new condition checks the value of the extra argument `n` which controls the number of times the recursive call will be executed.

Method 6.7

```
Turtle>>picRec: size n: n
  "Turtle clearWorld. Turtle new north; picRec: 200 n: 1 ; beInvisible"

| segSize |
n = 1
  ifTrue: [self go: size]
  ifFalse:
    [segSize := size / 3.
     self picRec: segSize n: n - 1.
     self turnLeft: 45.
     self picRec: segSize n: n - 1.
     self turnRight: 90.
     self picRec: segSize n: n - 1.
     self turnLeft: 45.
     self picRec: segSize n: n - 1]
```

3.1 A Classic Curve

The dragon is a classic recursive curve. It is a space filling self avoiding curve that is that it fills space but without crossing its own path. We do not want to go into the details and let you discover its definition shown in the method 6.8. Figure 6.2 results from the script 6.1.

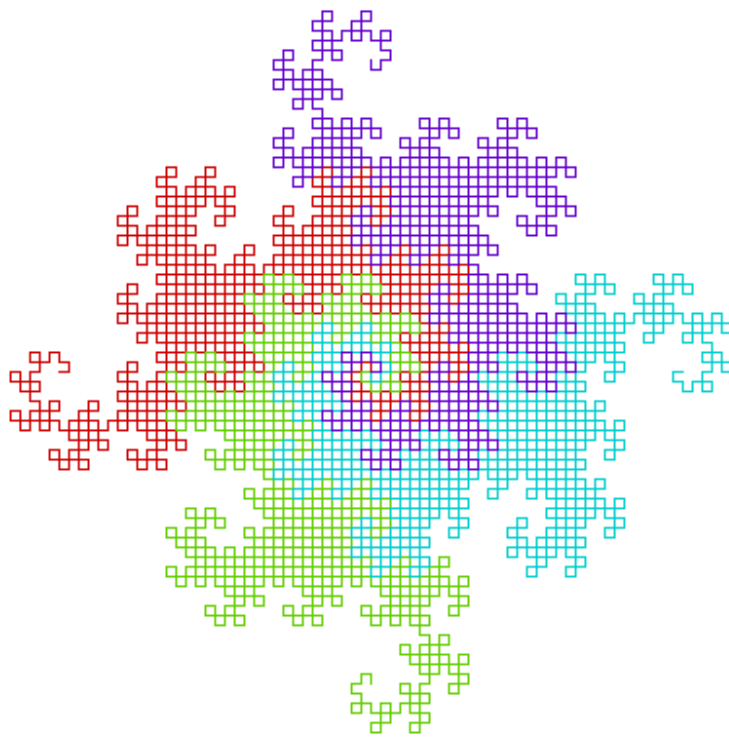


Figure 6.2: Four dragons filling the space produced by the script 6.1.

Method 6.8

```

Turtle>>dragon: n
"Turtle new dragon: 8"
"Draw a dragon curve of order n"

n = 0
ifTrue: [self go: 5]
ifFalse:
  [n > 0
   ifTrue:
     [self dragon: n - 1;
      turn: 90;
      dragon: 1 - n]
   ifFalse:
     [self dragon: -1 - n;
      turn: -90;
      dragon: 1 + n]]

```

The definition of the dragon curve shows how complex it can be to be sure that the non-recursive part of the definition is eventually reached. Here the condition $n = 0$ is not simply reached. Indeed when called when a positive number, the first recursive call (`dragon: n-1`) decreases n . This behavior is going towards the condition $n=0$. However, all the other recursive calls increase or negate the value of n . here the dragon curve definition terminates because of some mathematical properties.

Script 6.1 (Creating Figure 6.2)

```

|colors|
Turtle clearWorld.
colors := Color wheel: 4.
1 to: 4 do: [:i |
  Turtle new
    penColor: (colors at: i) ;
    turn: 90*i; dragon: 10 ;
    beInvisible]

```

4 About Keeping Context

Now we propose you to define a method that draws tree, regular and unrealistic trees but trees as shown in Figure 6.3. With this problem we illustrate the fact that sometimes we need a way to reset the context of execution before performing a recursive call. To draw a tree the principle is the following one, we draw one segment then we turn and draw a smaller tree, turn again, and draw another smaller tree. We follow this principle for the number of levels expected. From this principle we define the method `tree:level:` as shown in method 6.9 where the level specifies the number of segments that compose one branch (from the root to one leaf).

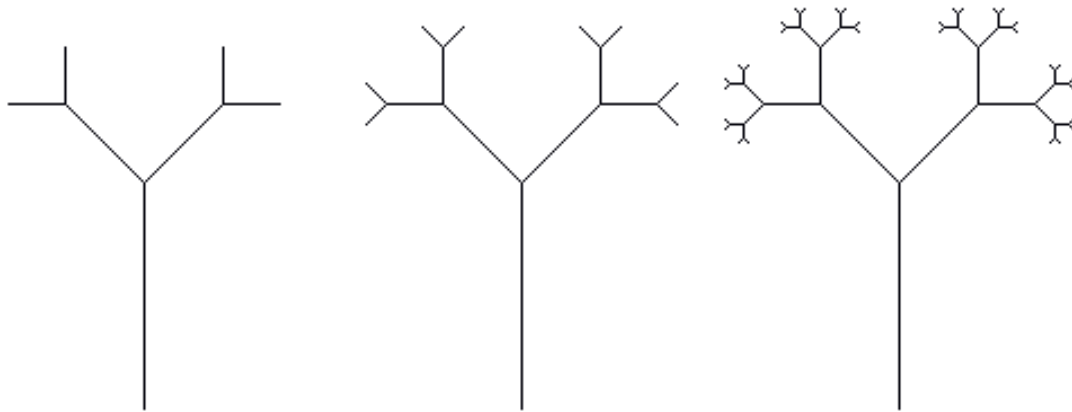


Figure 6.3: Some trees at different growing stages.

Method 6.9

```

Turtle>>tree: size level: n
  "Turtle clearWorld. Turtle new north; tree: 200 level: 1;
  beInvisible"

  n isZero
  ifFalse:
    [self go: size / 2.
     self turnLeft: 45.
     self tree: size / 2 level: n - 1.
     self turnRight: 90.
     self tree: size / 2 level: n - 1]

```

The definition of the method `plant:level:` answers the two questions we enounced for the method `picRec:`. There is a non-recursive part, here when the condition is true that is `n` is equal to zero then nothing is executed and the method terminates. There is also a part of the method drawing the expression `self go: size / 2`. Finally, the argument `n` on which is built the condition is decreased regularly so we are sure that at one point in time the condition will be true and that the recursion will stop. Executing the method with `n = 1` works well. However, with `n=2` we end up with the second drawing of Figure 6.4 and not the third as we were expecting it. Can you understand why? Step through the execution of the message `Turtle new tree: 200 level: 2`. Have you noticed that the first branch is drawn correctly? Comment out the second invocation to `tree:level:` to see it. What are the position and the direction of the turtle just before the second recursive call?

The problem with the current definition of `tree:level:` is that after the first recursive invocation, the turtle is not at right place ready to perform again the second recursion. Put a breakpoint to stop the execution as shown in the method 6.10 to obtain the same situation as the one illustrated in Figure 6.4. Imagine a solution to solve this problem.

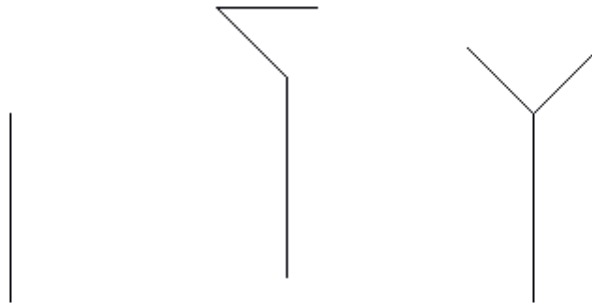


Figure 6.4: Trying to produce a tree.

Method 6.10

```

Turtle>>tree: size level: n
"Turtle clearWorld. Turtle new north; tree: 200 level: 1;
belInvisible"

n isZero
  ifFalse:
    [self go: size / 2.
     self turnLeft: 45.
     self tree: size / 2 level: n - 1.
     self halt.
     self turnRight: 90.
     self tree: size / 2 level: n - 1]

```

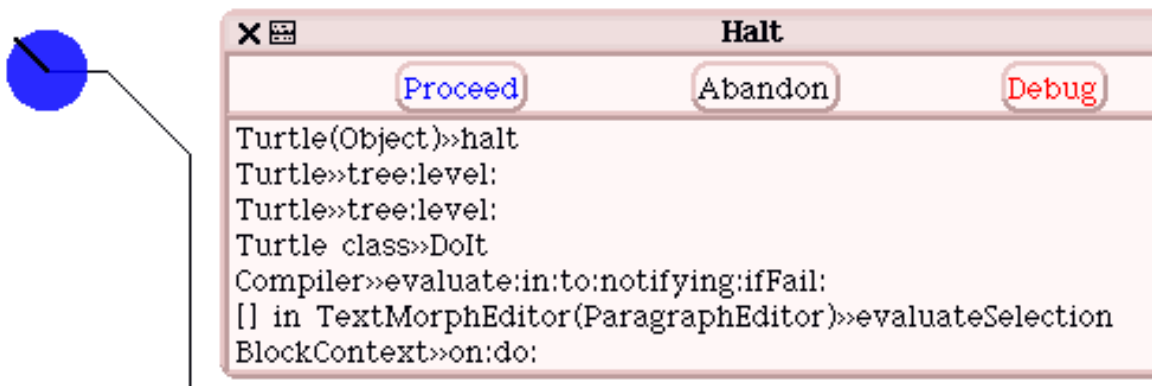


Figure 6.5: Stopping the method execution just after the first recursive invocation as described in the method 6.10.

This is a bit vicious. We cannot *simply* move back the turtle at its starting position after the first

call is terminated because it depends on the number of levels it already drew. Still there is one solution which is to store the position of the turtle and its direction before invoking the method and restoring this state after. This is what the method 6.11 does. The variables `pos` and `dir` for every call ensures that the turtle is in the right position and heading to draw the second tree.

Method 6.11

```
Turtle>>tree: size level: n
  "Turtle clearWorld. Turtle new north; tree: 200 level: 3;
  belnvisible "

  | pos dir |
  n isZero
  ifFalse:
    [self go: size / 2.
     self turnLeft: 45.
     pos := self center.
     dir := self direction.
     self tree: size / 2 level: n - 1.
     self center: pos.
     self direction: dir.
     self turnRight: 90.
     self tree: size / 2 level: n - 1]
```

5 A More Traditional View on Recursion

Recursion has been a way to define suite of numbers for centuries. It is interesting to look at them, as they present a simple context to understand how a result is defined out of the results obtained from recursive calls on smaller entities. We have a look at the factorial and fibonacci suites.

5.1 Factorial

Factorial is a classical mathematical function to present recursion. The mathematical definition of factorial is $n! = n*(n - 1), !0 = 1$. For simplicity we consider that for negative numbers, $n! = 1$, even if the mathematical function does not allow this. Therefore $3! = 3*2! = 3*2*1! = 3*2*1 = 6$. The method definition simply follows the mathematical definition (see 6.12).

Method 6.12

```
Integer>>fact
  "Answer the factorial of the receiver."

  ^ self <= 0
  ifTrue: [1]
  ifFalse: [self * (self - 1) fact]
```

What is interesting with this definition is that the result of the factorial is *composed of* the factorial of a previous factorial on smaller numbers. In particular, the computing the final result requires a sub computation of the same function to be computed.

The method is executed as follows:

1. the condition is executed,
2. when the condition value is true, the method terminates and returns 1.
3. when the condition is false, the conditional messages are executed one by one, `self -1` is executed, the method `fact` is invoked to the result of `self -1` again invoked, which is similar to the point 1. The part `self *` is “waiting” so that the previous call terminates by returning a value.

The following trace shows how the computation is evolving and composed over times. The bold numbers represent the result of a call. For example the trace shows that `0 fact` returns 1, `1 fact` returns 1 and `3 fact` returns 6.

```
5 fact
5 * 4 fact
5 * 4 * 3 fact
5 * 4 * 3 * 2 fact
5 * 4 * 3 * 2 * 1 fact
5 * 4 * 3 * 2 * 1 * 0 fact
5 * 4 * 3 * 2 * 1 * 1
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

An experiment, compare the first definition of `fact` with the following one that is the one defined in Squeak and make sure you understand their differences.

Method 6.13

```
Integer>>factorial
"Answer the factorial of the receiver."

self = 0 ifTrue: [^ 1].
self > 0 ifTrue: [^ self * (self - 1) factorial].
self error: 'Not valid for negative integers'
```

Factorial is a really simple function that can also be defined using a simple loops. Try to define it without using recursion.

5.2 The Fibonacci Suite

Another well-known suite is called the Fibonacci suite. This suite is present in the growth of plants or shells. Here it shows that a recursive method does not have to only call once itself but can do it several times as the dragon curve and the plant drawings where doing it.

The fibonacci suite is defined as follows: $fib(0) = 1, fib(1) = 1, fib(n) = fib(n - 1) + fib(n - 2)$. Therefore $fib(3) = fib(2) + fib(1) = fib(1) + fib(0) + 1 = 3$. The method `fib` directly defines the mathematical definition (see 6.14).

composition is the key point because we need to combine partial result to produce the complete result. With factorial this is just a multiplication but it is often more complex.

While the computation is performed, the memory of the computer is used to keep the rest of the computation. For example, the fact that 5 factorial leads to compute $5 * 4 *$ is kept in memory while 3 factorial is computed. We say that the stack grows as we are adding new computation to be finished on the stack. Computation that has to be finished first to be able to compute the rest. When 3 factorial terminates, the stack of unfinished computation is popped. Once the computation of 3 factorial terminates, the $4 *$ can be done and so on. However recursive computation expressed this way are limited by the memory of the computer available. Quickly the computer memory can be filled up by all these computation suspended.

However, the same recursive computation can be expressed differently without requiring to extra memory. The idea is to use a parameter that acts as an accumulator in which intermediate results are passed from one invocation to the other. We define the method `factorial: res` that computes the factorial when its argument is 1 (see 6.15). Then we define the method `fact` that invokes the method `factorial: res` with the right argument (see 6.16).

Method 6.15

```
Integer>>factorial: res

^ self <= 0
  ifTrue: [res]
  ifFalse: [self - 1 factorial: res * self]
```

Note that we named the argument of the method `factorial: res` to stress the fact that this argument is the result of the previous computations.

Method 6.16

```
Integer>>fact
"Answer the factorial of the receiver."

self = 0
  ifTrue: [^ 1].
^ self factorial: 1
```

The following trace shows how the computation evolves. Over times the argument contains the results of the previous computation and at the end is returned as result of the complete computation.

```
5 fact
5 -1 factorial: 5 * 1
4 factorial: 5
4 -1 factorial: 5 * 4
3 factorial: 20
3 -1 factorial: 20 * 3
2 factorial: 60
2 -1 factorial: 60 * 2
1 factorial: 120
120
```

120

As the trace shows it there is no computation pending, all the computation is done during each recursive call. Hence there is no need for extra memory to keep the pending computation. This kind of recursion is called tail-recursive to denote the fact that there is no computation required after.

Final Experiments. Do you guess the result of `5 fact` when we change the condition `self <= 0` by `self < 0`. To help you to understand how this works introduce a trace as follow.

Method 6.17

```
factorial: res
```

```
self = 0
  ifTrue: [^ res].
self > 0
  ifTrue: [Transcript show: self printString , ' res ' , res printString;
          cr.
          ^ self - 1 factorial: res * self]
```

Summary

- A recursive method is a method partly defined in terms of itself.
- A terminating recursive method should have at least a non recursive branch and the recursive branch should tend toward invalidating the condition to exit the recursive branch.

Collections: Grouping Objects

Up until now in all the problems we solved we had a couple of objects interacting with each other such as a turtle using a point or number. But we only manipulated one single object. However, there are a lot of situations where we would like to manipulate a group of objects in a similar fashion. For example we would like to perform the same actions over all the elements of a collection. In this chapter we present how to create some simple collections and how we define sequence of messages performed on all the collection elements. The concepts presented in this chapter and then reused in the two following chapters.

1 A First Look at Collection

A collection is an object that holds references to a group of other objects. For example, a collection can group together a set of points defining a polygon. A music partition holds a sequence of musical notes.

Smalltalk offers a large variety of collections. These collections differ in terms of the kinds of elements they can contain, whether they preserve the order of the elements they contain, whether they can contain several times the same object, whether they can grow when they are full. Here we just present the simplest collection named `Array`. An array can contain any kind of objects, any number of times. It conserves the order in which the objects are added to it but it cannot grow when it is full. Arrays are created by the class `Array`. The other really useful collection is ordered collection, it is similar to array except that it can grow with it is full.

Array Creation. The script 7.1 shows one handy way using the special characters `{` and `}` to create array. It creates an array containing 3 points. As arrays conserve the order, the first point of the collection is `300@300`, the second is `350@300`, and the last one `400@375`).

Script 7.1 (*Creating and filling arrays*)

```
{ 300@300 . 350@300 . 400@375 } "an array with three elements"
```

The character `{` defines the beginning of the array and the character `}` ends the array. The elements are separated by a dot `.`. Therefore, the first line of the script 7.2 creates an array with one element and the second line creates an array with no element.

Script 7.2 (Other arrays)

```
{ 300@300 } "an array with only one element"
{ } "an array with no element"
```

Important Messages 7.1

```
{ anObject1 . anObject2 . anObject3 . anObject4 . anObject5}
```

Script 7.3 (Another way to create arrays)

```
{ 300@300 . 350@300 . 400@375 }
myArray := Array new: 3.
myArray at: 1 put: 300@300.
myArray at: 2 put: 350@300.
myArray at: 3 put: 400@375.
```

Sending Messages to a Collection. As everything else in Squeak, collections are objects. As any objects they can be named using variables. The script 7.4 shows that we name `aCol` the previous collection. Then using this variable we interact with the collection the same way we send messages to a turtle. In the script 7.4 we send different messages to the collection to show some of the possibilities. We ask its size using the message `size`. We ask whether the collection is empty which is false using the message `isEmpty`. We ask whether it contains the point `300@300` using the message `includes:`. We ask the number of times it contains the point `300@300` and finally we ask to open an inspector on the array. The resulting inspector is shown in Figure 7.1.

Script 7.4 (Interacting with an Array)

```
| aCol |
aCol := {300@300 . 350@300 . 400@375}.
aCol size
-> 3
aCol isEmpty
-> false
aCol includes: 300@300
-> true
aCol includes: 310@300
-> false
aCol occurrencesOf: 300@300
-> 1
aCol inspect
```

Accessing and Modifying Collections. The next important actions we would like to perform with a collection is to access the element it contains. The Squeak collections offer a lot of possibilities to query their elements. Here we limit ourself to the strict minimum and show the two messages `aCollection at: anIndex` and `aCollection at: anIndex put: aValue`. The method `at: anIndex`

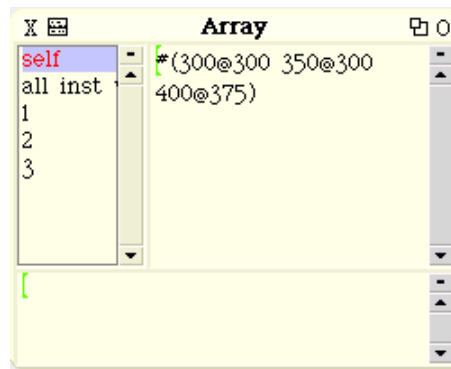


Figure 7.1: Inspecting a collection { 300@300 . 350@300 . 400@375 } inspect .

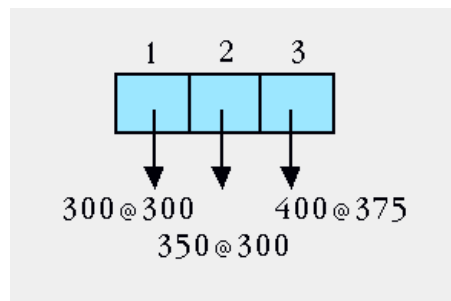


Figure 7.2: The array {300@300 . 350@300 . 400@375} .

accesses the element of a collection which is contained at the index specified. In the script 7.5 shows how to access the first element of a collection. Here we obtained the first point. The method `at: anIndex put: aValue` allows one to modify the element contained at a given index. In the script 7.5 we change the second element of the collection which is now `210@300`. Note that these messages only make sense for collections conserving the order of their contained elements.

Script 7.5 (Accessing and Modifying an Array)

```
| aCol |
aCol := {300@300 . 350@300 . 400@375 }.
aCol at: 1
-> 300@300

aCol at: 2 put: 210@300.
aCol at: 2
-> 210@300
```

As you may have guessed, collection index starts at 1 in Smalltalk. This means that the first element of a collection has the index 1, the *n*th element is at the index *n* as shown by figures 7.2 and 7.1. You can really think a collection as a group of variables, variables that contains addresses to objects they contain as we explained in Chapter ??.

Now the next question is what is happening when we try to access an element using an index that is out of the bounds of the collection. When this situation occurs the system raises an error as show

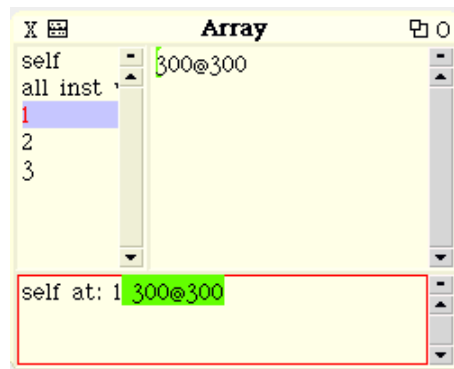


Figure 7.3: Querying the elements of a collection using an inspector.

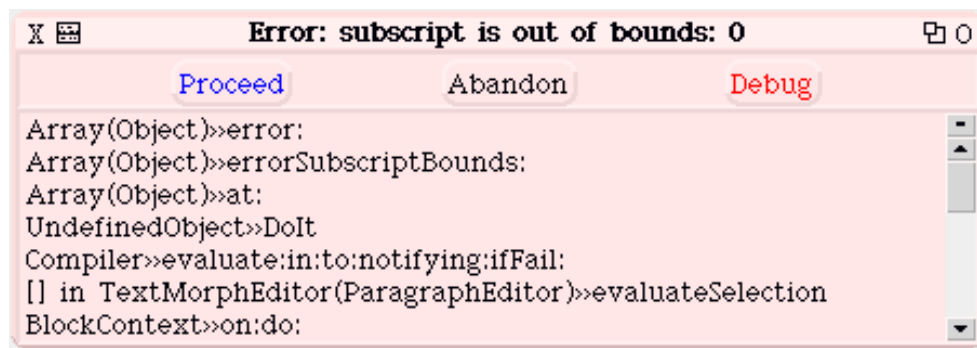


Figure 7.4: Error raised due to accesses out of the bounds of the collection.

by Figure 7.4 generated by the different scenario of the script 7.6.

Script 7.6 (Error caused by accessing out of the bounds elements)

```
| aCol |
aCol := { 300@300 . 350@300 . 400@375 }.
aCol at: 0
-> raises an error
aCol at: 5
-> raises an error
aCol at: -12
-> raises an error
```

Collections containing Turtles. Up until now we only manipulated points but collections can contain any objects and in Smalltalk even objects of different classes. We say that collections can be *heterogenous*. The script 7.7 shows how we manipulate a collection of turtles and send messages to the turtles that are contained in the collection.

Script 7.7 (Creating a Collection of Turtles)

```
| aCol |
aCol := {Turtle new . Turtle new . Turtle new}.
(aCol at: 1)
  color: Color red ; north; go: 100.
(aCol at: 2)
  color: Color green; jump: 100.
(aCol at: 3)
  west ; jump: 200.
```

2 Grouping Arguments

We can use collections to pass multiple objects as one single argument when sending a message. The following example illustrates how a basic collection is created, filled up with objects, and pass as arguments to a method. Grouping multiple objects like integers, points in a single collection is very useful because it allows us to create methods with limited number of arguments or to return multiple values. In the chapter ?? you defined and invoked the method `triangleFirstPoint:secondPoint:thirdPoint:` as shown by the method 7.1: and invoked script 7.8.

Script 7.8 (Use of `triangleFirstPoint:secondPoint:thirdPoint:`)

```
| caro |
caro := Turtle new.
caro
  triangleFirstPoint: 300@300
  secondPoint: 350@300
  thirdPoint: 400@375
```

Method 7.1

```
triangleFirstPoint: first secondPoint: second thirdPoint: third
  "Draw a triangle as specified by the points passed as argument"

  self jumpAt: first.
  self goAt: second.
  self goAt: third.
  self goAt: first.
```

Now to avoid to use three arguments we use a collection of points as argument. The script 7.2 shows the method `triangleFirstPoint:secondPoint:thirdPoint:` rewritten.

Method 7.2

triangle3Points: *pointCollection*

"Draw a triangle as specified by the points passed as argument"

self jumpAt: (*pointCollection* at: 1).

self goAt: (*pointCollection* at: 2).

self goAt: (*pointCollection* at: 3).

self goAt: (*pointCollection* at: 1).

Now to invoke the method triangle3Points: we need to create a collection containing three points.

Script 7.9 (Invoking the method triangle3Points:)

```
| caro |
caro := Turtle new.
caro triangle3Points: { 300@300 . 350@300 . 400@375 }
```

3 Iterating over a Collection

Been able to group several objects together and manipulate them is only a simple aspect of collection. Their real power is that we can treat *all* the elements of a collection without having to distinguish them. This means that we can apply the *same* operations to *all* the elements of a collection. Moreover in Smalltalk we are able to execute the same operations to *different* collection. The expression "to iterate over a collection" means that we go over all the collection and execute the sequence of message to each of them. Smalltalk proposes powerful ways of manipulating or iterating over collections as briefly presented in section 6. Here we limit ourself to a simple iteration.

An Example: Drawing Polygons. To draw a polygon we can define a script that similar to the script 7.10. First, we position a turtle on the first point and then we ask it to go at each point.

Script 7.10 (Polygon without collection use)

```
| caro |
caro := Turtle new.
caro jumpAt: 300@300.
caro goAt: 400@400; goAt: 433@348; goAt: 392@350.
caro goAt: 397@314; goAt: 363@322; goAt: 367@284.
caro goAt: 333@287; goAt: 321@252; goAt: 300@300
```

This is boring to repeat all the times that the turtle has to go at a given point. Moreover, every times we want the turtle to go to another point we have to repeat the same code. However, we cannot use loops like timesRepeat: because the code to repeat is *not exactly* the same. The locations where the turtle should go are changing each time. Here the action to be repeated is trivial but there are cases where the repeated actions are fairly complex. Worse, if we want to draw another polygon, we will have to rewrite everything.

We could use the same technique used in the method triangle3Points:. However this technique is limited because we could only draw polygons having the same number of points. We want to be able to draw polygons having *any* number of points. What we would like to be able write is a method that

will work independently of the number of points. We need a way to specify a sequence of messages to be sent to each of the elements of a collection without knowing their number.

Smalltalk offers a really powerful solution to this problem. It provides a way to *define* and to *execute* the same actions to each element of a collection. The solution is based on sequence of messages, blocks, that are executed with each individually elements of a collection.

A First Look at The Solution. The script 7.11 is strictly equivalent to the script 7.10. The difference is that we use a collection to hold all the points of the polygon and we execute to each of them the same group of actions.

Script 7.11 (*Polygon with collection use*)

```
| caro points |
caro := Turtle new.
points := {300 @300. 400@400 . 433@348 . 392@350. 397@314 . 363@322 . 367@284.
333@287. 321@252 . 300@300}.
points do: [ :aPoint | caro goAt: aPoint]
```

- First, we create a collection that we assign to the variable `points`.
- Second, the expression `points do: [:aPoint | caro goAt: aPoint]` sends the message `goAt:` to the turtle named `caro` for each of the points contained into the collection named `points`. The same sequence of messages with each element of the collection is executed using the method `do:`. The sequence of messages, a *block*, delimited by `[` and `]`, specifies the sequence of messages that will be sent to each elements.

The method is sent to the collection named `points` with a **block** as argument. As we already saw a block simply represents a *sequence of messages* and is delimited by the characters `[` and `]`. In a similar way than methods, a block can have arguments as we already see in Chapter ???. Here the block has one argument named `aPoint`. During the iteration over all the elements of the collection, this variable represents the current element of the collection that is currently treated.

Here we ask the turtle to move to each point contained in the collection, we have to have a way to refer to the current point treated while we are passing over it. Therefore we need a way to represent and be able to send messages to the current element of a collection with which the block is executed. This the role of the expression `[:aPoint |` in the expression `[:aPoint | caro goAt: aPoint]`. This expression declares that the block has one argument named `aPoint`. The symbol `|` in `[:aPoint |` separates the declaration of the argument from the sequence of messages that uses it.

To summarize, we have a block with one argument named `aPoint` that represents the current element of the collection. This is the `do:` method that executes the block with each elements of the collection. As we will see in the section 6 blocks can be used for other purposes.

Helper's Hints

A block is similar to mathematical function. The function $x \rightarrow 2x + 1$ can be represented as the block `[:x | 2*x + 1]`. $f(x) \rightarrow 2x + 1$ can be represented as `|f| f:=[:x] 2*x + 1]`. $f(3)$ is `f value: 3`.

Helper's Hints

Analysis. Using a collection makes our life easier because adding a new point into our polygon does not require to modify the code of the script but only adding a new element into the collection. The same block can be reused on an entirely different collection of points.

Note that the method `timesRepeat:` also needs a block as argument. The difference is that with `timesRepeat:` we did not need to manipulate anything because we are simply repeating **exactly** the same sequence of message. With the message `do:`, we repeat a sequence of messages parameterized with each element of the collection.

4 Collection as Method Arguments

Now if we want to draw another polygon we will write another script that is quite similar to the script 7.11. The only difference is the number and points. This is an opportunity for defining a method that given a collection of points draws a polygon. However, as different polygons are composed by different numbers of points, we need a way to define method having variable number of arguments. For this purpose we can use as argument a collection as we did in the Section section 2.

Define a method named for example `polygonFromPoints: aCollection` that draws a polygon from a collection of points and could be called as presented in the script 7.12. A possible definition of the method `polygonFromPoints: aCollection` is given in method 7.3.

Script 7.12 (Use of the method `polygonFromPoints:`)

```
| caro |
caro := Turtle new.
caro polygonFromPoints: {300 @300. 400@400 . 433@348 . 392@350. 397@314 . 363@322 . 367@284.
333@287. 321 @252 . 300@300}.
```

Method 7.3

```
polygonFromPoints: aPointCollection
  "Draws the polygon specified by aPointCollection"
  is in aPointCollection"

self jumpAt: aPointCollection first.
aPointCollection do: [:aPoint] self goAt: aPoint].
```

5 Translating and Tiling

Instead of considering the points contained the collection as the points defined from reference to the origin, we can consider that the points are relative to a reference point. This way it is easy to translate a polygon to all kinds of locations.

Define a method `polygonFromPoints: aPointCollection at: aReferencePoint` that draws a polygon from a list of points and starting at a given point. Such a method could be used as shown by the script 7.13. A possible definition is given by method 7.4.

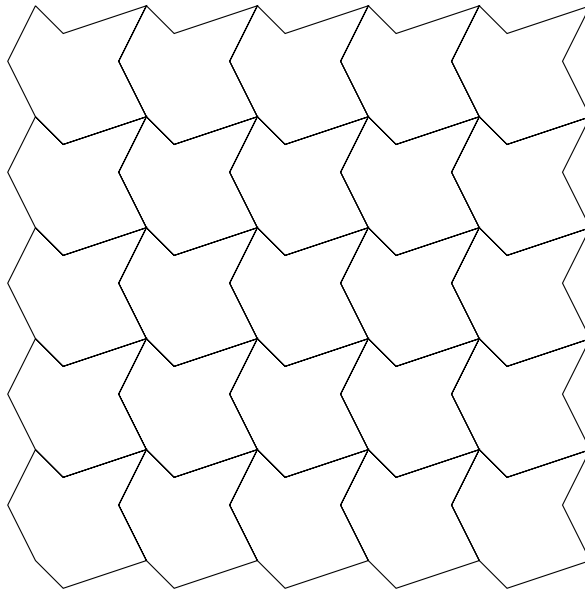


Figure 7.5: Repeating the same deformed square to produce repeated structures.

Script 7.13 (Use of the method *polygonFromPoints:at:*)

```
| caro points |
caro := Turtle new.
points := {300 @300. 400@400 . 433@348 . 392@350. 397@314 . 363@322 . 367@284.
333@287. 321@252 . 300@300}.
caro polygonFromPoints: points at: 100@100
```

Method 7.4

```
polygonFromPoints: aPointCollection at: aReferencePoint
  "Draws the polygon specified by aPointCollection at aReferencePoint"

self jumpAt: aPointCollection first + aReferencePoint.
aPointCollection do: [:aPoint| self goAt: aPoint + aReferencePoint].
```

Basic Escher Tiling. When the famous artist Escher started to work on repeating patterns to produce beautiful and intriguing pictures, he looked for systematical ways of producing and placing tiles that fit together. In his first studies he identified different forms that could be created and composed. The simplest technique used was to distort a square by making the same transformation on opposite edges two by two. This way the tiles fit exactly using a translation. Figure 7.6 produced by the script 7.14 illustrates an example of square shape change. Here we simply added a new point in each edge. The method 7.5 shows how to draw such a tile with a 100 pixel size.

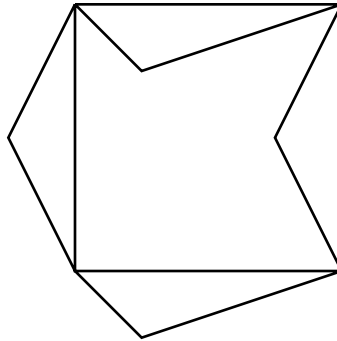


Figure 7.6: Transforming a square by changing symmetrically its edge.

Script 7.14 (*Producing the picture 7.6*)

```
| caro |
caro := Turtle new.
caro tileStart: 200@200.
caro jumpAt: 200@200.
4 timesRepeat: [caro go: 100. caro turnRight: 90]
```

Method 7.5

```
tileStart: startPoint
  "Draws a tile of 100@100"

  self jumpAt: startPoint.
  self goAt: startPoint + (25@25).
  self goAt: startPoint + (100@0).
  self goAt: startPoint + (75@50).
  self goAt: startPoint + (100@100).
  self goAt: startPoint + (25@125).
  self goAt: startPoint + (0@100).
  self goAt: startPoint + (-25@50).
  self goAt: startPoint.
```

This definition of the method `tileStart: startPoint` (method 7.5) is not really good because it reinvent the way we draw polygon. Express it using `polygonFromPoints:at:`

To produce a complete surface we need to draw a line of tiles. The idea is to draw one tile and draw the next one shifted by the size of the original square. Define a method called for example `lineTile` that produces such a line of tiles. The method 7.6 proposes a solution that does not allow one to specify the number of tiles drawn in the line. Propose a solution that would allow one to specify the number of tiles.

Method 7.6

```
lineTile
  "Draw 10 tiles horizontally"

  | start |
  1 to: 10 do: [:i | self tileStart: ((100*i)@0)]
```

To draw different lines we need to be able to specify the y coordinates where the line should start. Based on the method 7.6 defines a method called `lineTileAtY: anInteger` that draws a line of tiles at the ordinate specified by `anInteger`. Using the method 7.7 you can now produce the drawing shown in Figure 7.5.

Method 7.7

```
tilingWithColumns: anInteger
  "Draws anInteger times lines of tiles"

  1 to: anInteger do: [:j | self lineTileAtY: 100 *j]
```

Further Experiments. Having a fixed size for the tile is not really flexible, introduce the possibility to change the size of the tile. The method `tileStart: startPoint size: anInteger` that draws a tile of a given size could look as follow.

Method 7.8

```
tileStart: startPoint size: anInteger

  | quarter |
  quarter := anInteger // 4.
  self jumpAt: startPoint.
  self goAt: startPoint + (quarter@quarter).
  self goAt: startPoint + (anInteger@0).
  self goAt: startPoint + ((quarter*3)@(quarter*2)).
  self goAt: startPoint + (anInteger@anInteger).
  self goAt: startPoint + (quarter@(anInteger+quarter)).
  self goAt: startPoint + (0@anInteger).
  self goAt: startPoint + ((quarter negated)@(quarter*2)).
  self goAt: startPoint.
```

Again the method 7.8 does not reuse the code of the method `polygonFromPoints:at:.` Rewrite it to use it. A solution may look like the method 7.9.

Method 7.9

```

tileStart: startPoint size: anInteger

| quarter points |
quarter := anInteger // 4.
points := { 0@0. quarter@quarter . (anInteger@0) .
            ((quarter*3)@(quarter*2)). anInteger@anInteger .
            (quarter@(anInteger+quarter)) . (0@anInteger) .
            ((quarter negated)@(quarter*2))}.
self polygonFromPoints: points at: startPoint.

```

On Turtles. Again collections can contain any objects. The script 7.15 shows some iterations performed on a collection of turtles.

Script 7.15 (Iterating over turtle collection)

```

| aCol |
aCol := {Turtle new . Turtle new . Turtle new}.
(aCol at: 1)
  color: Color red ; north; go: 100.
(aCol at: 2)
  color: Color green; jump: 100.
(aCol at: 3)
  west ; jump: 200.
aCol do: [:aTurtle | aTurtle extent: 30@30].
aCol do: [:each | 4 timesRepeat: [each go: 10. each turn: 90]].

```

6 Other Important Operations on Collections

Performing the same actions to each element of a collection is really useful. In addition Smalltalk offers other really powerful collection operations that we present briefly now. These iteration operations are the basic tools that every Smalltalk programmer uses daily. Note that the design of the Smalltalk collections has inspired collection libraries in other languages for example Java 1.2.

All the scripts presented in this Section should be evaluated within the context of the script 7.15 in which the variable `result` should be added. Note that some results depend on the size of the screen so you may get different results.

Collecting Results of Actions. Some times we want not only to execute a sequence of messages on all the elements of a collection using `do: aBlock` but also we are interested in getting all the results of the block execution. The method `collect: aBlock` executes the specified block, `aBlock`, to all the elements of the collection *and* returns a collection of the same size than the receiver containing all the results of the block execution on each elements. The script 7.16 presents some experiences. Once again the script shows that block arguments can be named in different manner. Here it is named `aTurtle` and `each`.

Script 7.16 (Using *Collection*»collect: aBlock)

```

results := aCol collect: [:aTurtle | aTurtle center].
results at: 1.
—Printing the returned value: 591@224
results at: 2.
—Printing the returned value: 691@324
results at: 3.
—Printing the returned value: 391@324
results printString.
—Printing the returned value: #(591@224 691@324 391@324)
aCol collect: [:each | each direction].
—Printing the returned value: #(90 0 180)

```

So now you should be able to guess the results returned by the expression `aCol collect: [:each | each]`. Guess and experiment to validate your hypothesis.

Selecting Elements. Sometimes we want to select all the elements of a collection that satisfy a certain condition. For this purpose two methods exist: `select: aBooleanBlock` and `reject: aBooleanBlock`. Both blocks should represent a boolean expression that is an expression returning true or false as presented in chapter ???. The message `select:` executes a block to all the elements of the collection and returns all the elements for which the block is true. The script 7.17 illustrates this.

Script 7.17 (Using *Collection*»select: aBooleanBlock)

```

"to select all the turtles pointing to the east in aCol"
results := aCol select: [:each | each direction = 0].
results size.
—Printing the returned value: 1
"All the turtles pointing to the east are now yellow"
results do: [:aTurtle | aTurtle color: Color yellow].

```

Script 7.18 (Using *Collection*»select: aBooleanBlock to reject elements)

```

"to select all the turtles not pointing to the east in aCol"
results := aCol select: [:each | (each direction = 0) not].
results size.
—Printing the returned value: 2

```

The method `reject: aBooleanBlock` performs the opposite than the method `select:` in the sense that it returns all the elements of the receiving collection that do *not* satisfy the block, that is for which the block returns false. The script 7.19 returns exactly the same result than the one of the script 7.18.

Script 7.19 (Using *Collection*»reject: aBooleanBlock)

```

"to select all the turtles not pointing to the east in aCol"
results := aCol reject: [:each | each direction = 0].

```

Finding an element. Getting all the elements satisfying a criteria is powerful but sometimes overkill. We often want to get an element satisfying a criteria. The method `detect: aBooleanBlock` returns the first element of the receiving collection that satisfying the block. The script 7.20 shows how to get the first turtle having an ordinate bigger than 100. We get a turtle so we can send it a message, here color.

Script 7.20 (Using *Collection*»*detect: aBooleanBlock*)

```
"To get the first turtle with an ordinate bigger than 100"
aCol detect: [:aTurtle | aTurtle center y > 100]
```

The main difference between `detect:` and `select:` is that the first one returns an element while the second all the elements satisfying the condition. The script 7.21 shows that `detect:` with the same condition than used in the script 7.20 returns a collection of turtle on which new operations can be applied.

Script 7.21 (All the turtles having an ordinate bigger than 100.)

```
"To get all the turtle with an ordinate bigger than 100"
results := aCol select: [:aTurtle | aTurtle center y > 100].
```

```
"We get a new collection of turtles so we can apply other operations"
results collect: [:each | each color]
```

The question that you certainly asked yourself is what is happening whether the collection does not contain an element satisfying the block. The answer is: an error is raised. The script 7.22 shows an example of such a failure. However, a companion method `detect: aBooleanBlock ifNone: aBlock` allows one to specify an action to perform instead of raising an error. This method has the following behavior, it applies the first block `aBooleanBlock` to the elements of the receiving collection one by one and returns the first one that satisfies the block, when none of the elements satisfies the block, the second block is executed and its result returned.

Script 7.22 (*Collection*»*detect: aBooleanBlock Failing*)

```
"a detect call that fails because there is no turtle with 37 as direction"
aCol detect: [:each | each direction = 37]
```

The script 7.23 shows how to use the method `detect:ifNone:.` First we specify the condition that the element should hold, here that the turtle should have its ordinate bigger than 700, *i.e.*, be located below the horizontal line 600, then we say that whether there is no element for which this condition is true, the method should return nil. As `res` does not refer to a collection but or to a turtle or nil, we test to see whether we get nil. In such a case we make some noise, else we change the color of the receiver to yellow.

Script 7.23 (Using *Collection*»*detect: aBooleanBlock ifNone: aBlock*)

```
res := aCol detect: [:aTurtle | aTurtle center y > 600] ifNone: [nil].
res isNil
  ifTrue: [Beeper beep ].
  ifFalse: [res color: Color yellow]
```

Note that we do not have to specify `[nil]` as a second block in a `detect:ifNone:` call. We can define any action we want, so it depends of what we want to exactly do. In the script 7.24 we create a new turtle, this way we are sure that the result returned will always be a turtle therefore we do not have to test the result and can directly apply the color change.

Script 7.24 (Another Collection»detect: aBooleanBlock ifNone: aBlock)

```
res := aCol detect: [:aTurtle | aTurtle center y > 600] ifNone: [Turtle new].  
res color: Color yellow.
```

Summary

Collection

do:, block, select, size, isEmpty, includes:, collect:

Block: Messages with Delayed Execution

Blocks are sequence of messages. They are extremely powerful and we already used them a lot. In the chapter ?? we used blocks to repeat the same sequence of messages. In the chapter ?? we used blocks to combine loops and variables. In the chapter 7 we used blocks to execute the same sequence of messages to all the collection elements. In this chapter we detail the use of blocks. We present their definition and delayed execution. In a first reading you can skip this chapter.

1 An Example

A block is a sequence of messages whose execution is deferred. In this sense it is quite close to a method except that it does not have a name and does not have to be defined in a class. A block is delimited by the characters [and]. Any message enclosed by these characters is delayed. This means that they are not executed and will be executed only when the block is asked to execute itself using the messages `value` or `value: anObject` as we show later.

You can also think about a block as equivalent to a mathematical function. For example, the block the block `[:x] 2*x + 1` represents the function $x \rightarrow 2x + 1$. In a similar fashion that in mathematics we name the function f , we can define a variable named `f`. Therefore, $f(x) \rightarrow 2x + 1$ can be represented as `|f| f:=[:x] 2*x + 1`. Then the function application is performed using the message `value:`. Hence, $f(3)$ is `f value: 3` as shown by the script 8.1.

Script 8.1 (A block with one argument)

```
| f1 |  
f1 := [:x | 2*x + 1].  
f1 value: 3  
-> 7  
f1 value: 125  
-> 251
```

2 Block Definition

In fact blocks are quite close to methods. You can look at blocks as if they would be anonymous methods that could be stored into variable, passed as arguments of other methods, and whose execution is delayed. As a method a block can have several arguments and several local variables as shown by

the template 8.1. In the template 8.1 the block has two arguments named `arg1` and `arg2` and one local variable named `localVariable`.

Important Messages 8.1

```
[ :arg1 :arg2 |
  | localVariable |
  anExpression.
  anotherExpression ]
```

Block's arguments work exactly the same way method arguments do. The script 8.2 shows that the expression `:aPoint |` defines an argument as done in the definition of the method `pointAt: aPoint`. In the expression `:aPoint |` the syntax is a bit more awkward: the name of the argument is prefixed by `:` and after the last argument the character `|` is used to specify that the list of arguments ends. The script 8.2 shows also that arguments in the block are accessed without the character `:`, hence when the argument `:aPoint` is defined, `aPoint` is used to send message to the object passed during the invocation of the block.

Script 8.2 (Similarity between block and method arguments)

```
points do: [ :aPoint | aPoint blabla ]
```

```
points do: [ :aPoint |
  aPoint blabla]
```

```
"Methods"
do: aPoint
```

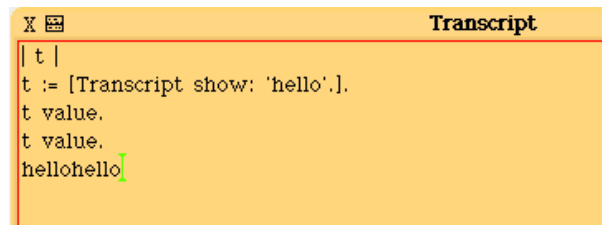
```
aPoint blabla
```

As with any variable, we can choose the name we want. The script 8.3 shows three equivalent expressions based on similar blocks having different argument names.


Script 8.3 (Three equivalent expressions)

```
points do: [:aPoint | caro goAt: aPoint]
points do: [:each | caro goAt: each]
points do: [:xfv123 | caro goAt: xfv123]
```

For the Expert. Up until the version 3.5 of Squeak and contrary to all the other Smalltalk available, Squeak does not implement full block closures. Normally blocks define their own variable environment, hence we can have two variables one as block argument and one as a local variable with the same name without problem (even if this is often considered as bad style because less readable). In Squeak this is not possible because blocks do not have their own environment and local variables or arguments are treated as special variable of their enclosing environment. So block in Squeak can give you errors that you would not get in other Smalltalk environment. In Squeak 3.6 a new compiler has been implemented that offers full block closures but at the price of some speed penalties. At the time of this writing it is not clear whether this new compiler will replace the old one.



```

X  Transcript
| t |
t := [Transcript show: 'hello'].
t value.
t value.
hellohello

```

Figure 8.1: Result of the script 8.4.

3 Invoking Block Execution

The sequence of messages contained in a block are not executed when the block is defined but only when we ask it explicitly using block methods `value`, `value: anObject`, and `value: anObject value: anotherObject`. Other methods exist but we limit ourselves to blocks with no, one and two arguments.

No Argument. In the script 8.4 we first define a block whose behavior is to write the string 'hello' to the Transcript. Once the block is defined we can execute it several times.

Script 8.4 (*Executing twice a block without argument*)

```

| t |
t := [Transcript show: 'hello'].
t value.
t value.

```

Note that the method `timesRepeat:` presented in chapter ?? and the conditional `ifTrue:ifFalse` presented in chapter ?? are heavily based on the use of block without argument. The only difference with the code here is that such methods are responsible to execute the block execution while in the script 8.4 you are responsible to execute the block.

One Argument. A block can have one argument. You already saw a lot of example of such uses. For example for the loops based on the message `aNumber to: anotherNumber do: aBlock` shown in chapter ?. All the collection operations presented in chapter 8 *i.e.*, `do: aBlock`, `collect: aBlock`, `select: aBlock`, `reject: aBlock`, `detect: aBlock`, and `detect: aBlock ifNone: aBlock`, required a block having one argument representing in this case an element of the collection. Take a look at the scripts 7.11, 7.17, and 7.6 to refresh your mind. The script 8.1 shows a simple example of block with one argument.

Two Arguments and more. Blocks can have more than one argument. The principle is the same: we specify the arguments and when we ask the block to execute itself we should then provide the corresponding number of arguments. For example the script 8.5 defines a block with two arguments and it shows that we should use the message `value: anObject value: anotherObject` to get the block executed. Here the block `[x :y | (x/3) + (y/5)]` has two arguments, namely `x` and `y`.

Script 8.5 (A block with two arguments)

```
| f2 |
f2 := [:x :y | (x/3) + (y/5)].
f2 value: 3 value: 5
-> 2
f2 value: 6 value: 1
-> (11/5)
```

The methods that require a block such as `timesRepeat:`, `ifTrue:ifFalse:`, `do:`, or `whileFalse:` controls the way the block is executed and as a programmer we do not have to worry about that. Eventually the block execution will be asked using the `value` related methods. The method 8.1 shows the definition of the method `timesRepeat:`. It shows that the block is executed using the message `value`.

Method 8.1

```
Integer>>timesRepeat: aBlock
"Evaluate the argument, aBlock, the number of times represented
by the
receiver."

| count |
count := 1.
[count <= self]
  whileTrue: [aBlock value.
    count := count + 1]
```

Blocks can contain *any* expressions so it can contains other blocks. The script 8.6 shows that the block of a `timesRepeat:` message can be used inside another block.

Script 8.6 (A block containing another block)

```
| t sq |
t := Turtle new.
sq := [:aTurtle :size |
  4 timesRepeat: [aTurtle go: size.
    aTurtle turn: 90]].
sq value: t value: 20.
sq value: t value: 30.
sq value: t value: 150.
```

4 About Returned Value of Blocks

As methods blocks return a value. When the construct `^` is not used, the value of a block is the value of its last expression as shown by the script 8.7. Hence printing the result returned by the expression `[Transcript show: 'hello'] value` is a `TranscriptStream` because the method `show:` returns its receiver, the `Transcript`, which is a `TranscriptStream`. The result returned by `[Transcript show: 'hello'. 2+3] value` is 5 because 5 is the value of the last expression `2+3` of the block.

Script 8.7 (About Block Value)

```
[Transcript show: 'hello'] value
—Printing the returned value:  a TranscriptStream
[Transcript show: 'hello'.
2+3] value
—Printing the returned value:  5
[Transcript show: 'hello'.
2+3.
6] value
—Printing the returned value:  6
```

We do not want to go in the full detail of the block semantics related to the use of explicitly return. For normal programming we consider that using `^` inside a block is the sign of a potential mistake.

For the Specialist. We said that a method always returns a value and that to explicitly return a value different than the receiver we should use `^`. We also mentioned that when a return expression `^` is executed it stops the execution of the method and provides a result to the caller and this at any levels. This behavior is also true for block. When a return is inside a block is executed, this leads to exit the method currently executing it.

The method `String>isAllDigits` defines on the class `String` that tells whether a string represents a number as illustrated in script 8.8, uses this escaping property of return execution as shown in the method 8.2.

Script 8.8 (Does a String represents a Number)

```
'12345' isAllDigits
-> true
'123d45' isAllDigits
-> false
```

The method iterates over the elements of the string (a string is composed by characters) and checks whether the character are representing digits. As soon as one single character is not representing a digit the method exits the loops and returns `false`.

Method 8.2

```
String>>isAllDigits
"whether the receiver is composed entirely of digits"

self do: [:c | c isDigit ifFalse: [^ false]].
^ true
```

Summary

- A block is a sequence of messages whose execution is delayed.
- A block is an anonymous method that can be assigned to variable and passed as argument. It can have arguments and local variable. Block arguments are declared by prefixing them with :

and the declaration is terminated by `|`. `[:aPoint | aPoint + (2@2)]` defines a block with one argument named `aPoint`

- Block are executed by sending them the message `value` for blocks with no argument, `value:` for blocks with one argument, `value:value:` for blocks with two arguments.

5 Other Loops with Variables

In this section we would like to show you two new methods `to:do:` and `to:do:by:` that are handy when working with loops as you do not have to declare a loop variable, initialize it and increase it explicitly.

In the script `??`, we defined and initialized the variable `length`, then we increase ten times its contents by 10 that we show again in the script 8.9.

Script 8.9 (Creating a flat growing stair.)

```
| caro length |
caro := Turtle new.
length := 10.
10 timesRepeat: [caro go: length.
                 caro turnLeft: 90.
                 caro go: 5.
                 caro turnRight: 90.
                 length := length + 10 ]
```

We can rewrite this script using the loop method `to:do:` as shown in the script 8.10.

Script 8.10 (Using `to:do:`)

```
| caro |
caro := Turtle new.
1 to: 10 do: [:length |
             caro go: length * 10.
             caro turnLeft: 90.
             caro go: 5.
             caro turnRight: 90]
```

If we look at the difference between the two scripts, we see that we do not need to declare the variable `length` in the script. However, we have to declare the variable inside the block, that is the sequence of messages we want to repeat. To declare a variable inside a block we prefix it with the character `:` and the variable declaration is terminated by the character `|`. Therefore the expression `[:length |` declares a variable named `length` in the block.

The block passed to the message `to:do:` and `to:by:do:` require an argument.

The other difference that we can see is that we do not have to initialize the variable nor increase explicitly its value using an expression such as `length := length + 10`. Indeed what the loop method does for us is that it will assign values from 1 to 10 to the variable of the block, here `length` each time the block will be repeated. Hence the variable `length` will get the values 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 and the block will be repeated 10 times with one of these values assigned to `length`. Note that a

loop does not have to start at 1 but can start at any number. The section ?? will present in detail these aspects.

With the loop method `to:do:` we were forced to multiply the length by 10 to get the distance that the turtle should move to draw the same stair. With the method `to:by:do:` we can avoid to that as this message allows one to specify the increment from one value to the other. In fact `to:do:` is equivalent to `to:by:1:do:`. The script 8.11 shows how the same method can be expressed. Note that here the values taken by the variable `length` are 10, 20, 30,... until 100 therefore we do not have to multiple `length` as in the other script. Notice also that we start at 10 and not 1 because we want that the first value taken by the variable is 10 and not 1.

Script 8.11 (Using `to:do:`)

```
| caro |
caro := Turtle new.
10 to: 100 by: 10 do: [:length |
    caro go: length.
    caro turnLeft: 90.
    caro go: 5.
    caro turnRight: 90]
```

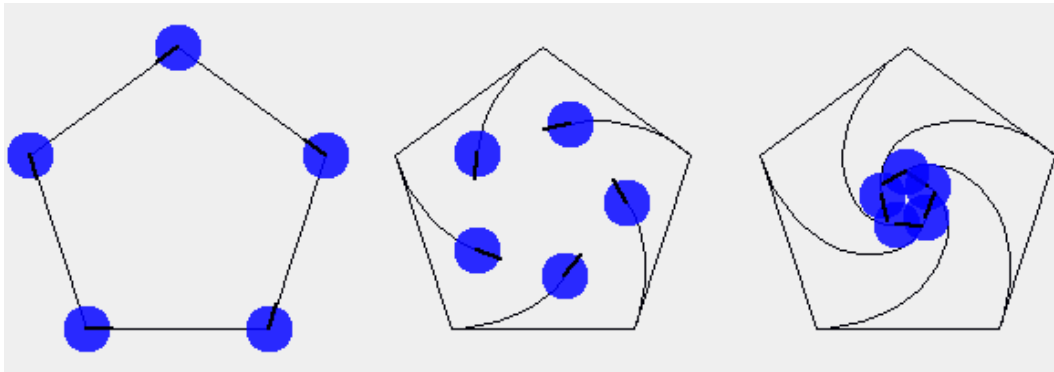
Block arguments are similar to method argument. `[:length | ...]` is a block defining an argument named `length`. `:length` is the argument name. As blocks may have several arguments, `|` ends the argument list.

Some technical details. The last argument of `to:do:` and `to:by:do:` is a block that is sequence of messages like for the `timesRepeat:` method. However, for `to:do:` and `to:by:do:` the block requires one argument. Block arguments are similar to method argument. `[:length | ...]` is a block defining an argument named `length`. `:length` is the argument name. As blocks may have several arguments, `|` ends the argument list. Do not give name to a block argument the name of a variable already existing in the script or method containing the block.

Summary

- The block passed to the message `to:do:` and `to:by:do:` require an argument.
- Block arguments are similar to method argument. `[:length | ...]` is a block defining an argument named `length`. `:length` is the argument name. As blocks may have several arguments, `|` ends the argument list.
- Do not give name to a block argument the name of a variable already existing in the script or method containing the block.

Fun with Collections



In this chapter we present three small problems that use collections. First we present a fractal drawing generator, then we present how to generate dot to dot picture. The third example shows how to generate the picture above.

1 Programming Fractal Drawings

We would like to generate drawings from simple descriptions. For this purpose we would like to have a way to describe the drawing and manipulate drawings. One solution is to define a small language that represents graphics and one way to interpret this graphical language.

Imagine a mini graphical language for the turtles composed by the symbols **F**, **f**, **+** and **-** and that the symbols have the following meaning:

- F** : the turtle goes forward and leaves a trace from a given distance. It corresponds to our `go`: method.
- f** : the turtle goes forward without leaving a trace from a given distance. It corresponds to our `jump`: method.
- +** : the turtle turns on the left from a given angle. It corresponds to our `turnLeft`: method.
- : the turtle turns on the right from a given angle. It corresponds to our `turnRight`: method.

A sequence of such symbols represents a drawing. We can ask a turtle to interpret long sequences such as the one shown in the script 9.1 to produce drawing such as the one shown in figure 9.1. Note that in a subsequent chapter we will show you how to produce these long sentences that are the results of LSystems.

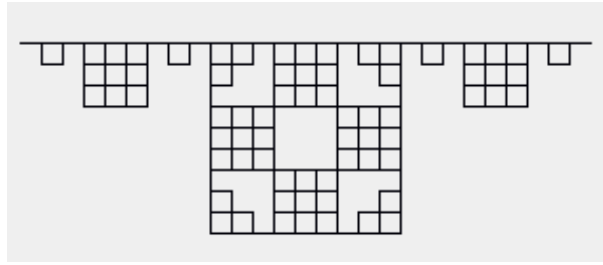


Figure 9.2: A Growing Fractal Line.

fixed size where the order is conserved. Note that we could have used a `String` which is a collection of characters. Do it by replacing `#$F $F $F` by `'FFF'`.

In chapter chapter ?? we define the method `interpret:` that given a character asks the receiver to execute the associated method. We slightly modified this method to associate the new symbols with the actions as shown in the method 9.1.

Method 9.1

interpret: aCharacter

```
"Turtle new interpret: $F"
```

```
aCharacter = $F
  ifTrue: [self go: 10]
  ifFalse: [aCharacter = $f
    ifTrue: [self jump: 10]
    ifFalse: [aCharacter = $+
      ifTrue: [self turnLeft: 90]
      ifFalse: [aCharacter = $-
        ifTrue: [self turnRight: 90]]]]]
```

Now what we need is to execute the method `interpret:` on each element of the collection. This is what the method `interpretSentence: anArray` does using the method `do:`.

Method 9.2

interpretSequence: aCollection

```
"Send the message interpret to all the elements of anArray"
```

```
aCollection do: [:aCharacter | self interpret: aCharacter]
```

2 Dot to Dot

Everybody plays once to the game to discovering a drawing based on number drawing a line from a number to another one. Let us have fun and develop a way to define the same functionality. The first think that we want to be able to do is to draw a picture from a list of points. If you want to know how we collected the points, here is how we proceed: we created an instance of the class `MemoryTurtle`,

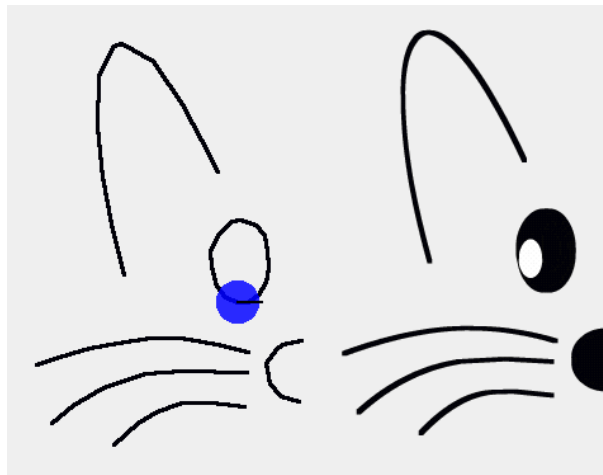


Figure 9.3: The copy and its Model

place the turtle on the desired position, send it the message `remember` and continue like that for a while. When we were satisfied we send it the message `points`.

To draw the picture shows in Figure 9.3 we used the script 9.2. Imagine how you would implement the methods `drawLineFromPoints`: that draws a line from a list of points and `drawClosedLineFromPoints`: that draws a closed path from a list of points joining the last and the first point.

Script 9.2 (*Drawing half of the mascotte*)

```
| c |
c := Turtle new.
c penSize: 3.
c drawLineFromPoints: {337@314 . 327@274 . 320@237 . 316@197 . 317@157 . 329@134 .
335@132 . 360@146 . 383@180 . 398@208 . 411@233}.
c drawLineFromPoints: {435@375 . 410@369 . 382@364 .358@364 .332@368 .310@372 .
286@379 .268@385}.
c drawLineFromPoints: {434@391 . 383@390 . 354@392 . 321@403 . 297@417 . 280@431}.
c drawLineFromPoints: {432@418 . 406@415 . 383@415 . 363@422 . 348@431 . 329@448}.
c drawLineFromPoints: {475@414 . 461@410 . 452@400 . 450@388 . 449@384 . 457@373 .
462@369 . 477@366}.
c drawClosedLineFromPoints: {427@336 . 417@332 . 410@323 . 406@308 . 405@295 .
412@282 . 422@272 . 429@271 . 441@275 . 448@283 . 450@295 . 451@308 . 449@320 .
443@330. 435@336}.

```

The method 9.3 proposes one way to draw a line from a collection of point. Then based on this method, the method 9.4 defines how to draw a close path from a collection of points. Note that we can also scale the image using the message `*` to a collection. The result we get is a collection whose points have been multiply by the factor we specify as shown in the script 9.3.



Figure 9.4: A small mascotte generated from half of the data.

Method 9.3

drawLineFromPoints: aCollection

```
aCollection size > 2  
  ifFalse: [^ self].  
self jumpAt: (aCollection at: 1).  
aCollection do: [:aPoint | self goAt: aPoint]
```

Method 9.4

drawClosedLineFromPoints: anArray

```
self drawLineFromPoints: anArray.  
self goAt: anArray first
```

Script 9.3 (*Scaling the mascotte*)

```
c := Turtle new.  
c penSize: 3.  
leftNose := {475@414 . 461@410 . 452@400 . 450@388 . 449@384 . 457@373 . 462@369 . 477@366} * 0.5.  
x := (leftnose at: 1) x.  
leftEar := {337@314 . 327@274 . 320@237 . 316@197 . 317@157 . 329@134 . 335@132 . 360@146 . 383@180 . 398@200} * 0.5.  
leftTopWhisker := {435@375 . 410@369 . 382@364 . 358@364 . 332@368 . 310@372 . 286@379 . 268@385} * 0.5.  
leftMiddleWhisker := {434@391 . 383@390 . 354@392 . 321@403 . 297@417 . 280@431} * 0.5.  
leftBottomWhisker := {432@418 . 406@415 . 383@415 . 363@422 . 348@431 . 329@448} * 0.5.  
leftEye := {427@336 . 417@332 . 410@323 . 406@308 . 405@295 . 407@290 . 412@282 . 422@272 . 429@271 . 441@270} * 0.5.  
  
c drawLineFromPoints: (leftEar * 0.5).  
c drawLineFromPoints: (lefttopWhisker * 0.5).  
c drawLineFromPoints: (leftmiddleWhisker * 0.5).  
c drawLineFromPoints: (leftbottomWhisker * 0.5).  
c drawLineFromPoints: (leftnose * 0.5).  
c drawClosedLineFromPoints: (lefteye * 0.5).
```

Now that we have half of our mascotte, we should be able to draw completely doing some simple vertical symmetry.

Script 9.4 (Complete small mascotte)

```

Turtle clearWorld.
c := Turtle new.
c penSize: 3.
leftNose := {475@414 . 461@410 . 452@400 . 450@388 . 449@384 . 457@373 . 462@369 . 477@366} * 0.5.
x := (leftNose at: 1) x.
leftEar := {337@314 . 327@274 . 320@237 . 316@197 . 317@157 . 329@134 . 335@132 . 360@146 . 383@180 . 398@200} * 0.5.
leftTopWhisker := {435@375 . 410@369 . 382@364 . 358@364 . 332@368 . 310@372 . 286@379 . 268@385} * 0.5.
leftMiddleWhisker := {434@391 . 383@390 . 354@392 . 321@403 . 297@417 . 280@431} * 0.5.
leftBottomWhisker := {432@418 . 406@415 . 383@415 . 363@422 . 348@431 . 329@448} * 0.5.
leftEye := {427@336 . 417@332 . 410@323 . 406@308 . 405@295 . 407@290 . 412@282 . 422@272 . 429@271 . 441@271} * 0.5.
rightEye := c verticalSymmetry: leftEye from: x.
rightEar := c verticalSymmetry: leftEar from: x.
rightNose := c verticalSymmetry: leftNose from: x.
rightTopWhisker := c verticalSymmetry: leftTopWhisker from: x.
rightMiddleWhisker := c verticalSymmetry: leftMiddleWhisker from: x.
rightBottomWhisker := c verticalSymmetry: leftBottomWhisker from: x.
c drawLineFromPoints: leftNose.
c drawLineFromPoints: leftEar.
c drawLineFromPoints: leftTopWhisker.
c drawLineFromPoints: leftMiddleWhisker.
c drawLineFromPoints: leftBottomWhisker.
c drawClosedLineFromPoints: leftEye.
c drawLineFromPoints: rightNose.
c drawLineFromPoints: rightEar.
c drawLineFromPoints: rightTopWhisker.
c drawLineFromPoints: rightMiddleWhisker.
c drawLineFromPoints: rightBottomWhisker.
c drawClosedLineFromPoints: rightEye.

```

Computing a vertical symmetrical point is based on the fact that the abscissa of the reference point is in the middle of the two symmetric points as shown in Figure 9.5. The idea is that the coordinates of the middle point are equal to summing the two points and divided by two. Therefore the symmetrical point is equal to the multiply by 2 the middle point and subtracting the first point. Mathematically said: $(x + x')/2 = x_{mid}$ therefore $x' = 2 * x_{mid} - x$. The method `verticalSymmetry` 9.5 presents a possible way to getting a collection containing all the symmetric points of the collection passed as argument.

Method 9.5**verticalSymmetry: aCollection from: anInteger**

"Return the collection of point symmetrical by reference to the abscissa anInteger"

```
^ aCollection collect: [:aPoint | 2 * anInteger - aPoint x @ aPoint y]
```

As you certainly noticed this is boring to have to repeat all the times the `drawLineFromPoints` invocation. Don't you think that you can find a solution? Why not using a collection of collection of points? The script 9.5 shows how we do it.

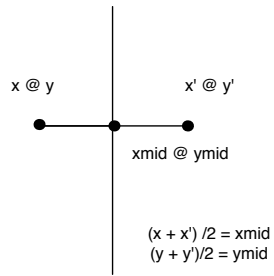


Figure 9.5: Mathematical relationship between a point and its vertical symmetrical point.

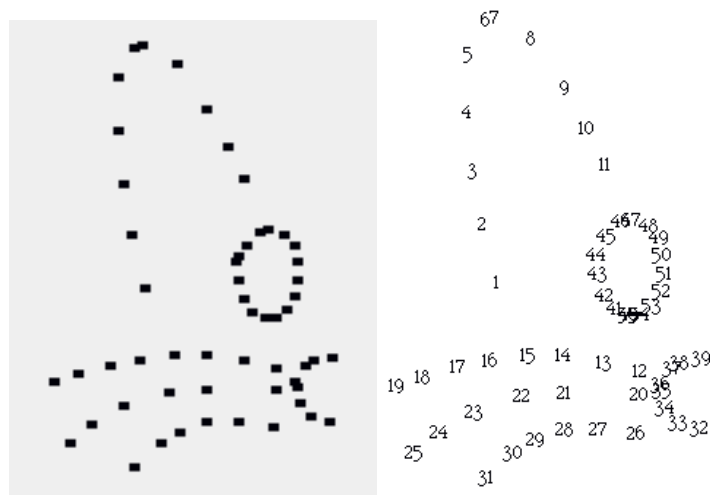


Figure 9.6: With dots or numbers

Script 9.5

```

...
{ leftNose . leftEar . leftTopWhisker . leftMiddleWhisker . leftBottomWhisker .
rightNose . rightEar . rightTopWhisker . rightMiddleWhisker . rightBottomWhisker}
do: [:aCol | c drawLineFromPoints: aCol ].
{leftEye. rightEye}
do: [:aCol | c drawClosedLineFromPoints: aCol].

```

Now we can for example draw dot instead of lines. Propose a solution to this problem as when in Figure 9.6. The methods `drawDotFromPoints: 9.6` and `doAt: 9.7` present a possible solution.

Method 9.6

`drawDotFromPoints: aCollection`

```
aCollection do: [:aPoint | self dotAt: aPoint]
```

Method 9.7**dotAt: aPoint**

```
"Make a dot without changing the position nor the direction of the receiver"
self jumpAt: aPoint.
self go: 1.
self turn: 180.
self go: 1.
self turn: 180
```

The script 9.6 shows how to display a number at the location 200@300.

Script 9.6 (*Displaying a number*)

```
StringMorph new contents: 10000 printString;
  center: 200@300;
  openInWorld
```

Now we create the method `number:at:` 9.8 which displays a given number at a point.

Method 9.8**number: aNumber at: aPoint**

```
"Make a dot without changing the position nor the direction of
the receiver"

self jumpAt: aPoint.
StringMorph new contents: aNumber printString;
  center: aPoint;
  openInWorld
```

To generate a dot to dot we have to take care that an integer is displayed, then that the next one with the following integer is displayed and so on for every parts of the drawing. Moreover, each part should start at a number where another one finished. Therefore the method drawing a collection, let us call it `drawStarting: aNumber fromPoints: aCollection`, has to start drawing number at the specified number and it also have to return the next number to draw so that other call can start at the right number. The script 9.7 shos for example how these methods would be invoked and how the number would be then passed from one to the other. The script 9.8 shows the same behavior but with the solution based on collection of point collection we proposed before. Here again a number is passed between the method invocations.

Script 9.7 (*Half dotted mascotte*)

```
...
aNumber := 1.
aNumber := c drawStarting: aNumber fromPoints: leftNose.
aNumber := c drawStarting: aNumber fromPoints: leftEar.
aNumber := c drawStarting: aNumber fromPoints: leftTopWhisker.
aNumber := c drawStarting: aNumber fromPoints: leftMiddleWhisker.
aNumber := c drawStarting: aNumber fromPoints: leftBottomWhisker.
aNumber := c drawClosedStarting: aNumber fromPoints: leftEye.
```

Script 9.8 (*Doted mascotte*)

```

...
aNumber := 1.
{ leftNose . leftEar . leftTopWhisker . leftMiddleWhisker . leftBottomWhisker .
rightNose . rightEar . rightTopWhisker . rightMiddleWhisker . rightBottomWhisker }
  do: [:aCol | aNumber := c drawStarting: aNumber fromPoints: aCol ].
{leftEye. rightEye}
  do: [:aCol | aNumber := c drawClosedStarting: aNumber fromPoints: aCol].

```

Imagine how you would implement the methods `drawStarting:fromPoints:` and `drawClosedStarting:fromPoints:` `anArray` before looking at our solutions 9.9 and 9.10.

Method 9.9**drawStarting: aNumber fromPoints: anArray**

```

| number |
number := aNumber.
self jumpAt: (anArray at: 1).
anArray
  do: [:aPoint |
    self number: number at: aPoint.
    number := number + 1].
^ number

```

Method 9.10**drawClosedStarting: aNumber fromPoints: anArray**

```

| number |
number := self drawStarting: aNumber fromPoints: anArray.
self goAt: anArray first.
self number: number at: anArray first.
number := number + 1.
^ number

```

3 Follow

As a third project we would like to be able to create the drawings shown at the beginning of this chapter. Such drawings are created by asking each turtle to move in direction of another one which is also moving in direction of another one and so on. As we want to have a symmetric drawing we place the turtle on a circle. For that we need a method that places a turtle at a given vertex of a regular polygon. We implemented the method `placeOnPolygon:atVertex:` 9.11. For example `caro placeOnPolygon: 5 atVertex: 2` asks a turtle to move to the second vertex of a pentagon.

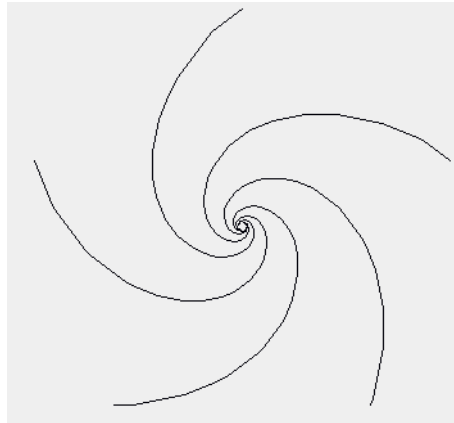


Figure 9.7: 5 turtles following each other.

Method 9.11

placeOnPolygon: aNumber atVertex: place

```
place timesRepeat:
  [self go: 200.
   self turn: 360 / aNumber]
```

Then we define the method `pointAtAndMove: aTurtle` that make the receiver pointing and moving a certain distance, here in method 9.12 5 pixels, in the direction of another turtle passed as argument.

Method 9.12

pointAtAndMove: aTurtle

```
self pointAt: aTurtle center.
self go: 5
```

Now we have to define some scripts to create several turtles, place them on polygon vertices, and make them moving in the direction of other turtle. As the scripts are long and that we would like to reuse them, we decided to implement them as *class* methods. For this you need to get a class browser on the `Turtle` (there is one in the **advanced** flap) and click on the class button. The method that you defined there are methods that are executed when a message is sent to the class `Turtle` itself and not a single turtle such as `caro`. You already used such a kind of messages without paying attention: for example `Turtle new` send the message `new` to the class `Turtle` itself. Note that to make clear that these scripts are defined on the class itself we use the notation `Turtle class`. However, when you are entering the method you should ignore this expression. To invoke these methods you just send a message to the class. Here is an example on how to invoke the method `createArrayOfTurtles`: `to Turtle createArrayOfTurtles: 6`.

We decompose the problem into simple task: first create a collection of turtles, then placing them

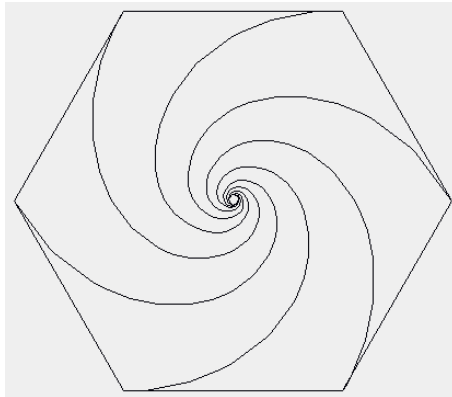


Figure 9.8: 6 turtles following each other.

on a regular polygon, then asking them to move. The first task is defined by the `createArrayOfTurtles: aNumber` as shown in 9.13. An array of size the number of turtles is created. Then the array is filled with turtles.

Method 9.13

```
Turtle class>>createArrayOfTurtles: aNumber
  "Turtle createArrayOfTurtles: 6"

  | anArray |
  anArray := Array new: aNumber.
  1
    to: aNumber
    do: [:i | anArray at: i put: Turtle new].
  ^ anArray
```

The second phase, that is placing the turtles on the vertices of the polygon is done by the method `placeTurtles: anArray` 9.14. The idea is that we iterate over the array and we ask each turtle to place itself that the current index of the array. The size of the array represents the number of edges and vertices the polygon has. Hence the second turtle will be placed on the second vertex. Note that we could have done both phases all in once but this would have been less readable.

Method 9.14

```
placeTurtles: anArray
  "Turtle placeTurtles: (Turtle createArrayOfTurtles: 6)"

  | edges |
  edges := anArray size.
  1
    to: anArray size
    do: [:i |
      (anArray at: i)
        placeOnPolygon: edges
        atVertex: i].
  ^ anArray
```

The final step is to ask the turtle to move. For this purpose we defined the method `moveOneStep:` (see 9.15). For this, we ask the first turtle to point on the second one and so on. However, we have to pay attention before for the last turtle such a practice would be an error. Therefore we should not iterate over all the collection but stop before the last one. Then we should not forget to ask the last turtle to move toward the first one.

Method 9.15

```
Turtle class>>moveOneStep: anArray
  "Make each turtle move a step forward in the direction of the next one.
  The last following the first one"

  1 to: anArray size -1 do:
    [:i | (anArray at: i) pointAtAndMove: (anArray at: i + 1)].
  anArray last pointAtAndMove: anArray first
```

Now we are ready to get our drawing done. The script 9.9 asks 80 iterations of a simple move. This approach is not satisfying because we have to guess the number of steps to complete the drawing. Propose a solution. The script 9.10 it moves the turtles until the first one is at the distance smaller than 1 of the last one.

Script 9.9 (Fixed number of Steps)

```
| anArray |
anArray := Turtle placeTurtles: (Turtle createArrayOfTurtles: 6).
80 timesRepeat: [self moveOneStep: anArray.]
```

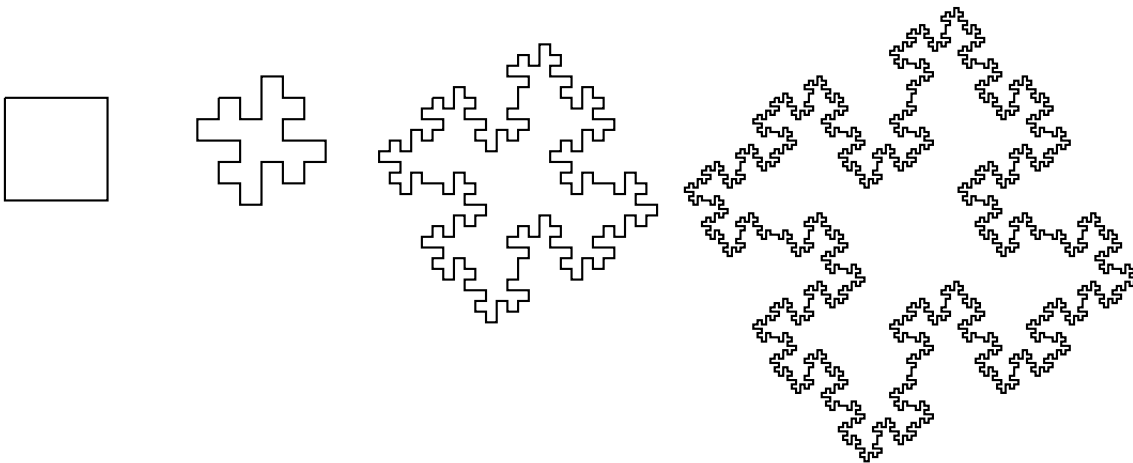
Script 9.10 (Number of Steps Resolved)

```
| anArray |
anArray := Turtle placeTurtles: (Turtle createArrayOfTurtles: 6).
[(anArray first center dist: anArray last center) < 1]
  whileFalse: [self moveOneStep: anArray.]
```

3.1 An Alternate Solution

In the solution we proposed, we have to take care that the last element of the collection is pointing to the first one. Hence we loop until the element before the last and have to make a special case for the last one. An alternate approach is to create a collection having one extra place at the end where we add a second time the first element. This solution allows one not to bother about the case of the last element. However, we have to take care we do not treat the last element because we would try to access an element out of the bounds of the collection as in the first solution. Try to come up with an implementation of this solution.

Simple L-Systems and Fractal Production



Aristide Lindermayer worked on the understanding and representation of plant growth. He invented a way to describe how simple plants and algae grow. This approach is named L-Systems. Although L-Systems are used to describe the plant growth, they can also be applied to produce fractal graphics – graphics based on repetitions of themselves, as the ones above. Moreover, the graphical interpretation of L-Systems is easy with a turtle. This is what we explore in this chapter. In this chapter, we propose you to implement the simplest form of L-System.

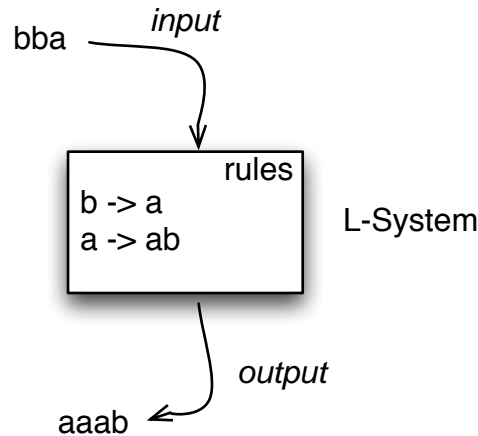
1 L-Systems

L-Systems is a generic term of designing different rule-based rewriting systems. A rewriting rule-based system is a system that given an *input* like a sequence of characters, strings, or numbers and the *set of rules* explaining how an element of the input is replaced by new ones, produces an *output*.

Let us look at the simple example¹ below.

Fibonacci

¹A fun aspect of this L-System is that the length of the production is the Fibonacci suite.



Rule 1: $b \rightarrow a$
 Rule 2: $a \rightarrow ab$

It is composed by two rules, the first means that every b has to be replaced by an a , the second means that every a has to be replaced by ab . Now if we start with b and apply the two rules we obtain the following results where each line represents the application of the rules to the result of the previous line.

| | |
|-----------------------|------------------------------|
| <i>Input</i> | b |
| (Rule 1) | $\rightarrow a$ |
| (Rule 2) | $\rightarrow ab$ |
| (Rules 2, 1) | $\rightarrow ab a$ |
| (Rules 2, 1, 2) | $\rightarrow ab a a b$ |
| (Rules 2, 1, 2, 2, 1) | $\rightarrow ab a a b a b a$ |

We call the first *input*, in the example b , the *original input* of the L-System. Note that we say that we *apply* a rule. We call the symbols a and b that constitutes the first input and output the vocabulary of the L-System.

Analysis. Let us analyse this simple L-System² and understand the basic mechanisms used here:

- An L-System can contain several rules. Here we have two rules namely Rule 1 that transforms every element b into a and Rule 2 that transforms every a into the sequence of elements ab .
- A rule is composed by two parts: a *left* and a *right* part. The left part identifies the element in the input that will be replaced by the right part. The left part of a rule only identifies *one* element while the right part can be *one or multiple* elements.
- *Applying a rule* means replacing in the current input the element (left part of the rule) by the sequence of elements described in the right part of the rule. For example, the third line above is produced by applying the Rule 1 and substituting a by ab .

²Certain L-Systems are much more complex and take into account the context of the element or some other parameters before applying the rule.

- The rules are applied each time they can be applied. For example, the production of the fourth line is the result of applying the Rule 2 and the Rule 1 to the sequence *ab*. Note that the application order is irrelevant as long as it is performed in parallel that is that a rule is only applied on the input and not on the result of another rule application.
- A rule can be applied several times. The production of the last line is the result of the application of the rules 2, 1, 2, 2 and 1.
- The input does not have to be necessary one single element.

2 The Graphical Interpretation of L-Systems

Besides representing plant growth, L-Systems are also interesting systems because we can use them to produce graphics representing plants or fractals. Fractals are graphics that are built using repetitive patterns and that also partially or fully contain themselves. The idea to produce such graphics based on L-Systems is to use L-System whose vocabulary is mapped to turtle actions.

2.1 A Turtle Dialect for L-Systems

To produce graphics we chose to define the vocabulary of the L-Systems the following way: input and rules only manipulate a limited set of predefined symbols representing turtle methods. These symbols are:

- F** : the turtle goes forward and leaves a trace from a given distance. It corresponds to our `go`: method.
- f** : the turtle goes forward without leaving a trace from a given distance. It corresponds to our `jump`: method.
- +** : the turtle turns on the left from a given angle. It corresponds to our `turnLeft`: method.
- : the turtle turns on the right from a given angle. It corresponds to our `turnRight`: method.

This simplest form of L-System are based on an angle and a length to move forward that are defined *once* for all the rules of the L-System. The value of these data does not change during the rules application or graphical interpretation. This is the reason why you have to pay attention of the length we define if we want to have reasonable output.

2.2 A First L-System based on Turtle Actions

Let us look at the following example whose first application steps are illustrated by Figure 10.1.

Figure 10.1

| | |
|--------------|-------------------------------|
| <i>Input</i> | <i>F</i> |
| <i>Angle</i> | 90 degree |
| <i>Rule</i> | $F \rightarrow F+F-FFF+F+F-F$ |

- The input represents the geometrical shape that we take as starting point. Here *F* means that we start with a line, *F+F+F+F* a square as the angle is 90 degree.
- The rule describes how one segment is transformed in the second steps of Figure 10.1.

Script 10.1

```
|aTurtle|
aTurtle := Turtle new.
aTurtle interpretChar: $F length: 100 angle: 90.
aTurtle interpretChar: $+ length: 100 angle: 90.
aTurtle interpretChar: $F length: 100 angle: 90.
aTurtle interpretChar: $+ length: 100 angle: 90.
aTurtle interpretChar: $f length: 100 angle: 90.
aTurtle interpretChar: $+ length: 100 angle: 90.
aTurtle interpretChar: $F length: 100 angle: 90.
```

Method 10.1

In category L-System

```
interpretChar: aChar length: len angle: degree
```

```
aChar = $F
  ifTrue: [self go: len]
  ifFalse: [aChar = $+
    ifTrue: [self turnLeft: degree]
    ifFalse: [aChar = $-
      ifTrue: [self turnRight: degree]
      ifFalse: [aChar = $f
        ifTrue: [self jump: len]]]]
```

Now we want to treat *sequences* of characters and not characters one by one. Therefore define the method `interpret: aCollection length: len angle: degree` that knows how to interpret a string, that is a collection of characters. Hints: `String` is a subclass of `Collection` so operations available on collections such as `do:` are also possible on strings. The Script 10.2 should also draw an U.

Script 10.2

```
|aTurtle|
aTurtle := Turtle new.
aTurtle interpret: 'F+F+f+F' length: 100 angle: 90.
```

A possible solution is shown in method 10.2

Method 10.2

In category L-System

```
interpret: aString length: len angle: degree
```

```
aString do: [:each | self interpretChar: each length: len angle: degree]
```

Now we are ready to implement the application of rules on an input.

4 L-Systems with a Single Loops

For now we limit our approach to the simplest L-System: a system containing only one rule. We took this decision so that you could have a fast idea of the process. What is missing now is the application of the rule on an input. We want to be able to specify an input, the rule to apply, a number of times the rule should be applied. Define the method `applyOnInput: aString leftPart: leftPart rightPart: rightPart level: n`. Just remember that computing the input a certain number of time requires to apply any possible rules then reiterate the rule application on the result you obtained previously. To help you, the method `copyReplaceAll:with:` defined on the class `SequenceCollection` replaces all the occurrences of a string by another in the receiver as shown by the script 10.3.

Script 10.3

```
'How now brown cow?' copyReplaceAll: 'ow' with: 'ello'
—Printing the returned value: 'Hello nello brellon cello?'
```

Script 10.4

```
|aTurtle|
aTurtle := Turtle new.
aTurtle applyOnInput: 'F' leftPart: 'F' rightPart: 'F+F-F-FF+F+F-F' level: 1
—Printing the returned value: 'F+F-F-FF+F+F-F'
aTurtle applyOnInput: 'F' leftPart: 'F' rightPart: 'F+F-F-FF+F+F-F' level: 0
—Printing the returned value: 'F'
aTurtle applyOnInput: 'F' leftPart: 'F' rightPart: 'F+F-F-FF+F+F-F' level: 2
—Printing the returned value: 'F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F-F+F-F-
FF+F+F-FF+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F'
```

Here is a possible implementation of the method `applyOnInput:leftPart:rightPart:level:` is shown in the method definition 10.3.

Method 10.3

In category L-System

```
applyOnInput: input leftPart: start rightPart: repl level: n
"Generate the nth output of a simple L-System composed of one rule
replacing leftPart by rightPart in input"
```

```
|production|
production := input.
n timesRepeat: [production := self
                 replace: start
                 by: repl
                 in: production].
^ production
```

Method 10.4

```
replace: aCharacter by: aCollection in: aSequence
^ aSequence copyReplaceAll: aCharacter with: aCollection
```

Now you are in the position to produce your first graphics. You have to use the methods `applyOn-Input:leftPart:rightPart:level:` and `interpret:length:angle:.`. Try to program the L-System we show previously.

Figure 10.1

| | |
|--------------|--------------------------------|
| <i>Input</i> | F |
| <i>Angle</i> | 90 degree |
| <i>Rule</i> | $F \rightarrow F+F-F-FF+F+F-F$ |

5 A Gallery of 1-rule based L-Systems

We show you some of the pictures that you can now create. The first pictures of this chapter are generated by the following L-System. This family of curves are called Koch curves. The other pictures of this chapter are generated using the described L-Systems. Note that some of them may take some time to draw.

First Pictures of the Chapter

| | |
|--------------|--------------------------------|
| <i>Input</i> | $F-F-F-F$ |
| <i>Angle</i> | 90 degree |
| <i>Rule</i> | $F \rightarrow F-F+F+FF-F-F+F$ |

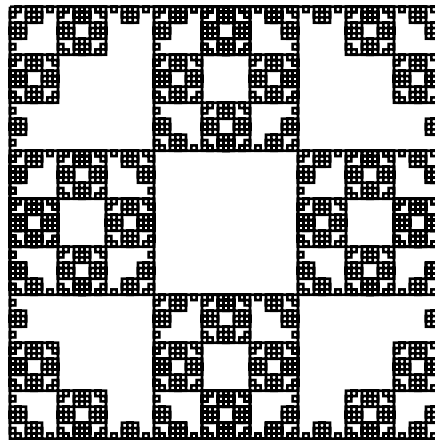
Figure 10.3: Variation on Koch curve 2: $n = 4$, angle = 90, Input: $F-F-F-F$, $F \rightarrow FF-F-F-F-FF$

Figure 10.3

| | |
|--------------|-----------------------------|
| <i>Input</i> | $F-F-F-F$ |
| <i>Angle</i> | 90 degree |
| <i>Rule</i> | $F \rightarrow FF-F-F-F-FF$ |

Figure 10.4

| | |
|--------------|------------------------------|
| <i>Input</i> | $F-F-F-F$ |
| <i>Angle</i> | 90 degree |
| <i>Rule</i> | $F \rightarrow FF-F-F-F-F+F$ |

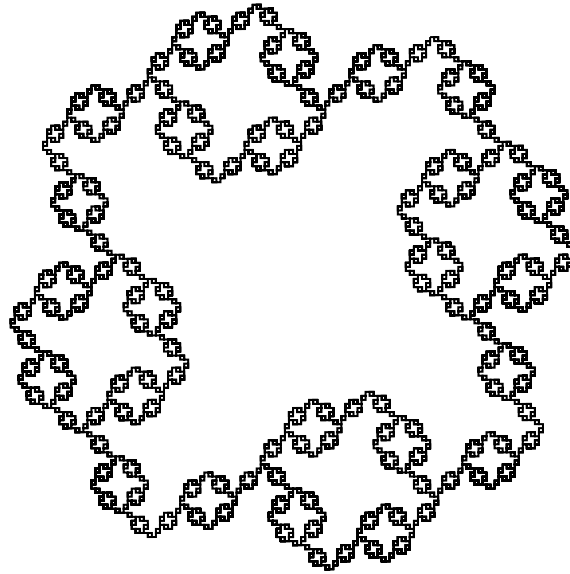


Figure 10.4: Variation on Koch curve 1: $n = 4$, angle = 90, Input: $F-F-F-F, F \rightarrow FF-F-F-F-F+F$

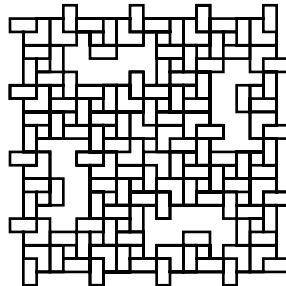


Figure 10.5: Variation on Koch curve 3: $n = 3$, angle = 90, Input: $F-F-F-F, F \rightarrow FF-F-F+F-F-FF$

Figure 10.5

Input $'F-F-F-F'$
Angle 90 degree
Rule $F \rightarrow FF-F+F-F-FF$

Figure 10.6 with $n=4$ and 10.7 with $n=5$

Input $F-F-F-F$
Angle 90 degree
Rule $F \rightarrow FF-F- -F-F$

Figure 10.8

Input $F-F-F-F$
Angle 90 degree
Rule $F \rightarrow F-F+F-F-F$

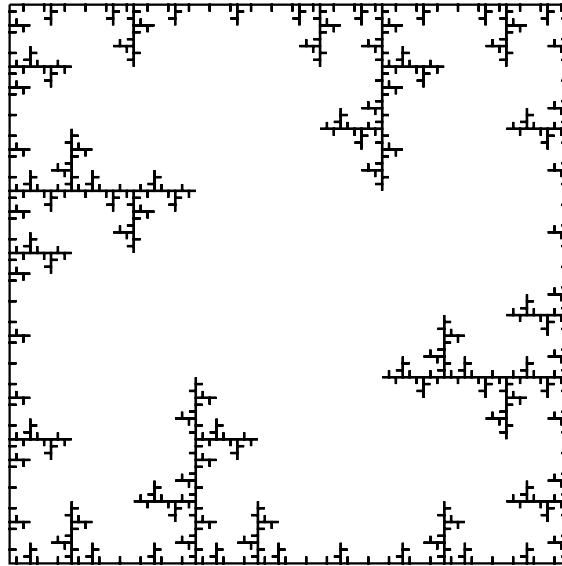


Figure 10.6: Variation on Koch curve 4: $n = 4$, angle = 90, Input: $F-F-F-F, F \rightarrow FF-F- -F-F$

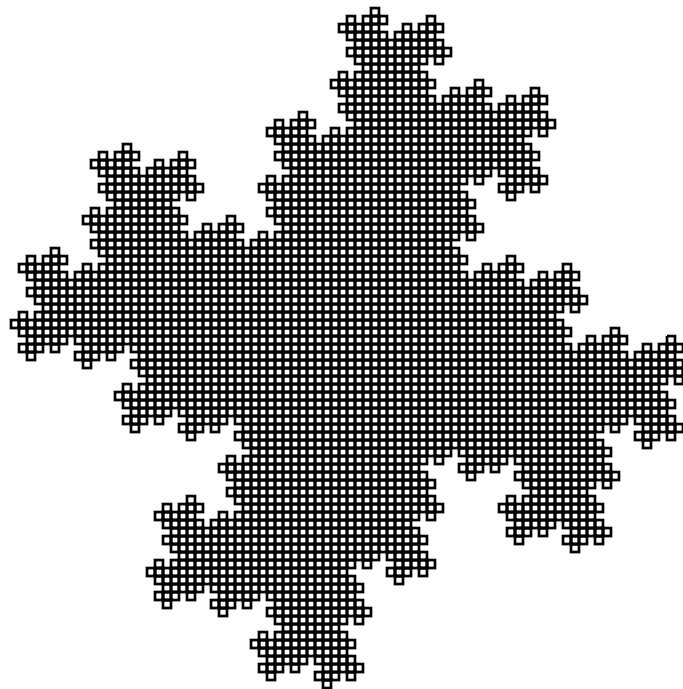


Figure 10.7: Variation on Koch curve 5: $n = 5$, angle = 90, Input: $F-F-F-F, F \rightarrow F-FF-F-F$

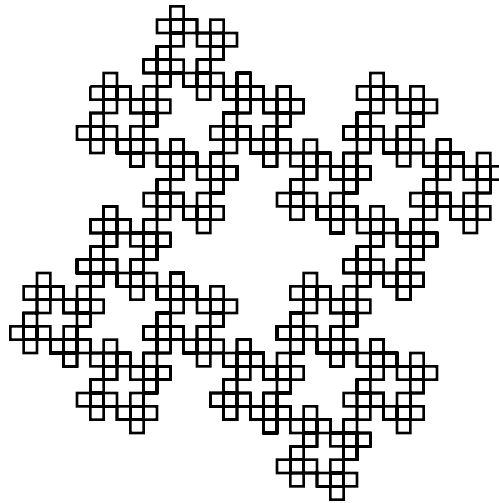


Figure 10.8: Variation on Koch curve 6: $n = 4$, angle = 90, Input: $F-F-F-F$, $F \rightarrow F-F+F-F-F$

6 Experimentation

We suggest you to try the following L-Systems that we only describe here then try to create the L-System that produce the snow flakes shown in Figure 10.9.

Interesting at level 2

Input $F-F-F-F$
Angle 90 degree
Rule $F \rightarrow F+FF-FF-F-F+F+FF-F-F+F+FF+FF-F$

Interesting at level 3

Input F
Angle 90 degree
Rule $F \rightarrow F+F-F-F+F$

Experiment 10.1

Experiment and find the L-System generating the previous figures. Hints: start with the smallest interesting input, the smallest level of derivation and play with the production rule to generate the transformation of a side.

7 Analysis of the Solution

The fact that the solution only allows the expression of single rule L-System is not really a problem because we learnt this way the essence of the problem and are now ready to develop a more complex system. From a methodological point of view this is a good point. Some people think that producing the best solution handling all kinds of variations is better, however often we cannot foresee where the complex part will be and what are the possible variation. So having a small implementation up front

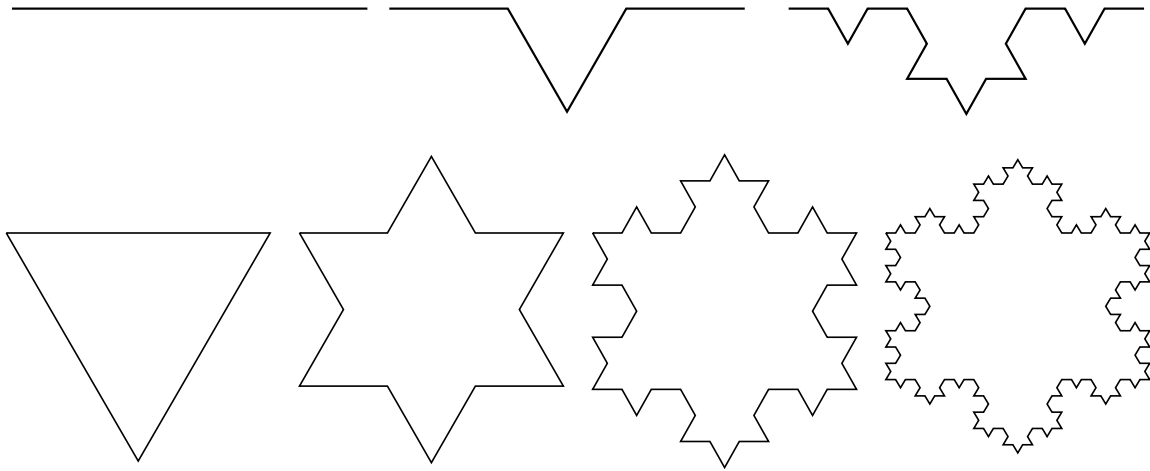


Figure 10.9: Steps to create a nice snow flake.

working even in a limited setting is a way to understand the problem. We should only be prepared to change it heavily or to simply throw away what we made.

Even if the solution we proposed here is successful to manage trivial L-Systems, its design is not really good from an object-oriented perspective. As this is not the topic of this book this is not really a problem. However, we just want to tell you why. In fact methods should represent the behavior or responsibilities that an object is carrying. Here a method such as `interpret:length:angle:` clearly represents the behavior of a turtle. This method allow one to represent messages sent in another form, here strings. But this is not the case for the methods `replace:by:in:` and `applyOnInput:leftPart:rightPart:level:` that have nothing to do with a turtle. Second these methods do not even use the state of the turtle or its methods. This is a sign that they are defined in the wrong place. A good solution would define a class representing an L-System and these methods would be moved there but this is the topic of another book.