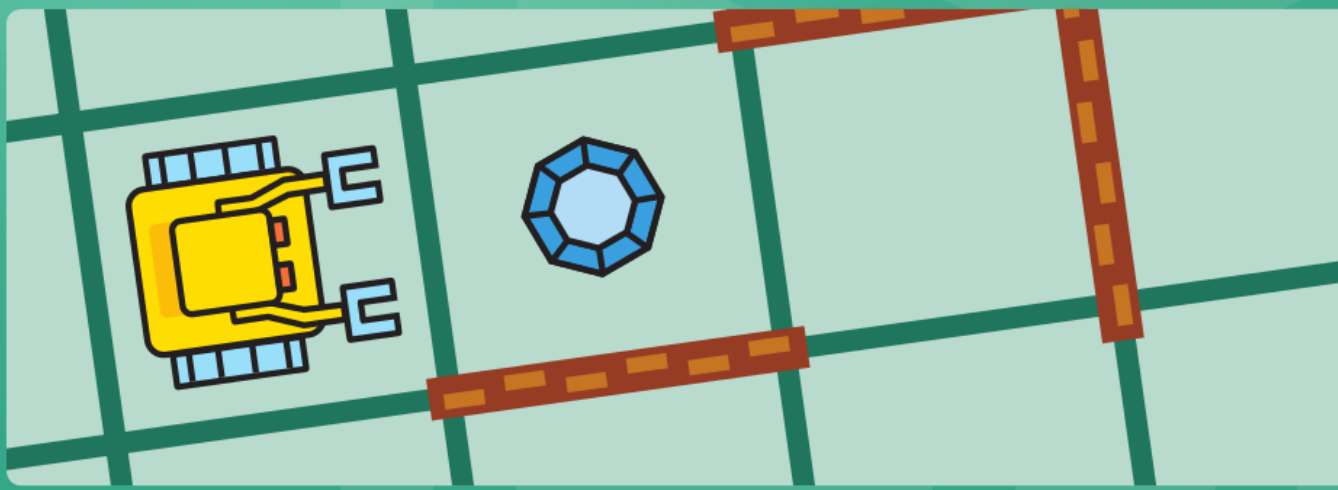
nclab
Online STEM Laboratory

First Course in **Programming with** Karel the Robot



Revision Nov-22-2012

About this Textbook

This free textbook is provided as a courtesy to NCLab users. It will help you develop *algorithmic thinking* which is the most challenging skill to acquire when learning computer programming. Once you are able to "think like a computer", learning any computer language is almost effortless. Karel is a famous educational language created and still used at the Stanford University. After taking this course, you will be able to transition swiftly to Python and other modern programming languages.

Become a Co-Author

We do not plan to publish the textbook with a commercial publisher since this would make it unnecessarily expensive for kids and students who are the main target audience. Feel free to contribute to the textbook with any material or suggestions. There is never enough illustrations and exercises, and there always are bugs to report. Translating the textbook into other languages would benefit thousands of kids worldwide. Instructions for contributors can be found below.

How to Contribute (for LaTeX and Git users)

The textbook is written in LaTeX, a high-quality typesetting system that you can learn and use in NCLab. In the future it will be possible to contribute to the textbook directly in NCLab, but at this time, the sources are stored in a public Git repository `nclab-textbook-karel` at Github (http://github.com).

How to Contribute (for all others)

We will gladly accept new interesting exercises for Karel and/or Python, as well as new images of good quality that will make the textbook more interesting and fun. You can send those at any time via email to `pavel@femhub.com`.

List of Contributors

- Pavel Solin, University of Nevada, Reno (primary author).
- Martin Novak, Czech Technical University, Prague, Czech Republic.
- Salih Dede, Coral Academy of Science High School, Reno, NV.
- Nazhmiddin Shapoatov, Sonoran Science Academy, Phoenix, AZ.
- Veronica McVey, Winnemucca Junior High School, NV.
- Jozsef Hollosi, Westfield, NJ.

Graphics Design: TR-Design `http://tr-design.cz`

Table of Contents

II Programming Exercises

III Review Questions

Foreword

This course provides an efficient introduction to modern algorithmic design and computer programming using the legendary educational programming language *Karel the Robot*. The language was first introduced by Richard E. Pattis at Stanford University in his famous book *Karel the Robot: A Gentle Introduction to the Art of Programming*.

With Karel, learning feels more like playing. You will not be exposed to discouraging technical complications of conventional programming languages or mathematics, and in no time you will get used to "thinking like a computer". This essential skill, otherwise known as *algorithmic thinking*, is much more important for computer programming than knowledge of any particular programming language.

We have seen many students struggle with computer programming and we know that there is absolutely no need for that. Often the only problem is the widely adopted mistake of instructors who use a powerful mainstream programming language such as Java, C or even C++ for a first programming course. This, however, is similar to teaching someone how to ride a motorcycle without teaching him or her how to ride a bicycle first. While there is nothing wrong with motorcycles, a beginner needs to learn to keep the balance first, and for that the motorcycle simply may not be the best tool.

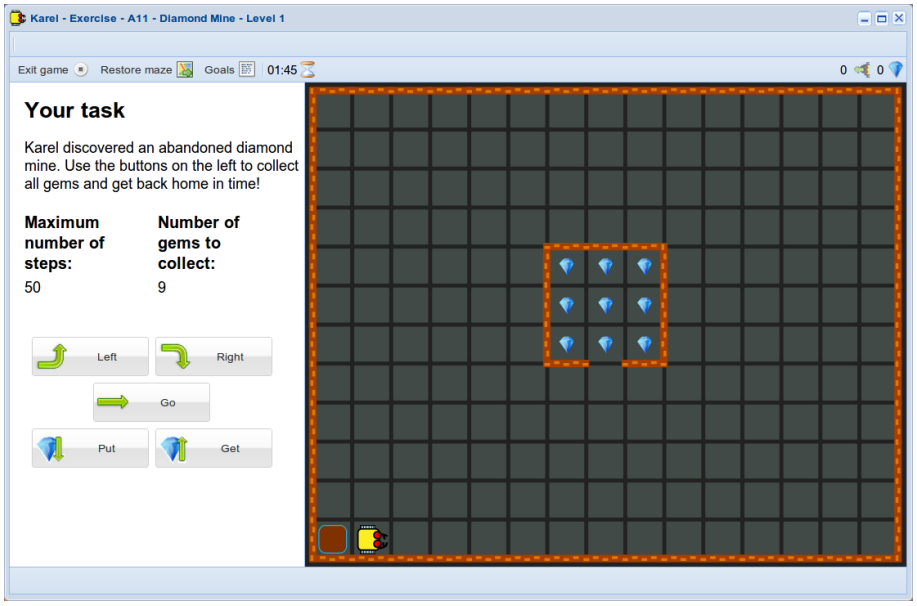


Fig. 1: Sample game *Diamond Mine* in Manual mode.

Learning to ride a bicycle before riding a motorbike is exactly what Karel the Robot is about. The robot only knows a handful of simple commands and has a few sensors

to navigate through the maze. There is no mathematics or confusing technicalities of conventional programming languages to worry about.

The course starts out in Manual mode (Level 0) where the robot can be guided via clicking on five buttons *Go* (make one step forward), *Left* (turn left), *Right* (turn right), *Put* (put a gem on the groud) and *Get* (pick up a gem from the ground). In the next level which is called *Bridge to Programming* students keep solving problems by typing the commands `go`, `left`, `right`, `put` and `get` instead of clicking on buttons.

The need for higher functionality such as loops, conditions, and custom commands arises naturally as game goals become more complicated. Students learn quickly that it is advantageous to break complex tasks into smaller ones, which is one of the most important principle of computer programming. The textbook is written by programming experts, and in addition to advanced programming skills the students gain an overview of good and bad programming habits.

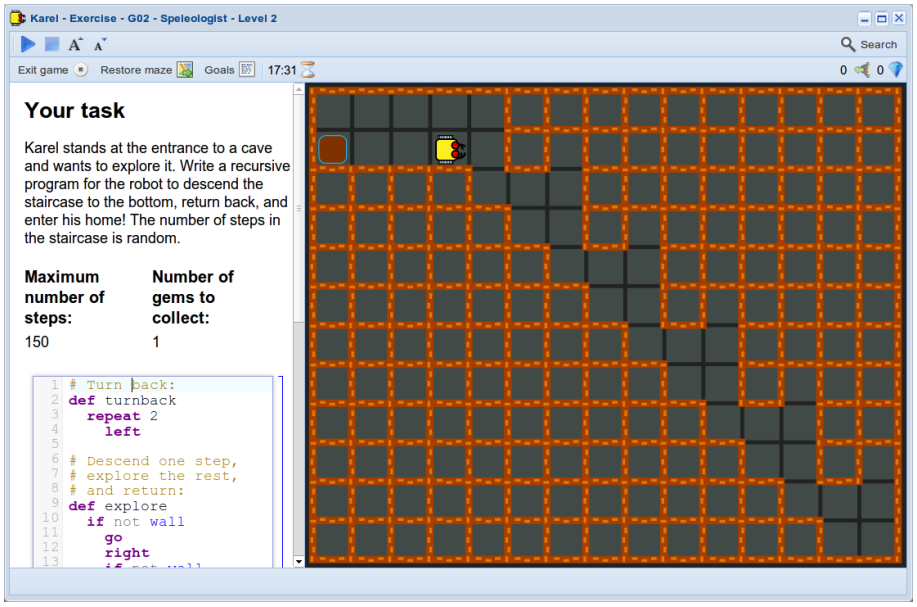


Fig. 2: Sample game *Speleologist* in Programming mode.

The syntax of Karel the Robot is very close to Python, a modern high-level dynamic programming language that is widely used in science, engineering, business and other areas today. In fact Python feels like Karel's older brother – the transition from Karel to Python is as seamless as the transition from one Karel's Level to another.

Part I

Meet Karel

1 Introduction

1.1 Objectives

- Learn basic facts about the Karel language and its history.
- Learn how Karel differs from other programming languages.
- Learn what skills this course will teach you.

1.2 Brief history

The educational programming language Karel the Robot was introduced by Richard E. Pattis in his book *Karel The Robot: A Gentle Introduction to the Art of Programming* in 1981. Pattis first used the language in his courses at Stanford University, and nowadays Karel is used at countless schools in the world. The language is named after Karel Čapek, a Czech writer who invented the word "robot" in his 1921 science fiction play R.U.R. (Rossum's Universal Robots). Various implementations of the language can be downloaded from the web, as shown in Fig. 3.

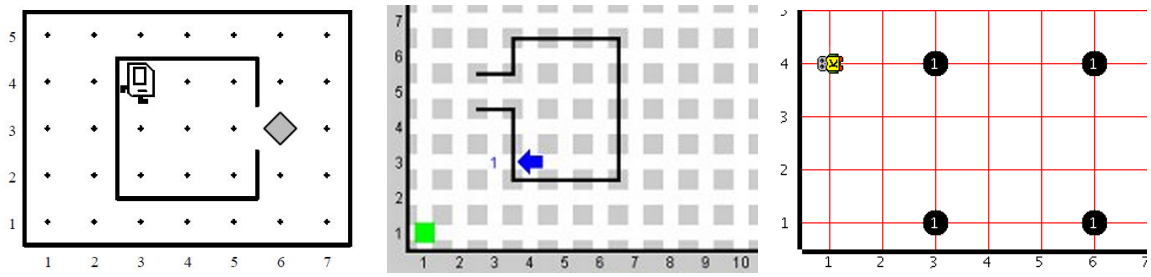


Fig. 3: Various implementations of Karel that can be found on the web.

The original Karel language was strongly influenced by Pascal, a popular language of the 1980s. Since Pascal is no longer being used today, we refreshed the language and changed it to be close to Python, a modern high-level dynamic programming language that is widely used in science, engineering, and business applications. Our changes made the language much easier to use. For illustration, compare the original Karel program

```
BEGINNING-OF-PROGRAM

DEFINE turnright AS
BEGIN
    turnleft
    turnleft
    turnleft
END

BEGINNING-OF-EXECUTION
    ITERATE 3 TIMES
    BEGIN
        turnright
        move
    END
    turnoff
END-OF-EXECUTION

END-OF-PROGRAM
```

with its exact NCLab equivalent

```
def turnright
    repeat 3
        left

repeat 3
    turnright
    go
```

In fact, NCLab' Karel has a built-in command `right` for the right turn, so the above program can be written using just three lines:

```
repeat 3
    right
    go
```

The new command was added after careful consideration because it makes Karel's motion easier to watch (in more complicated mazes he resembled a raging tornado). Of

course, orthodox Karel fans can still define and use their own `rightturn` command. We made a few additional cosmetic changes to the language in order to make it more accessible to kids – for example, beepers were replaced with gems, Karel has a home in the maze, and longer commands were replaced with simpler ones, such as `leftturn` with `left`, `move` with `go`, `pickbeeper` with `get` and `putbeeper` with `put`.

NCLab's Karel has a Manual Mode and a Programming Mode with three levels. While Levels 1 and 2 exactly correspond to the original Pattis' book, Level 3 contains additional material. It introduces variables, lists, functions returning values, and a GPS device for the robot. Hence Karel can store his path, and experiment with entry-level Artificial Intelligence (AI) algorithms. In the rest of this textbook, when we say "Karel" we always mean "NCLab's Karel".

1.3 Who is Karel?

Karel is a little robot that lives in a maze and loves to collect gems. He was manufactured with only five simple commands in his memory:

- `go` ... make one step forward.
- `get` ... pick up a gem from the ground.
- `left` ... turn to the left.
- `right` ... turn to the right.
- `put` ... put a gem on the ground.

He also has five built-in sensors that allow him to check his immediate surroundings:

- `wall` ... helps the robot detect that a wall is right ahead.
- `gem` ... helps the robot detect that a gem lies under him.
- `north` ... helps the robot detect that he is facing North.
- `home` ... helps the robot detect that he is at home.
- `empty` ... helps the robot detect that his bag with gems is empty.

1.4 What will this course teach you?

Computer programming skills are highly valued today, and they will be even more valued in the future. Karel is the perfect language for beginners. It will teach you how to design algorithms and write working computer programs without struggling with technical complications of mainstream programming languages. Thanks to its simplicity, you should be done with Karel fairly quickly, and in no time you will be ready to move on to other programming languages. In particular, NCLab offers Python programming: Textbook with exercises and review questions is available at `http://femhub.com/textbook-python`). After finishing Karel, you will be able to transition to Python smoothly.

1.5 Is Karel a toy language?

No, definitely not. Despite its playful appearance, Karel features all key concepts of modern procedural programming. Computer programming includes two fairly independent tasks – first to *design an algorithm* (sequence of steps leading to the solution of the problem at hand), and second, to *translate the algorithm into a suitable programming language*. The former skill is by far more important and therefore Karel puts maximum emphasis on it. This is why the language itself is kept so simple. As a matter of fact, the complexity of algorithms that you will encounter in this textbook ranges from trivial to extremely tough. Towards the end of the course you will encounter exercises which you will probably not call toy problems.

1.6 How does Karel differ from other programming languages?

The biggest conceptual difference between Karel and mainstream procedural programming languages such as Python, C, C++, Java or Fortran is that *the robot does not know math*. Math is not present in the first two levels that correspond to the Pattis' book, and it is strongly suppressed in the advanced Level 3 as well. This is done for a reason – math brings complications of its own which are unrelated to programming. In fact, many programming courses are designed in such a way that students are struggling more with math than with programming. That's not good. Math (except for logic perhaps) is not needed to understand how to design great algorithms and to translate them into efficient computer programs.

2 Launching Karel

2.1 Objectives

- Learn to launch Karel and work with the graphical application.
- Learn that Karel has several modes and how they differ.

The simplest way to launch Karel is to click on the icon *Programming* and select *Karel* in the menu. This will launch the application in *Programming mode* with a randomly generated maze, as shown in Fig. 4.

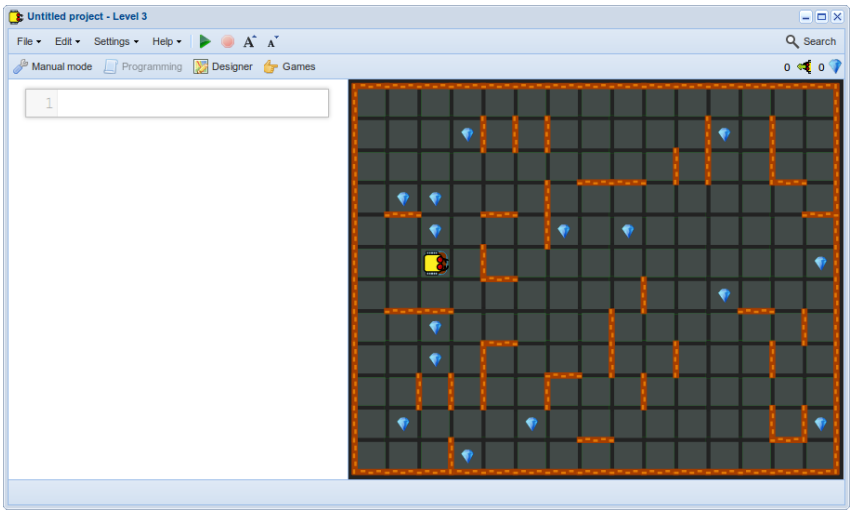


Fig. 4: Launching Karel in Programming mode, with a random maze.

The Programming mode is the most frequently used one. From there one can switch to *Manual mode*, *Designer mode*, and *Game mode* in the main menu. These modes will be discussed in more detail Paragraph 2.2.

The application window contains the main menu on top, work area on the left, maze on the right, and status bar on the bottom. The menus are fairly intuitive, so let us explain just a few selected functions. In the *File* menu:

- *Open* will open a Karel file that you created and saved, or cloned previously.
- *Clone* will help you search and download public Karel projects from the database.
- *Add static URL* will create a HTML address for your project. Such a link can be included in any web page.

The *Maze* menu facilitates operating with mazes, including creating a new random one, duplicating an existing maze, restoring maze to its saved version, and save and remove maze. The *Edit* menu enables operation with code cells and HTML cells (to be discussed in Section 6). In *Settings* one can change Karel's Level, change his speed, and adjust sound preferences.

The green and red buttons are used to run and stop programs, respectively, and the two buttons next to them on the right can be used to increase and decrease font size. The pair of icons on far right is the step counter (that can be reset by clicking on it) and gem counter that indicates how many gems Karel has in his bag.

2.2 Karel modes

Karel operates in four modes:

– *Manual mode:* The robot is controlled using the mouse and five buttons Go, Get, Left, Right, and Put. Watch out and do not crash!
– *Programming mode:* The robot is controlled using written programs (computer code). This mode has three Levels:
  - Level 1 serves as transition layer between the Manual and Programming modes. Programs are written using only five commands `go`, `get`, `left`, `right`, and `put` that exactly correspond to the buttons Go, Get, Left, Right, and Put in Manual mode.
  - Level 2 is where the actual programming begins. On top of the commands from Level 1, programs can contain conditions, loops, and custom commands.
  - Level 3 goes beyond the original Pattis' book by introducing variables, lists, and functions that return values. In this level Karel also gets a GPS device that allows him to determine his position in the maze. As we already mentioned, the functionality of Level 3 can be used to design entry-level artificial intelligence (AI) algorithms.
– *Designer mode* allows the user to create custom mazes.
– *Game mode* makes it possible to create and play games.

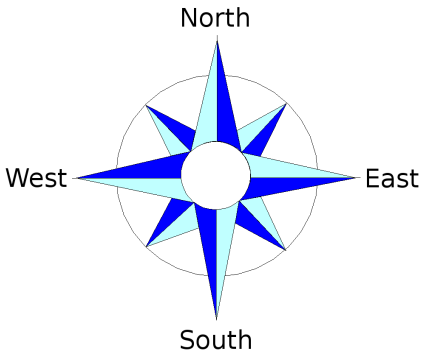
3 Manual Mode

3.1 Objectives

– Learn to operate the robot in Manual mode.

3.2 Compass

Before we begin, let us review the four directions on the compass:



3.3 Control buttons

When launching Karel through the Programming icon, switch to Manual mode using the corresponding menu button. Then, five buttons Left, Right, Go, Get and Put will appear in the panel on the left, as shown in Fig. 5.

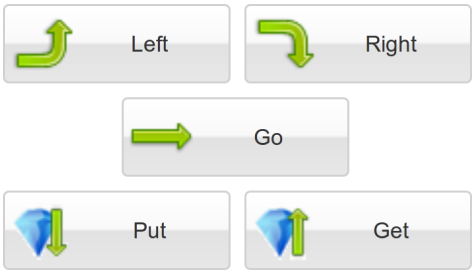


Fig. 5: Karel's buttons in Manual mode (robot facing East).

The function of the buttons is self-explanatory – pressing Left will turn the robot 90 degrees to the left, pressing Right will turn him 90 degrees to the right, and pressing Go will move him one step forward. Upon pressing Put the robot will reach into his bag with gems, take one, and put it on the ground where he stands. An indicator showing how many gems are in the bag can be found in the upper right corner of the window. Last, upon pressing Get the robot will pick up a gem from the ground where he stands. If one asks the robot to get a gem where there is none, he will complain.

When the robot turns, the arrows on the buttons adjust automatically to his new direction. This is illustrated in Fig. 6.

Fig. 6: Karel's buttons in Manual mode (robot facing West).

4 Programming Mode

4.1 Objectives

- Start operating the robot in Programming mode.
- Understand the difference between *algorithm* and *program*.
- Learn the difference between *logical* and *syntax* errors.
- Understand that *debugging* is an indivisible part of computer programming.

4.2 Typing programs

Programs for the robot are entered into a *code cell* located on the left. In Level 1, we can use the commands `left`, `right`, `go`, `get`, and `put`. Their function is the same as the function of the corresponding buttons in Manual mode. One or more commands form a *computer program (computer code)*. Often we just say *program* or *code*. There are two simple rules to remember:

1. Always type one command per line, it greatly adds to code readability.
2. Indentation matters - do not enter extra empty characters in front of commands.

Ignoring these rules would not necessarily make your program invalid, but the code would be difficult to read. It is very important to write a clean, well readable code.

4.3 Algorithm

Karel will always follow your commands *exactly*. No exceptions. Sometimes it will happen to you that the robot does something else than you expected. In this case most likely your *algorithm* was wrong. An *algorithm* is a sequence of steps (operations) that the robot follows in order to fulfill his task. Algorithms are written using normal human language, not in terms of computer code. Imagine that Karel's task is to pick up the gem and return home, as shown in Fig. 7.

Fig. 7: Collect the gem and get home!

An algorithm to solve this task would be:

```
Make two steps forward
Collect the gem
Turn around
Make three steps forward
```

4.4 Program

Computer program or just *program* is created by translating an algorithm into a particular programming language – in our case the Karel language. It is always important to design a good algorithm. If the algorithm is good, then translating it into a computer program is straightforward. The above example is trivial of course, we will have a chance to work with more complicated algorithms later. One possible program corresponding to the above algoritm is:

```
go
go
get
left
left
go
go
go
```

Often there is more than one way to translate an algorithm into a computer program. For example, the above algorithm can be also translated into

```
go
go
get
right
right
go
go
go
```

4.5 Logical and syntax errors

Mistakes in algorithms are called *logical errors*. Let's return to the setting shown in Fig. 7 and consider the algorithm

```
Make three steps forward
Collect the gem
Turn around
Make three steps forward
```

That would hurt! This algorithm makes the robot crash into the wall.

Mistakes such as mis-spelling a command, writing "1O" instead of "10", or forgetting indentation are related to *syntax* and they are called *syntax errors*. Find two syntax errors in the following program!

```
go
go
got
r1ght
right
go
go
go
```

Mistakes or either kind are called *bugs* and the procedure of eliminating them is called *debugging*. Depending on how careful we were while preparing our algorithm and writing the program, debugging takes either a short time or a long time. It does not happen often that a program works correctly right away.

Of logical and syntax errors, the former are much harder to find.
Always make sure to design a good algorithm before you start coding!

When our program contains a syntax error, the robot outputs an *error message* and does nothing. If the algorithm contains a logical error, then multiple things may happen: The robot may execute the program without fulfilling the goals. Or, he may do something that will trigger and error message and stop program execution. This includes:

- Crashing into a wall.
- Attempting to collect a gem where there is none.
- Attempting to put a gem on the ground while his bag is empty.

5 Counting Loop

In Section 4 we learned to command the robot using written words rather than mouse clicks, but it hardly could be called programming. The first programming concept – the

counting loop – awaits us in this section. Counting loops are present in all procedural programming languages and they allow us to repeat some action a given number of times (such as three times, ten times). In Karel, counting loops are done via the `repeat` command, other languages use various constructs based on the keyword `for`.

5.1 Objectives

- Start learning higher-level programming concepts.
- Learn to make the robot repeat something a given number of times.

Please switch Karel to Level 2 in Settings to enable the higher-level programming functionality presented in this section.

5.2 The `repeat` command

Imagine that Karel needs to go get the gem as shown in Fig. 8, and then return home.

Fig. 8: Repeating an action a given number of times.

In principle we could type

```
go
go
go
go
go
go
go
go
go
go
get
left
left
```

```
go
go
go
go
go
go
go
go
go
go
go
```

and this program would get the job done. But it is not very elegant. Instead, the same can be achieved by telling Karel to `repeat` the `go` command `10` times, get the gem, turn back, and `repeat` the `go` command another `11` times:

```
# Walk to the gem:
repeat 10
    go
# Pick it up:
get
# Turn back:
repeat 2
    left
# Walk home:
repeat 11
    go
```

We even saved space for comments! All text that follows the hash symbol '`#`' is ignored by the interpreter till the end of line.

> Commenting your code is a very good habit. It helps others to understand your code, and it also helps you when you return to your own code after some time.

5.3 Body of the loop and indentation

Note that in the last program, commands to be repeated (the *body of the loop*) are indented. In this case, each loop's body is formed by a single command, but if there were more of them, all of them would be indented. In Karel, you can choose between a 2-

indent and a 4-indent. The former yields more compact code with not-so-long lines, but the latter is easier to read.

Having the indentation right is extremely important. If you make a mistake here, you can easily damage your robot! Assume the maze shown in Fig. 9. Karel's task is to walk around the block and stop in the upper left corner, facing South.

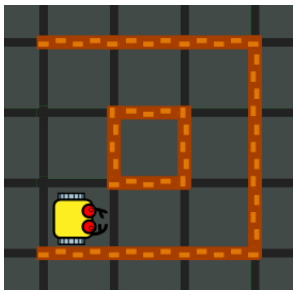


Fig. 9: Karel needs to go around a block.

Here is the program that will do it:

```
repeat 3
    go
    go
    left
```

After this program is executed, the robot's position is as shown in Fig. 10.

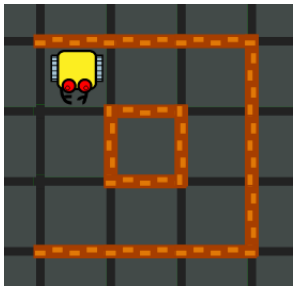


Fig. 10: Karel is where he is supposed to be.

However, let's say that the last command of the loop's body is unindented by mistake:

```
repeat 3
    go
    go
left
```

Then the robot will make just two steps before crashing right into the wall:

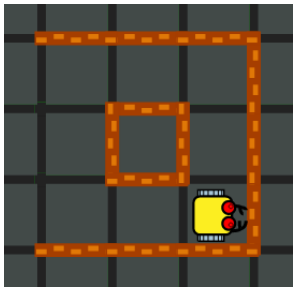


Fig. 11: Karel just crashed!

There will be an error message telling where the problem occurred:

```
Ouch, you crashed me!
Line 2:go
```

5.4 Two nested `repeat` loops

In the last two programs, the `go` command was written twice. This is not that bad since using the `repeat` command and writing one command per line, we would also need two lines. But consider the situation shown in Fig. 12 and imagine that Karel has to walk around the block now, returning to his original position.

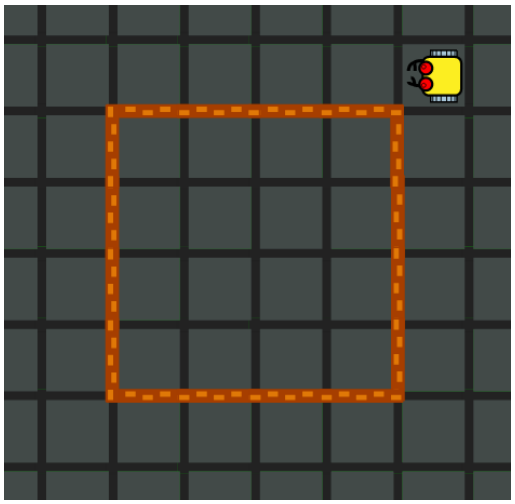


Fig. 12: Walk around a larger block.

Writing the `go` command inside the `repeat` loop five times would not be too nice. Instead, the following code does the job elegantly:

```
repeat 4
    repeat 5
        go
    left
```

When a loop is used within another loop's body, we say that the loops are *nested*. Notice that the same indentation scheme applies to each of the two loops. The `go` command that is inside of the internal loop, is indented by eight empty characters, while the `left` command which does not belong to the internal loop anymore is only indented by four.

5.5 Three nested `repeat` loops

Let's look at one last example in this section. Karel stands on the intersection of two streets that separate four blocks, as shown in Fig. 13. His task is to walk around each block, always returning to his initial position before starting a new block.

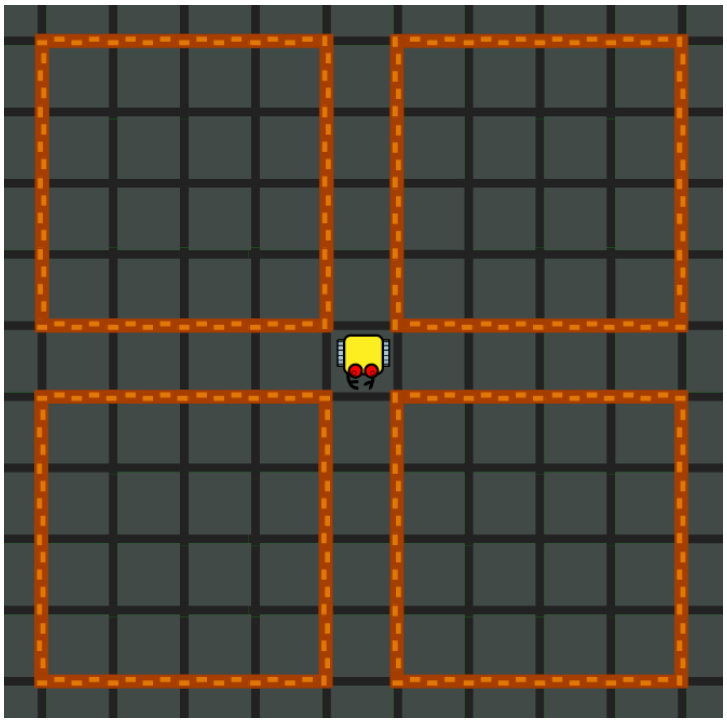


Fig. 13: Standing on an intersection.

Here is the program that will do it:

```
repeat 4
    repeat 4
        repeat 5
            go
        left
    right
```

This program features three nested `repeat` loops. Take the time to reconstruct this example in Karel and run the program – it is well worth the effort!

6 Working with Code and HTML Cells

6.1 Objectives

– Learn how to add and use new code cells and HTML cells.

– Learn how to change the order of cells.
– Learn how to run all code cells at once, and how to run them individually.
– Learn how to clear, collapse, remove and merge cells.

6.2 Code cells

As we know well by now, programs are entered in code cells. So far we have worked with a single code cell, but our worksheet can have more of them. Having multiple code cells can be useful to insert nice descriptions between various parts of the code, to test various versions of our program, or if we want to run parts of the program separately. Fig. 14 shows a sample code cell including its bottom menubar.

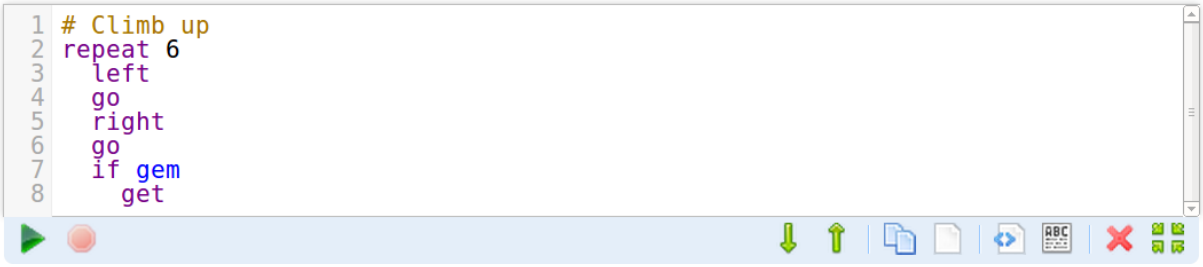


Fig. 14: Sample code cell.

Left to right, the icons under the code cell have the following meaning:

– Run the program in this particular code cell.
– Stop the code cell (this icon becomes active when the program is running).
– Move the cell under the one below it (changes the order of cells in the worksheet).
– Move the cell above the one above it (changes the order of cells in the worksheet).
– Duplicate the cell (new code cell with the same contents is created).
– Clear the cell (erase all contents).
– Add empty code cell under the cell.
– Add empty HTML cell under the cell.
– Remove the cell.
– Collapse the cell.

6.3 HTML cells

HTML cells use a WYSIWYG text and HTML editor to add descriptions and illustrations to the worksheet. Fig. 15 shows a sample HTML cell.

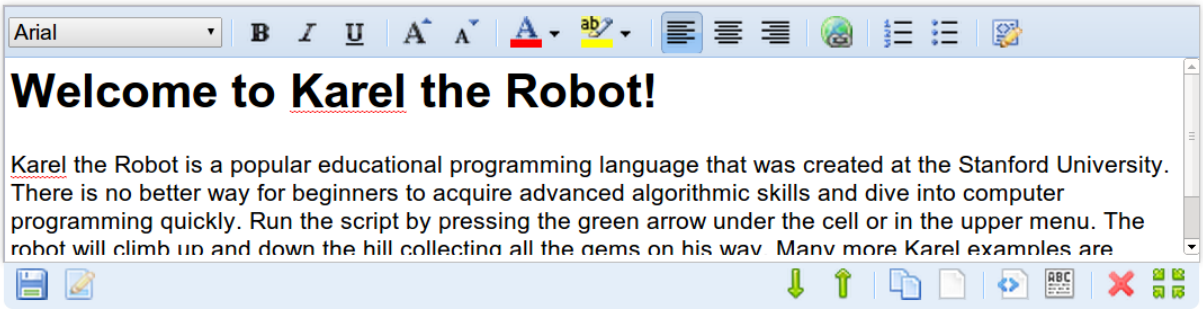


Fig. 15: Sample HTML cell.

This cell has two menus. The top one is related to text editing and inclusion of images, the bottom one is analogous to the menu of code cells. Let us begin with the top menu: Left to right, the buttons and icons have the following meaning:

- Select text font.
- Make selected text boldface.
- Make selected text italics.
- Underline selected text.
- Increase font size for selected text.
- Decrease font size for selected text.
- Choose foreground text color
- Choose background text color
- Align text left
- Center text
- Align text right
- Add a hyperlink.
- Create enumerated list.
- Create bullet list.
- Edit source HTML code. This is also how images can be added via external links.

The bottom menu, left to right:

- Save the HTML cell.
- Edit the HTML cell.
- Move the cell under the one below it (changes the order of cells in the worksheet).
- Move the cell above the one above it (changes the order of cells in the worksheet).
- Duplicate the cell (new HTML cell with the same contents is created).
- Clear the cell (erase all contents).

– Add empty code cell under the cell.
– Add empty HTML cell under the cell.
– Remove the cell.
– Collapse the cell.

Additional operations with cells such as their merging can be done via the Edit menu.

7 Conditions

After a short intermission in Section 6 we are going to explore another fundamental programming concept – *conditional execution of code*. Conditions are present in all programming languages and they allow us to handle various situations as they arise while the program is running. In Karel's case, conditions will mostly be related to checking his surroundings via the five sensors `wall`, `gem`, `empty`, `north` and `home`.

7.1 Objectives

– Understand the role of conditions in programming.
– Learn to use Karel's sensors in conjunction with conditions to help the robot check his surroundings and react accordingly.

7.2 Using the `wall` sensor

The robot can use the `wall` sensor to determine whether it is safe to make one step forward. The usage of this sensor can be illustrated using a simple program "Careful step" where Karel first checks whether there is a wall ahead before making a step. If there is wall, he turns back:

```
# Program "Careful step".
if wall
    repeat 2
        left
else
    go
```

Imagine that Karel stands in front of a gem as shown in Fig. 16.

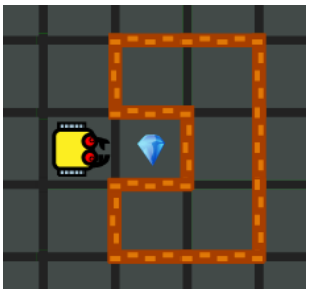


Fig. 16: Karel's initial position.

Now let us run the program three times for fun (three clicks on the green arrow button will do). The sequence of Karel's positions after each evaluation is shown in Fig. 17.

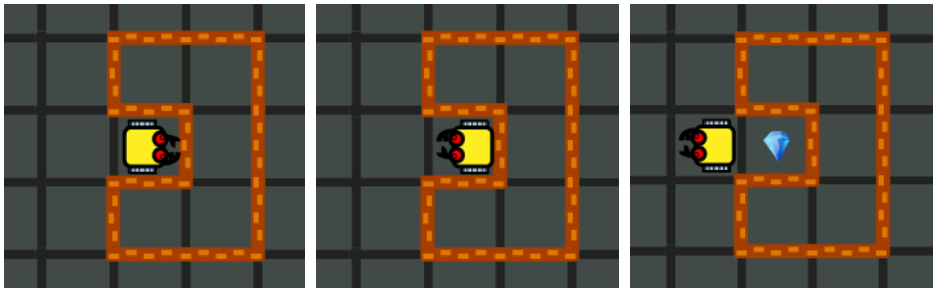


Fig. 17: Left to right – Karel's positions after executing the program "Careful step" one, two, and three times.

Note that commands forming the body of the `if` and `else` branches are indented. This is done for the same reason as in the `repeat` loop.

7.3 Using the keyword `not`

Karel can also use the keyword `not` (negation). For illustration, the last program can be rewritten as follows, without changing its function:

```
# Program "Careful step".
if not wall
    go
else
    repeat 2
        left
```

7.4 Using the `gem` sensor

The `gem` sensor allows Karel to check whether he stands at a gem. This is the only way for the robot to pick up a gem: His hands are too short to reach into another square. And, he cannot "see" gems which lie in other squares – simply because he does not have "eyes". Returning to the situation depicted in Fig. 16, the robot does not have a chance to know that a gem lies one step ahead.

Sometimes, we tend to make a mistake of exploring the maze with our own eyes instead of using the robot's sensors. Use your empathy skills and feel youself into the robot. He cannot hear, because he does not have a sensor for that. He does not have a sense of smell either. Let's always view the problem at hand from the robot's perspective. If you can do this, your programming ability will expand tremendously. Consider the situation shown in Fig. 16 again,

Fig. 18: Karel's initial position, same as before.

and let's extend the program "Careful step" in such a way that Karel picks up the gem on the way. This is not too difficult:

```
# Program "Careful step II".
if gem
    get
if wall
    repeat 2
        left
else
    go
```

Let us run the program three times again. The sequence of Karel's positions after each evaluation is shown in Fig. 19.

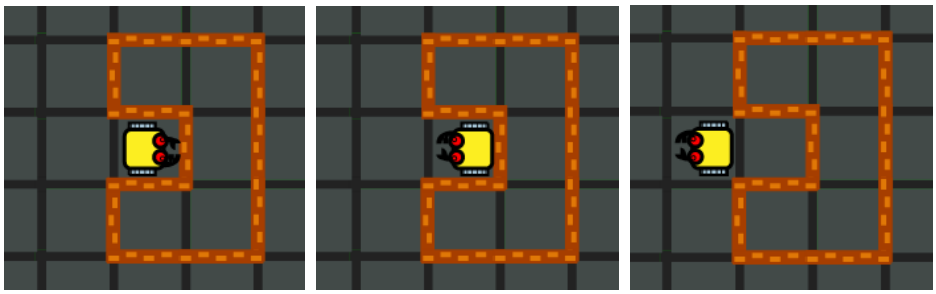


Fig. 19: Left to right – Karel's positions after executing the program "Careful step II" one, two, and three times.

7.5 Using the `empty` sensor

The `empty` sensor allows the robot to check whether his bag is empty, or, in the combination with the keyword `not`, whether his bag contains at least one gem. Imagine that Karel's block has four houses with pavements around them, as shown in Fig. 20. After the winter the pavements are damaged and some tiles (gems) are missing. Karel has an unknown number of tiles in his bag. Write a program for him to repair the pavements. Important: Your program must not throw an error if Karel runs out of gems!

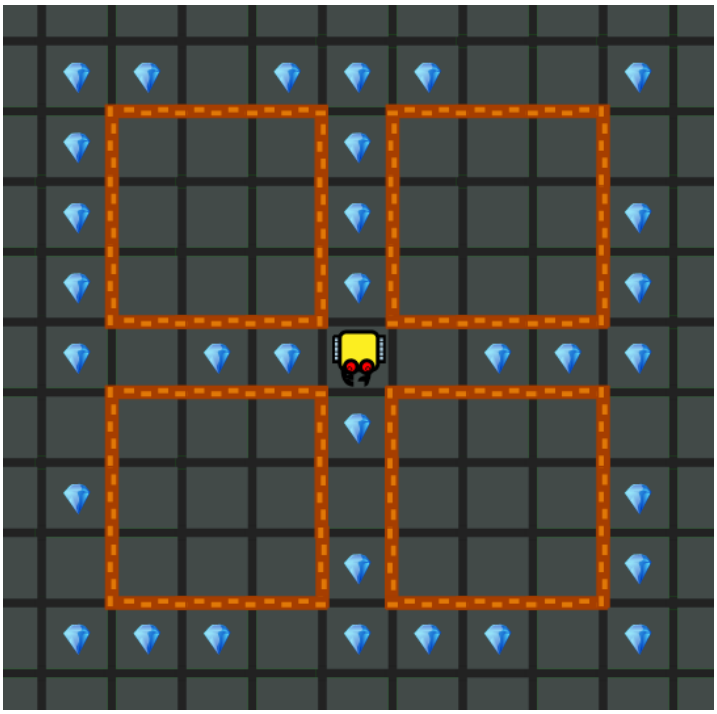


Fig. 20: Karel is repairing pavement.

We can use the program from Subsection 5.5 as the basis for our new program. First let's adjust the numbers of repetitions in the loops to match the current layout of the maze. In fact just the deepest one needs to be changed from five to four since the new blocks are only three units long. The other two loops remain unchanged since each square has still four edges and there are four squares as before. So the new program is:

```
repeat 4
    repeat 4
        repeat 4
            go
        left
    right
```

This will work, but Karel will not be doing any repairs, just walk all pavements and return to his initial position. To repair the pavement, we need to insert an additional condition in front of each `go` command:

```
repeat 4
    repeat 4
        repeat 4
            if not gem
                put
            go
        left
    right
```

Is this program OK? Almost, but not quite. If the number of gems in Karel's bag is less that the number of missing tiles in the pavement, the program will stop with an error message. So, we need to check the `empty` sensor before each `put` command. The final program has the form:

```
repeat 4
    repeat 4
        repeat 4
            if not gem
                if not empty
                    put
            go
        left
    right
```

Nice work! Fig. 21 shows the pavement after the program is execured. Karel started with 15 gems in the bag (gems can be added to the robot's bag in Designer).

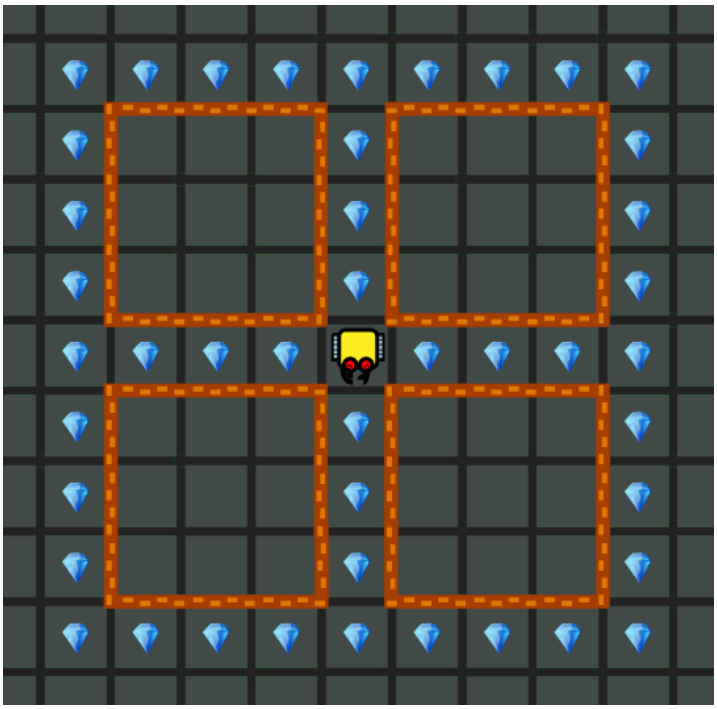


Fig. 21: After running the program, the pavement is flawless again!

As an exercise, run this program with just 10 gems in the robot's bag!

7.6 Gambling

Returning for a moment to Fig. 18,

Fig. 22: Karel's initial position.

it is not clear whether the robot stands on a gem or not – a gem under the robot would not be visible. So let's try our luck and execute the command

```
get
```

Nay, instead of a gem, we get an error message!

```
Oops, there is no gem here!
Line 1: get
```

We say that the code *crashed*, even though Karel did not crash into a wall.

We must not write codes that can crash.

The error could have been avoided easily by using the `gem` sensor before the `get` command:

```
if gem
    get
```

Also the commands `go` and `put` can potentially lead to an error. Unless we are really sure that this will not be the case, we can always check the corresponding sensor before executing these commands. For the `go` command this would be

```
if not wall
    go
```

and for the `put` command

```
if not empty
    put
```

So, keep the following in mind:

Do not gamble. If you are not 100 % sure, check a sensor prior to using commands `go`, `get` and `put`. Checking sensors is quick, it does not make your program slower.

7.7 The `north` and `home` sensors

By now you understand the concept of sensors very well, so there is no need to spend much more time on the remaining two. Let us just mention that the `north` sensor can be used to make the robot face any given direction. For this, we always must turn him to face North first, because this is the only direction he can verify. From there, one `left` command will turn him West, etc. The `home` sensor helps the robot to make sure that he reached his home.

8 Conditional Loop

The *conditional loop* is a major concept that is used in all procedural programming languages. It allows us to repeat some action without knowing in advance how many repetitions will be needed. This can be the case, for example, when Karel needs to walk straight until he reaches a wall. Recall that he cannot see a wall that is not right in front of him. Another example is when the robot needs to turn to face North. Recall that he can't know which direction he is facing – he can only check whether he faces North or not. In most languages (including Karel), the conditional loop is represented via the keyword `while`. In others, one can find constructs such as `do - while`, `do - until`, `repeat - until` etc.

8.1 Objectives

– Learn to repeat a command or a sequence of commands when it is not known in advance how many repetitions will be needed.

8.2 The `while` command

Consider for a moment the situation shown in Fig. 23. Karel stands at a random position in the maze and he knows that there is one gem somewhere at the exterior wall, but he does not know the exact location. Let's write a program for the robot to go get the gem!

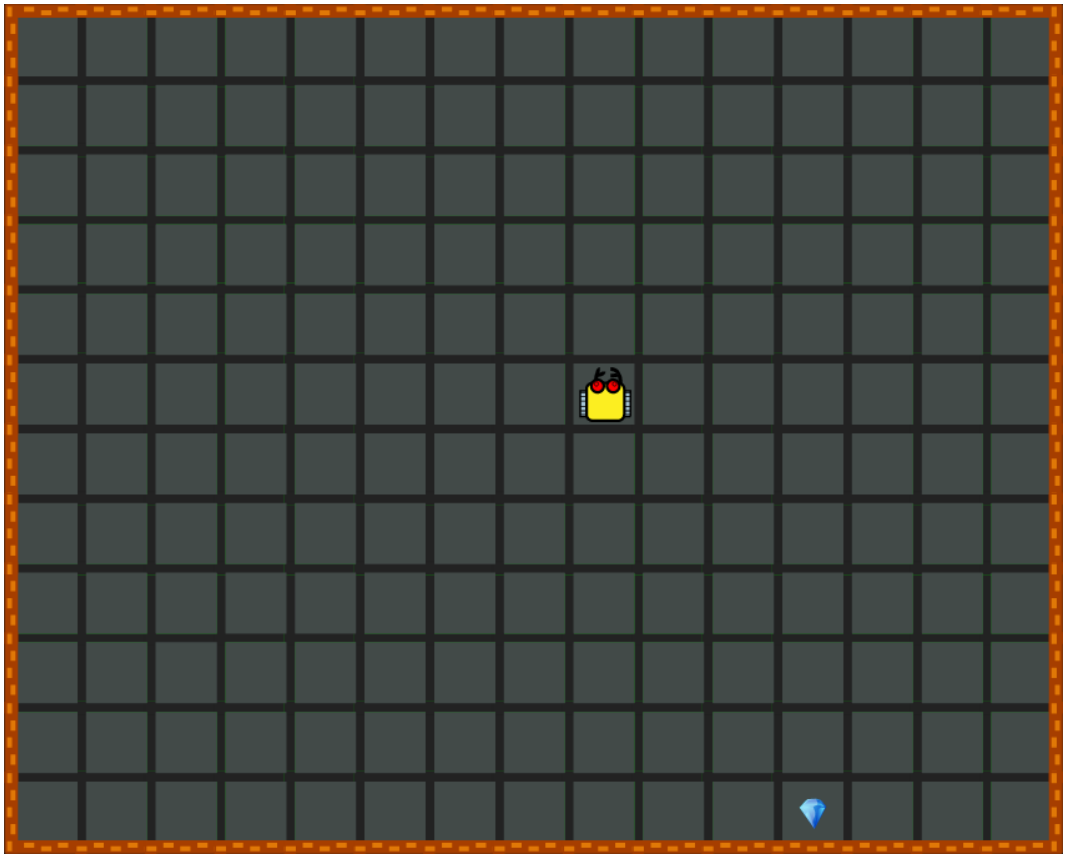


Fig. 23: There is one gem in the maze, located at the exterior wall.

The program would be:

```
while not gem
    while not wall
        go
    left
get
```

Notice that the body of the `while` loop is indented, same as the body of the `repeat` loop. The above program is written well since it can also handle the more difficult situation shown in Fig. 24. There, the robot runs into a corner where he needs to turn twice.

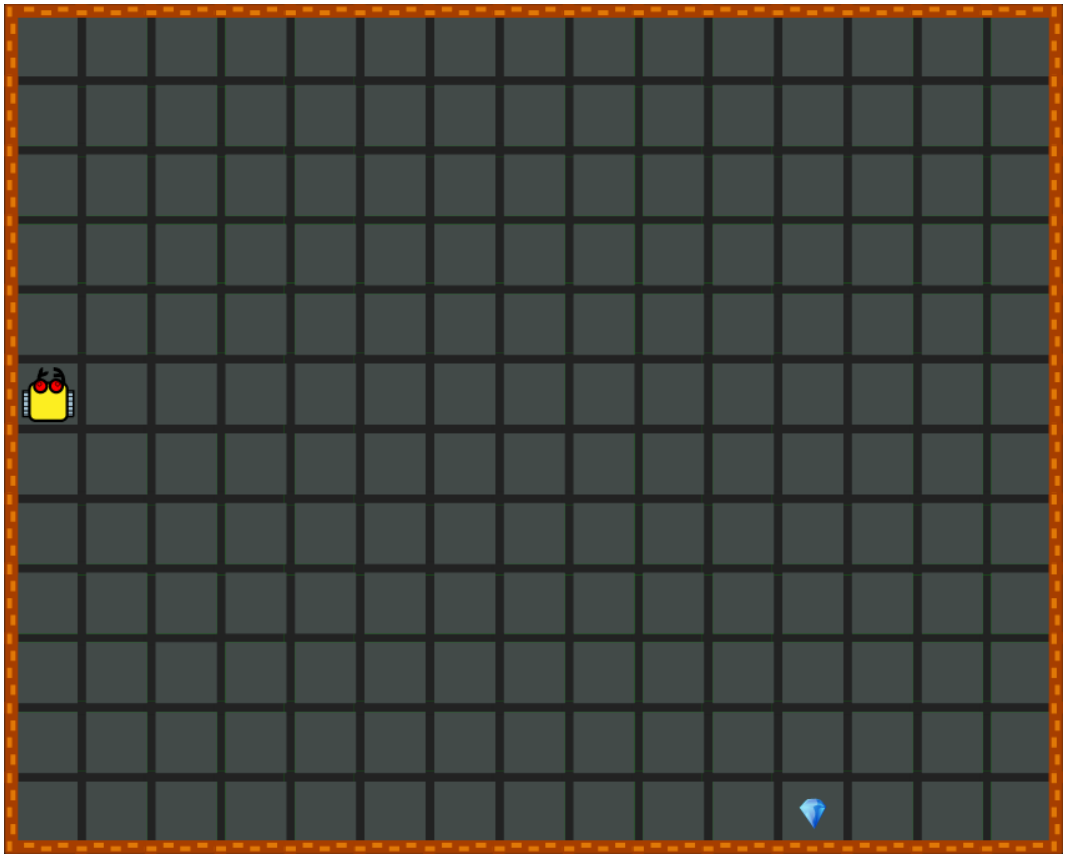


Fig. 24: Karel will react correctly even if he needs to turn twice.

8.3 Think like the robot

Next let us solve the problem shown in Fig. 25, where Karel knows that his home is straight ahead of him, there are no walls on the way, and there are gems that he needs to collect.

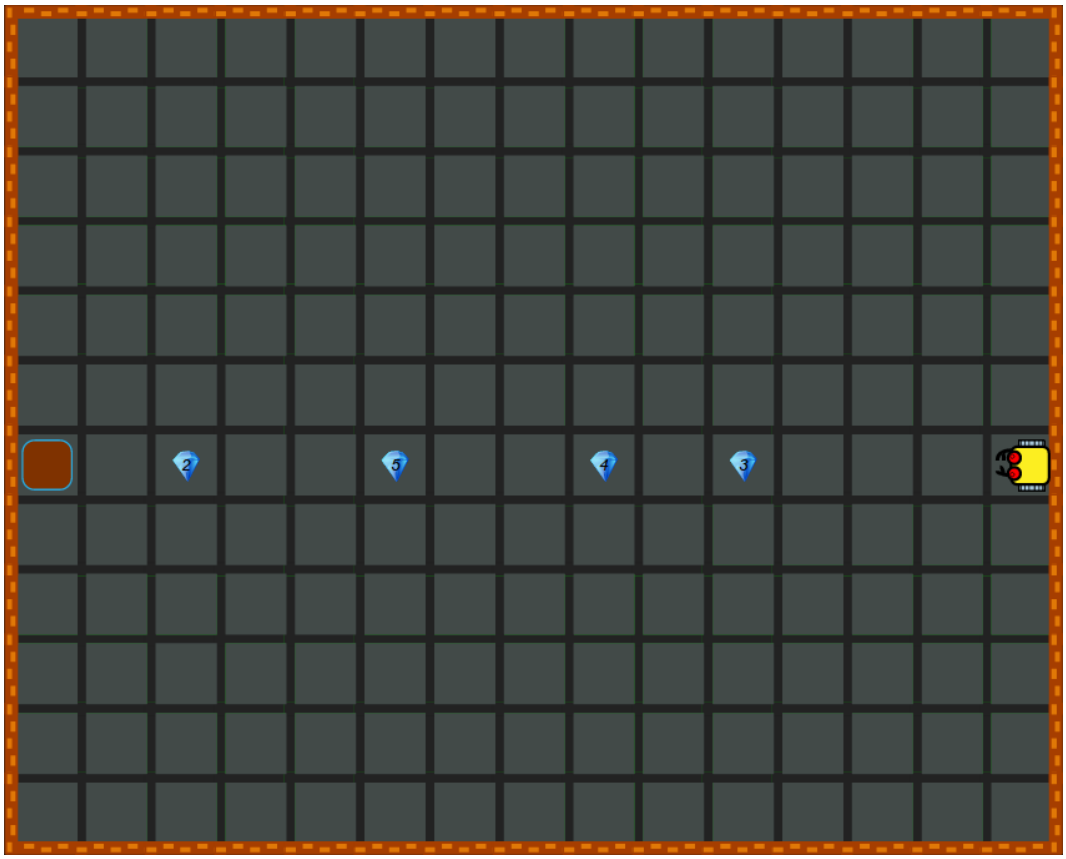


Fig. 25: Karel needs to get home and collect all gems on the way.

First let us show a bad program to do this:

```
repeat 4
    go
repeat 3
    get
repeat 2
    go
repeat 4
    get
repeat 3
    go
repeat 5
    get
```

```
repeat 3
    go
repeat 2
    get
repeat 2
    go
```

The program works, so what is the problem? The problem is that while writing the program, the author was thinking like the observer and not like the robot. The observer knows everything about the maze – the exact position of all gems as well as how many gems are in each pile. However, the robot does not know any of that. As a result, the program is very fragile – it will crash as soon as the position of one of the gems is changed, or if the number in one of the piles will change. That is not good.

A much better approach to writing the program is to think like the robot. This means, cover the maze so that you do not see it, and write the program knowing only what the robot knows – that the home is straight ahead, that there are no walls on the way, and that there are gems that need to be collected. Then the program is completely different:

```
while not home
    while gem
        get
    go
```

Notice that the second `while` loop also takes care of the situation when there is no gem where the robot is. No need to use an extra `if` condition to check that.

As a bonus for thinking like the robot, our program is now universal. It will work for any maze where the home is straight ahead of the robot with no walls and some gems on the way. One such maze is shown in Fig. 26.

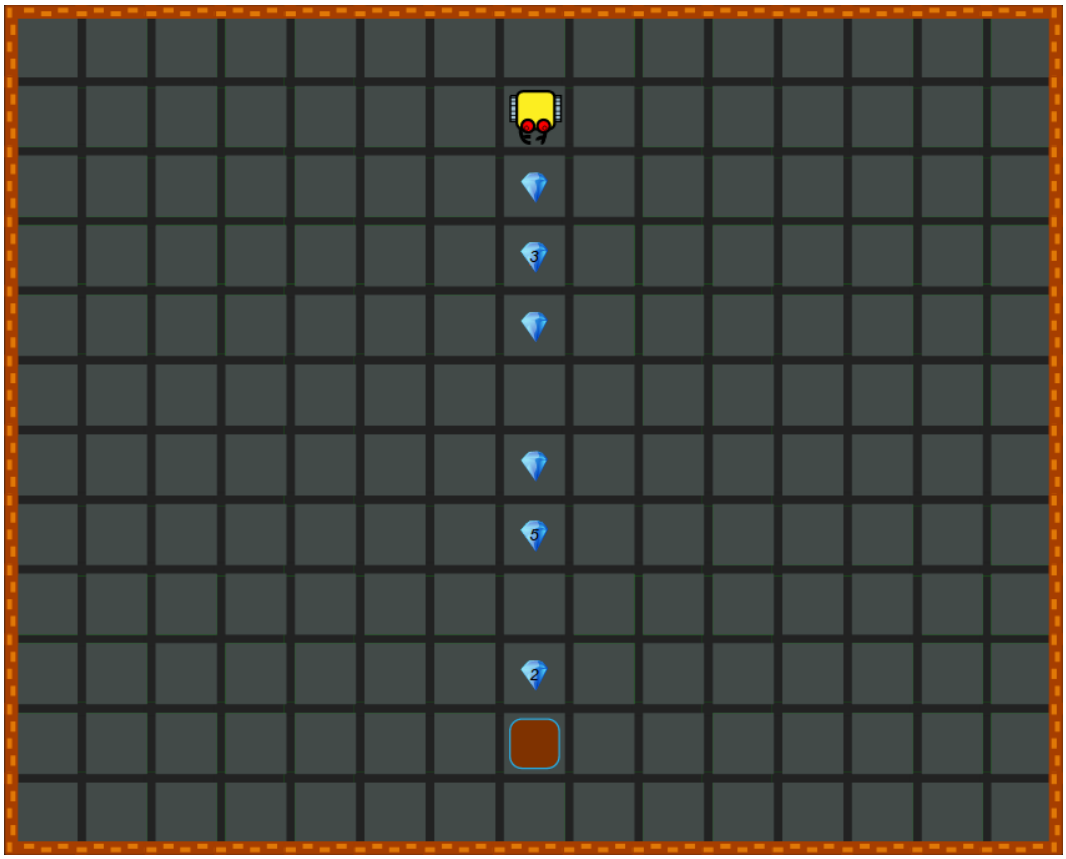


Fig. 26: The program now works for many different mazes!

9 Custom Commands

When solving a complicated task, it is a good idea to always check whether it contains simpler problems that could be solved first. Usually there are one or more of them. Once these smaller problems are solved, and custom commands (*subprograms*) are created for them, then the original difficult task is not that difficult anymore. Custom commands can represent multiple commands or even a longer program. Computer code forming a custom command is easy to reuse in various parts of our program. When a change to the custom command is needed, it is done only in its definition, and the change then propagates automatically to all occurrences of the command in our program.

9.1 Objectives

- Learn that splitting a big task into smaller ones will simplify its solution.
- Learn that creating new commands brings more clarity into your program.
- Learn that repeating the same code in various parts of our program is a bad habit.

9.2 Defining new commands

New commands are defined using the reserved word `def`. For example, in a program where the robot needs to turn back many times, it is a good idea to define a new command `turnback` as follows:

```
def turnback
    repeat 2
        left
```

Note the indent – the body of a new command needs to be indented analogously to the body of loops and conditions.

9.3 Arcade game

This time Karel needs to go through several levels of an arcade game shown in Fig. 27, collect all gems in each level, and reach his home which is located at the East end of the top level. There is only one opening between each two levels. Initially, the robot stands somewhere in the bottom level. The example can be cloned from user "nclab", its name is "Arcade game".

Clearly, this problem is a bit more complicated than the problems that we solved before. The way to solve it is to first look for tasks that the robot will be doing in each level. For sure, he will need to turn around from time to time, so let's start with introducing the `turnaround` command that we know from before:

```
# New command to turn around:
def turnback
    repeat 2
        left
```

Since he does not know the exact positions of gems, the robot will always need to sweep the entire level. Let's introduce a new command for this. We will assume that the robot stands at the West end of the level, facing East:

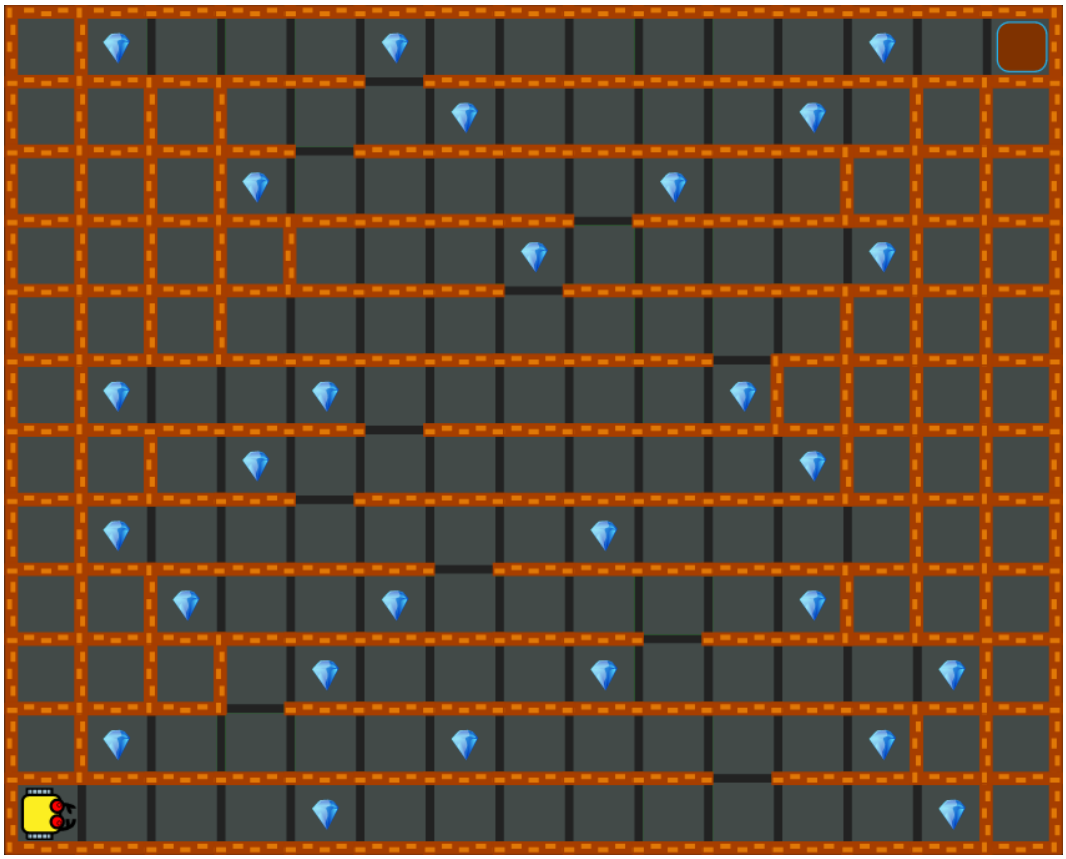


Fig. 27: Karel is playing an arcade game.

```
# Sweep one level from left to right.
# Assumes that robot stands at the
# West end, facing East:
def sweep
    while not wall
        while gem
            get
        go
        # Do not forget the last gem:
        while gem
            get
```

Next we need a new command that will make sure that the robot stands at the West end of the level, facing East, as required by the command `sweep`. Let us call this new

command `gowest`

```
# Reach West end of the current level
# and turn around to face East:
def gowest
    # Turn West:
    while not north
        left
    left
    # Go to West end:
    while not wall
        go
    # Turn around:
    turnback
```

Almost done! The last new command we need is to get to the next level. Let us call it `moveup`:

```
# Find the opening and move one
# level up. Assumes that robot is
# at the East end of a level,
# facing East:
def moveup
    if not home
    # Face North:
    left
    # Find opening:
    while wall
        left
        go
        right
    # Pass through opening:
    go
```

In the last step we put together the previously defined commands `gowest`, `sweep` and `moveup` to define a new command `arcade`:

```
# Main function:
def arcade
    while not home
        gowest
        sweep
        moveup
```

The main function is then called as follows:

```
# Main program:
arcade
```

Fig. 28 shows the arcade after the program has finished:

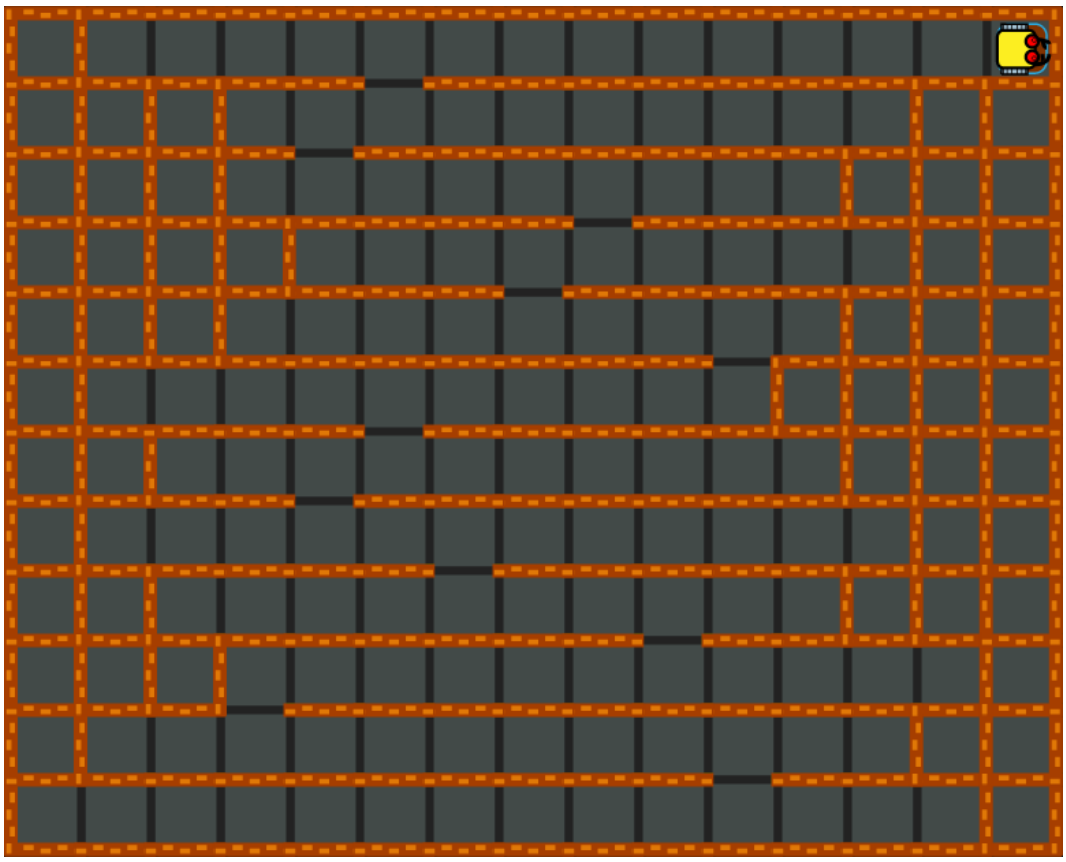


Fig. 28: Arcade after program has finished.

9.4 Never replicate computer code

Your Karel programs will probably not have thousands of lines, and in shorter programs copying and pasting of the same code will not cause much harm. Nevertheless, even now we should understand why this is an extremely bad programming practice that should be avoided.

A new command should be defined whenever it becomes clear that the same action is repeated in the algorithm multiple times. When we start writing a program and then realize that the same code repeats itself at various places, then probably we did not do a good job designing the algorithm.

Sometimes it might be tempting to just replicate the same code several times in your program, because it does the same thing. Do not do this. Sooner or later, you will need to adjust something in the block of code that you copied, and you will forget to make the change in all occurences of the code.

Then your code will start acting strange and you will not know why. It will sometimes work and sometimes not. You will spend lots of time looking for mistakes. You will find some of them but not all, and your program will eventually become unreliable. Later, you will find that the only solution is to throw the program away and rewrite it from scratch.

Avoid code repetition as well as very long blocks of code. Define new commands whenever appropriate. Keep the code well commented. This is the only way to keep your code reliable, correct, and healthy.

10 Recursion

10.1 Objectives

- Understand what recursion is and when it can be useful.
- Learn to write good recursive algorithms.

By a *recursive* algorithm we mean an algorithm that makes a call to itself. How does it sound? In our life we use recursion all the time, without noticing it. For example, when we descend a staircase, our algorithm is:

```
Descend_staircase
    Descend_one_step
    If this_was_not_the_last_step
        Descend_staircase
```

Recursion is not applicable to all types of problems, but it can be very helpful, especially for problems where we can

- reduce the problem to a similar one which is smaller in size,
- apply the same algorithm to the smaller problem.

On program level, this means that some command calls itself, either directly or through other commands.

10.2 How it works

Consider the following Karel program:

```
def reach_wall
    if not wall
        go
        reach_wall

reach_wall
```

Imagine that the initial position of the robot is like in Fig. 29.

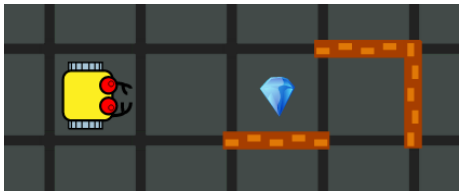


Fig. 29: Robot's initial position.

When the command `reach_wall` is first called, the robot stands three steps away from the wall and thus the `if not wall` condition passes. Then the command `go` follows and the robot's position changes as shown in Fig. 30.

Fig. 30: Robot's position after making the first step forward.

Next the robot executes the `reach_wall` command that follows the `go` command. A good way to understand what happens is to imagine that the command is replaced with its own body. So the corresponding code would look as follows:

```
if not wall
    go
    if not wall
        go
        reach_wall
```

Since the robot is two steps away from the wall, the second `if not wall` condition passes and he makes a second step forward. His new position is shown in Fig. 31.

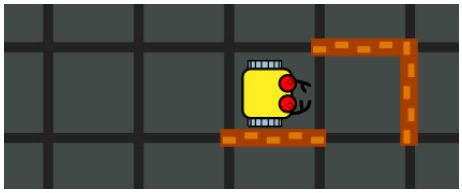


Fig. 31: Robot's position after making the second step forward.

Next the robot executes the third `reach_wall` command. Again we can imagine that the command is replaced with its own body. The corresponding code would look as follows:

```
if not wall
    go
    if not wall
        go
        if not wall
            go
            reach_wall
```

Since the robot is one step away from the wall, the third `if not wall` condition passes and he makes a third step forward. His new position is shown in Fig. 32.

Fig. 32: Robot's position after making the third step forward.

Now the robot stands in front of the wall, so it gets more interesting. The `reach_wall` command is executed one more time, so we can imagine that its body is pasted into the code one last time:

```
if not wall
    go
    if not wall
        go
        if not wall
            go
            if not wall
                go
                reach_wall
```

However, now the `if not wall` condition does not pass, which means that the program is finished!

10.3 The base case

In the last example we have observed one important fact: To prevent infinite recursion, we used an `if` statement. The `else` statement was omitted which meant "else do nothing". In recursive algorithms, we always need an `if` or `if-else` statement of some sort, where one branch makes a recursive call while the other one does not. The branch without a recursive call is called the *base case*. A bad example of a recursive command without a base case would be

```
def left_forever
    left
    left_forever
```

This program is an infinite recursion that needs to be stopped using the red stop button.

10.4 When should recursion be used?

The recursive command `reach_wall` that we defined above may not be the most useful example since the same functionality could be achieved more elegantly without recursion, just with

```
def reach_wall
    while not wall
        go
```

If there is a small task that needs to be solved repeatedly in order to get a bigger task done, then often one can write both a non-recursive and recursive version. Recall for example the Diamond Staircase example from Section 9. The small task there was to climb one step and pick up the gem, ending up facing east. A non-recursive version of the algorithm would be:

```
def climb_one_step
    left
    go
    right
    go
    get

while not home
    climb_one_step
```

A recursive version of the same:

```
def climb_the_stairs
    if not home
        left
        go
        right
        go
        get
        climb_the_stairs

climb_the_stairs
```

If an algorithm comes in both a recursive and a non-recursive version, then the following should be taken into account:

- The recursive version will be slightly slower than a non-recursive one. The reason is the overhead related to creating a new instance of the recursive command and calling it.
- The recursive version will also require more memory – when a recursive command is called 1000 times, then it actually exists in 1000 copies in the memory. Hence, recursion is not recommended with very large numbers of repetitions.

Recursive algorithms are used mainly where non-recursive ones are cumbersome to design. For example, much code written for traversing tree-like data structures is recursive. Also certain sorting algorithms are more naturally written in recursive form. We will discuss these subjects in more detail later.

10.5 Mutually recursive commands

Recursion can have interesting forms. For example, there can be a pair of commands that call themselves mutually, such as the commands `odd` and `even` in the following example (that also solves the Diamond Staircase problem). Note the presence of base case in both recursive commands:

```
def climb_step
    left
    go
    right
    go
    get

def odd
    if not home
        climb_step
        even

def even
    if not home
        climb_step
        odd

odd
```

11 Variables and Functions

In this Section we are entering exciting Level 3 where Karel grows up and leaves the home of his parents to experience life on his own. Therefore, his home will not be present in the maze anymore. There are additional changes that reflect Karel's growing up – there is a GPS device that he can use to determine his position in the maze, he can print messages, work with variables and lists, employ functions that return values, use more complex logical operations, make random decisions and more. Do not forget to switch to Level 3 in Settings. A compact overview of new functionality in Level 3 can be found in Section 14.

11.1 Objectives

- Learn to make a random decision.
- Understand the concept of variables.
- Learn to work with numerical and logical variables and text strings.
- Learn to use functions that return values.
- Understand that variables defined inside functions are local.

In programming, variables are used to store useful information for later use. To give a few examples, this information can be a number, word, sentence, or a logical value ("true" or "false").

11.2 Types of variables

All of us use variables in our lives. One of the first ones is our own name.

Text strings

With a bit of abstraction (and say that your name is "Melissa"), when you were about two years old, you did the following:

```
my_name = "Melissa"
```

The variable `my_name` stores a text (string of characters). Since then, each time someone said a name, you retrieved in your brain the value of the variable `my_name`, compared it to the name that you heard, and if you got a match then you turned around to see who was calling you.

Numbers

We also use numerical variables such as

```
seconds_per_minute = 60
```

or `minutes_per _hour` whose value is 60 as well, `hours_per_day` whose value is 24, and so on. The last three variables do not change too often, most likely they will not change during our lives. But we also use variables whose values change, such as `days_per_year`, `number_of_my_pets`, etc. In Karel, we will only use integers (not general real numbers).

Logical values

Logical variables can only store two possible values: `True` or `False`. We use many of them. One such example:

```
I_speak_a_foreign_language = True
```

Of course, for someone else this variable will have the value `False`. The important thing is that when someone asks you whether you speak a foreign language, you can retrieve this information quickly. The value is *stored*, it does not have to be *created* again. Logical variables and operations will be discussed in more detail in Section 13. In the rest of this section we will work with integers and text strings.

11.3 Using the GPS device and the `print` command

Karel's coolest Christmas present was a new GPS device that allows him to determine his position in the maze. He can retrieve his coordinates at any time via the commands `gpsx` and `gpsy`. He also has a new ability to output text messages via the `print` command. The usage of these commands is best illustrated using the following short program where Karel determines his coordinates in the maze and prints them:

```
print "Horizontal position:", gpsx
print "Vertical position:", gpsy
```

The south-west corner of the maze is the origin of the coordinate system and it has coordinates [0, 0]. With Karel's position shown in Fig. 33,

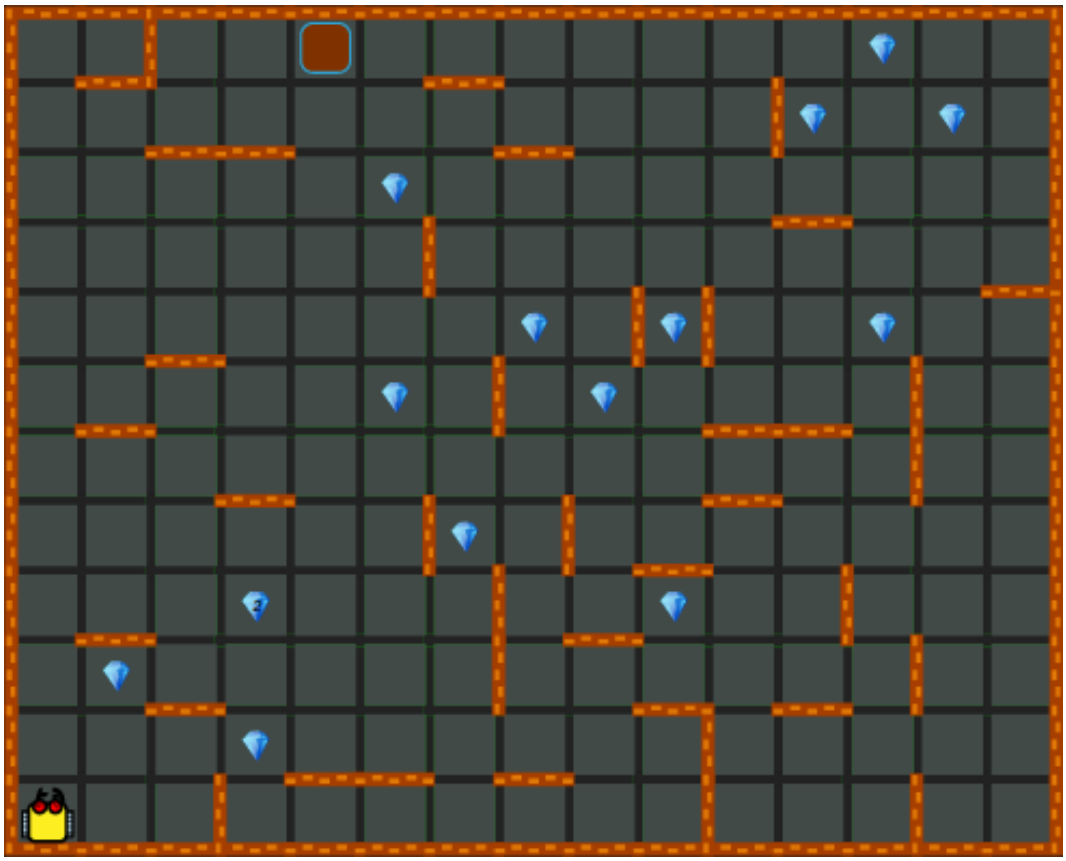


Fig. 33: South-west corner of the maze has GPS coordinates [0, 0].

the above program will have the following output:

```
Horizontal position: 0
Vertical position: 0
```

The maze's width (in west-east direction) is 15 tiles, and its height (in south-north direction) is 12 tiles. If Karel stands in the north-east corner as shown in Fig. 34,

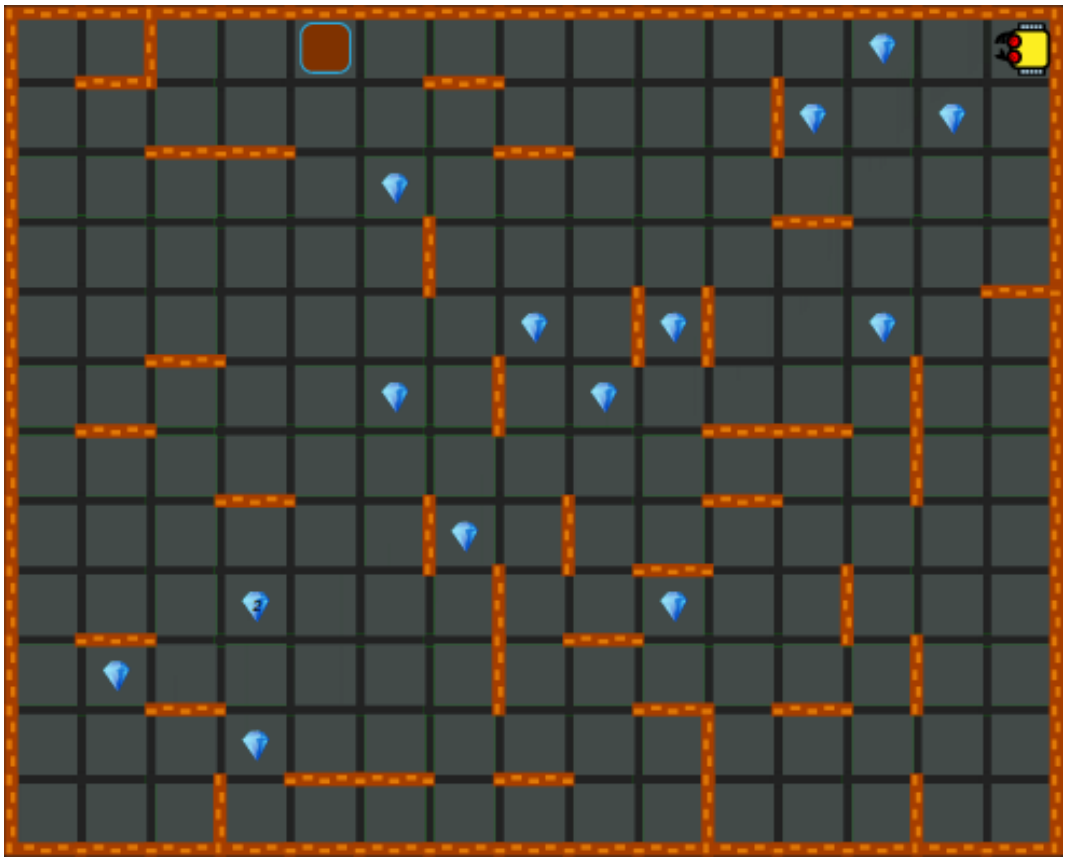


Fig. 34: North-east corner of the maze has GPS coordinates [14, 11].

then the output of the program is

```
Horizontal position: 14
Vertical position: 11
```

Try to move Karel to other parts of the maze via simple commands or in Designer, and run the program again, to make yourself familiar with how the GPS device works.

The `print` command can be used to display more complicated sentences where text and variables are separated by commas:

```
print "My GPS coordinates are", gpsx, "and", gpsy
```

Notice that each comma means the inclusion of one empty character:

11.4 Defining custom functions

In Level 3 we can use the reserved word `return` inside the body of a command to return a value. Such commands are then called *functions*. They are also defined using the keyword `def`. For example, the following function `count_steps` will return the number of steps the robot needed to make in order to reach the closest wall in the direction that he was facing:

```
def count_steps
    n = 0
    while not wall
        go
        inc(n)
    return n
```

Notice couple of things here:

- The variable `n` was created and initialized by `0` using `n = 0`. In Karel, we do not have to state the type of a variable in advance – the interpreter will figure it out from the type of value that is first assigned to it.
- The `inc()` function increases the value of an integer variable by one. There is also a function `dec()`, not used in the above program, that decreases the value of an integer variable by one. More about these two functions will be said in Paragraph 11.7.

The function can then be used as follows:

```
num = count_steps
print "Number of steps:", num
```

Here, we create a new variable `num` and initialize it using the integer value that is returned by the `count_steps` function. The result is then printed. For the situation shown in Fig. 35,

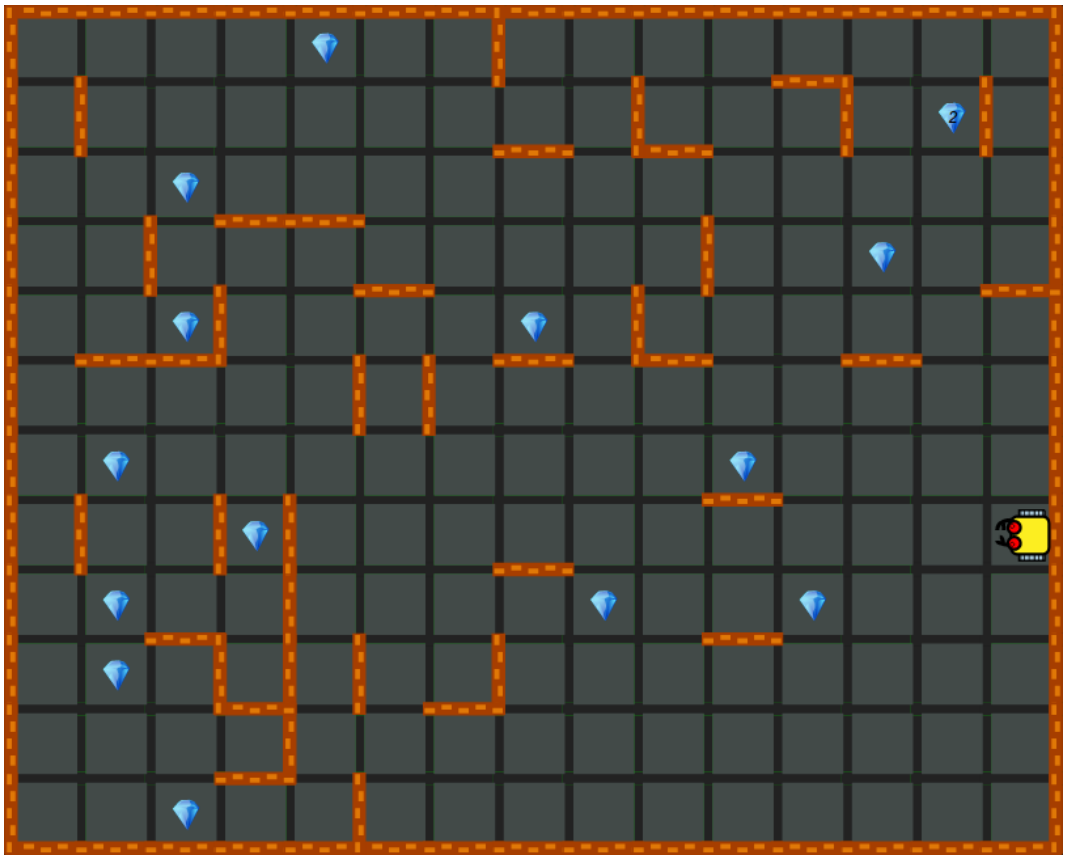


Fig. 35: Testing the function `count_steps`.

the output is

```
Number of steps: 10
```

11.5 Creating and initializing numerical variables

In Karel, numerical variables can be created and initialized in several different ways:

1. By setting them to an integer number. For example, new variable `a` is created and set to zero by typing

```
a = 0
```

2. By setting them to `gpsx`. For example, new variable `var` is created and set to `gpsx` by typing

```
var = gpsx
```

3. By setting them to `gpsy`. For example, new variable `pos1` is created and set to `gpsy` by typing

```
pos1 = gpsy
```

4. Initialize them with an existing value. For example, if there already is an integer variable `var1`, then a new variable `var2` can be created as follows:

```
var2 = var1
```

5. Initialize them with value returned by an existing function, as it was shown with the function `count_steps` in the previous paragraph:

```
num = count_steps
```

11.6 Changing values of numerical variables

The value of a numerical variable can be updated at any time by redefining it via one of the five options described in the previous paragraph. For example, let's say that Karel stands as shown in Fig. 36.

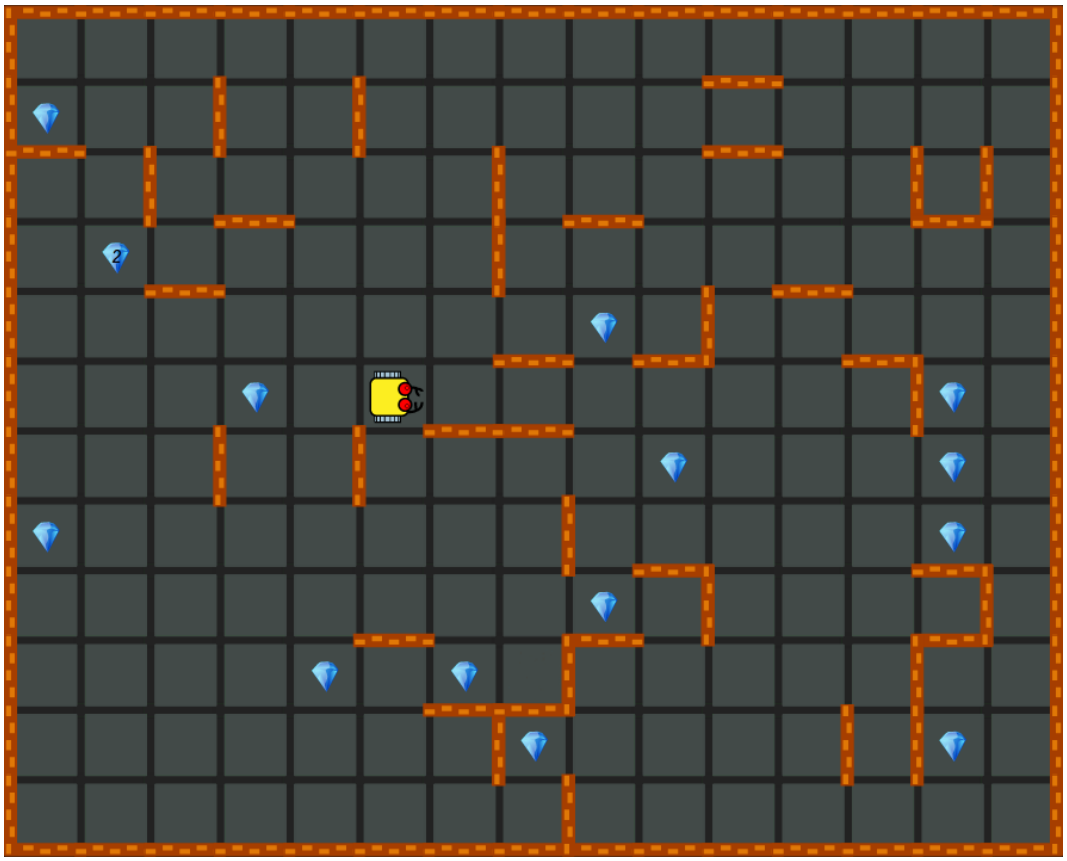


Fig. 36: Karel's initial position.

Then the program

```
a = gpsx
print "Start position:", a
repeat 5
    go
a = gpsx
print "End position:", a
```

will produce the following output:

```
Start position: 5
End position: 10
```

11.7 Using functions `inc()` and `dec()`

As we saw before, we can increase or decrease the value of a numerical variable by one using the functions `inc()` and `dec()`, respectively. The full version of these functions is `inc(n, num)` and `dec(n, num)` where `n` is the name of the variable and `num` is an integer number that is set to one by default.

Consider again Karel's initial position as shown in Fig. 36. Then the code

```
a = 0
while not wall
    go
    inc(a)
print "Went", a, "steps before reaching a wall."
```

will have the following output:

```
Went 7 steps before reaching a wall.
```

11.8 Comparison operations

Integer numbers and numerical variables can be compared using the standard operators `>`, `<`, `>=`, `<=`, `==`, `!=`, `<>`. In this order, they read "greater than", "less than", "greater than or equal to", "less than or equal to", "equal to", "not equal to" and "not equal to" (the last two operations have the same meaning). The result of each such operation is a logical value `True` or `False`, and it can be either used in a condition or conditional loop, or assigned to a variable. For example,

```
a = 1
b = 5
print a < b
```

yields the output

```
True
```

The code

```
a = 1
b = 5
c = a > b
print c
```

yields

```
False
```

The code

```
a = 0
while a <= 5
  print a
  inc(a)
```

yields

```
0
1
2
3
4
5
```

11.9 Text string variables

Text string variables are created and initialized analogously to numerical variables:

```
robot_name = "Karel"
```

They can be printed as usual. The code

```
print "Robot's name is", my_name
```

yields

```
MRobot's name is Karel
```

A text string variable can be used to initialize a new one. Let's say that someone wants to rename the robot to "Carlos" (which is the Spanish version of "Karel"), and store his original name in the variable `robot_name_orig`. This is done as follows:

```
robot_name_orig = robot_name
robot_name = "Carlos"
```

Text string variables can be useful, for example, to store texts for error messages and warnings in larger programs. Then, they can be defined in one place, although they are used in many different parts of the program. When we need to change such error message, we can do it very elegantly where the variable is defined, without having to go through the entire program and change the text everywhere.

11.10 Local and global variables

Note that a variable that is defined inside a function is *local to that function*, meaning that it can be used in the function only. If we attempt to use it outside, an error is thrown. For illustration, the code

```
def myfunction
  a = 1

myfunction
print a
```

throws an error message

```
Unknown variable/procedure "a"
```

Although using local variables might seem constraining, in reality it helps us to keep our code fit. It is a very good habit to keep variables in our programs as local as possible. More about local and global variables will be said in the Python textbook.

12 Lists

In this Section we introduce the concept of a *list*. The syntax and philosophy is the same as in the Python programming language, so all you learn here is directly usable in Python. The Python language provides additional functionality for working with lists which is not covered in Karel.

12.1 Objectives

- Understand the concept of a list.
- Learn to create empty lists, add items, and delete items.
- Learn to parse lists and work with indices.

Lists are very useful data structures that can be used to store multiple integer values, logical values, or text strings at the same time. Values of different types can be combined and lists can even contain other lists. Objects in a list are ordered and they can be added to the end of a list, accessed by their index, and deleted from any position of the list. Let us illustrate this functionality on examples.

12.2 Creating a list

An empty list `U` is created via

```
U = []
```

Lists can be also created non-empty:

```
V = [1, 2, 3, 4, 5]
```

One can use variables when creating a list:

```
c = 100
W = [0, 50, c]
print W
```

whose output is

```
[0, 50, 100]
```

Integer numbers can be combined with text strings:

```
X = [1, "Hello", 2]
```

Lists can contain other lists as their elements:

```
Y = [1, "Hello", 2, [1, 2, 3]]
```

One can print a list as expected:

```
print "This is the list Y:", Y
```

Output:

```
This is the list Y: [1, "Hello", 2, [1, 2, 3]]
```

12.3 Accessing list items by their index

Any list item can be accessed and either printed, assigned to a variable, or used in an operation, via its index. Indices always start from zero. In other words, `L[0]` is the first item in the list `L`, `L[1]` is the second one, etc. Working with indices can be illustrated using a simple code

```
L = [8, 12, 16, 20]
print "First item:", L[0]
print "Second item:", L[1]
print "Third item:", L[2]
print "Fourth item:", L[3]
```

whose output is

```
First item: 8
Second item: 12
Third item: 16
Fourth item: 20
```

12.4 Appending items to a list

An arbitrary object `obj` (integer, text string, logical value, another list, etc.) can be appended to the end of an existing list using the `append()` function. For a list `L` this would be

```
L.append(obj)
```

For illustration, the code

```
K = [1, 11]
K.append(21)
print K
```

has the output

```
[1, 11, 21]
```

12.5 Removing items via the `pop()` function

The `i`th item (where indices start from zero) can be deleted from a list `L` and assigned to a variable `x` via

```
x = L.pop(i)
```

For illustration, let us create a list `X` containing three text strings "Monday", "Tuesday" and "Wednesday", and then delete the second item:

```
X = ["Monday", "Tuesday", "Wednesday"]
day = X.pop(1)
print day
print X
```

The output of this code is

```
Tuesday
['Monday', 'Wednesday']
```

If the `pop()` function is used without an index, it removes and returns the last object of the list:

```
day = X.pop()
print day
print X
```

Output:

```
Wednesday
['Monday']
```

12.6 Deleting items via the `del` command

The purpose of the `del` command is similar to the `pop()` function except that the deleted object is destroyed (it cannot be assigned to a variable). The `i`th item can be deleted from a list `L` via

```
del L[i]
```

For illustration, the output of the code

```
L = ["Monday", "Tuesday", "Wednesday"]
del L[0]
print L
del L[0]
print L
```

is

```
['Tuesday', 'Wednesday']
['Wednesday']
```

12.7 Length of a list

The function `len(X)` returns the length of the list `X`. For illustration, the code

```
M = ["John", "Josh", "Jim", "Jane"]
n = len(L)
print "Length of the list is", n
```

has the output

```
Length of the list is 4
```

12.8 Parsing lists

In Karel, lists can be parsed via the `repeat` command. For illustration, the following sample code defines a list `M` consisting of four numbers `1, 2, 3, 4` and prints all of

them increased by two:

```
M = [1, 3, 5, 7]
n = len(L)
i = 0
repeat n
    c = M[i]
    print inc(c, 2)
    inc(i)
```

12.9 Storing the robot's path in a list

Lists can be used to store the robot's path. The following is a simple program that only tells the robot to go straight to the nearest wall (this part does not really matter) and store the GPS coordinates in a list `L`:

```
L = []
while not wall
    L.append([gpsx, gpsy])
    go
L.append([gpsx, gpsy])
print "Path: ", L
```

In fact, each time we are appending a list consisting of the horizontal and vertical GPS coordinates – there is no other way to represent two numbers in one variable unless the variable is a list. When the robot's initial position is as shown in Fig. 37,

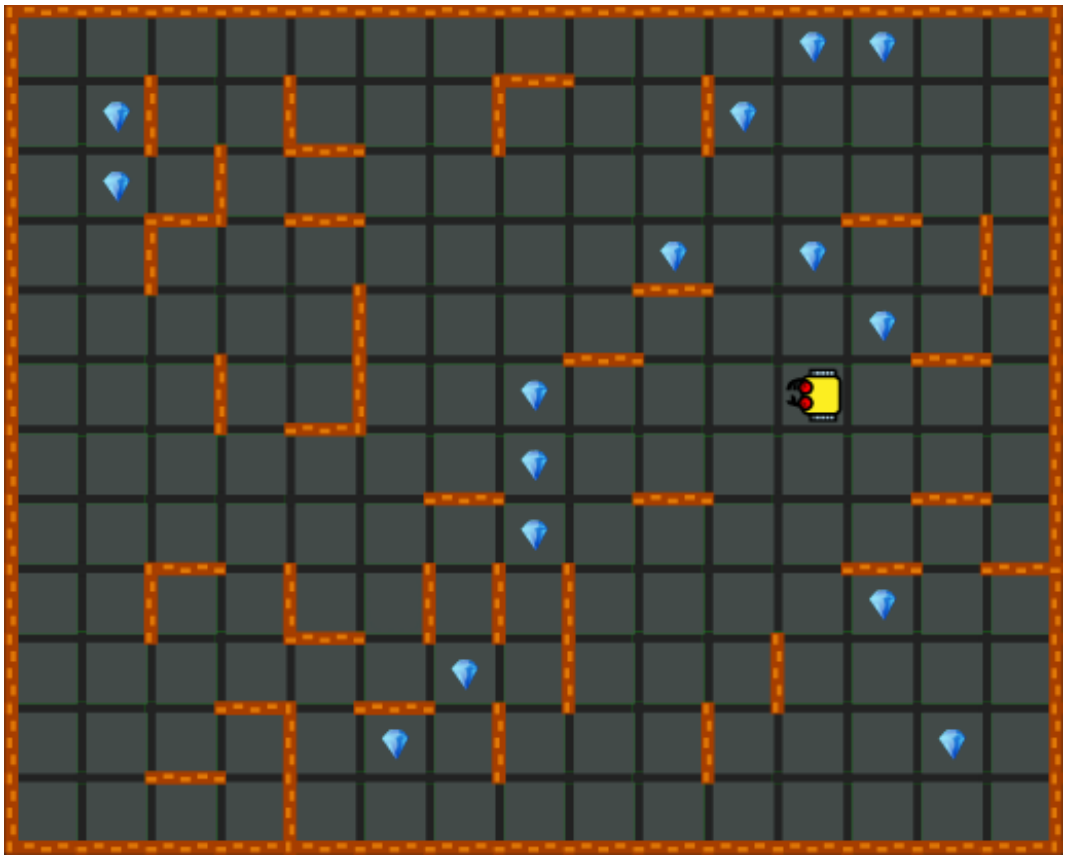


Fig. 37: Robot stands at position [11, 6] facing west.

then the output of the above program is

```
Path: [[11, 6], [10, 6], [9, 6], [8, 6], [7, 6], [6, 6],
[5, 6]]
```

To keep Karel's language simple, we do not allow double indices. Nevertheless there is a simple way to get to the items of lists that are contained in other lists:

```
a = L[0]
print "Horizontal coordinate of initial position:", a[0]
print "Vertical coordinate of initial position:", a[1]
a = L[1]
print "Horizontal coordinate after one step:", a[0]
print "Vertical coordinate after one step:", a[1]
...
```

13 Logic and Randomness

13.1 Objectives

- Review elementary logic.
- Practice working with more complex logical expressions.

13.2 Simple logical expressions

As we already know, logical expressions are expressions that can be answered with either `True` or `False`. We say that the `True` or the `False` is their *value*. Here are some real-life examples, try to answer them with `True` or `False`:

- "I am 15 years old."
- "My dad is a teacher."
- "My school's name is Coral Academy."

And here are some Karel examples:

- `wall` ... `True` if the robot is facing a wall, `False` otherwise.
- `gem` ... `True` if the robot stands on a gem, `False` otherwise.
- `north` ... `True` if the robot is facing North, `False` otherwise.
- `home` ... `True` if the robot is home, `False` otherwise.
- `empty` ... `True` if the robot does not have any gems on him, `False` otherwise.

Let's consider the situation shown in Fig. 38.

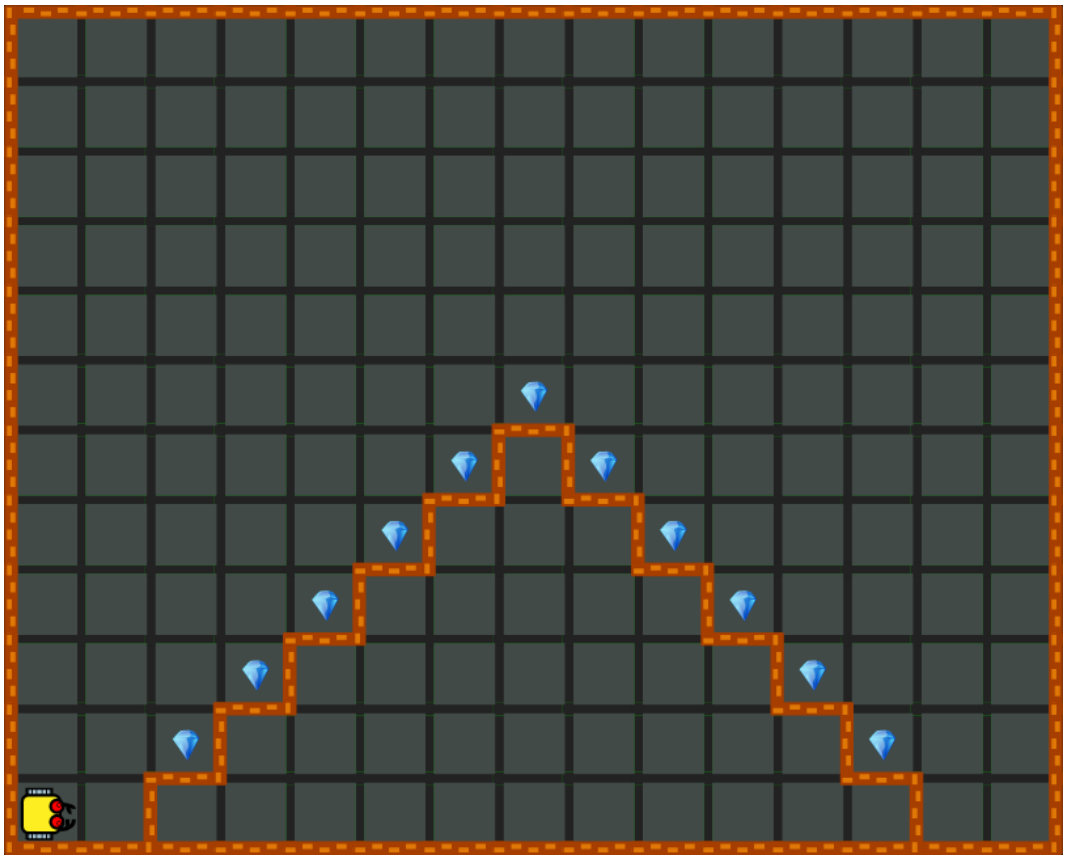


Fig. 38: Karel is testing the `wall` sensor.

The following code shows different ways we can operate with the logical value provided by the `wall` sensor:

```
wall
print "Value of the wall sensor:", wall
value = wall
if value
    print "I am standing in front of a wall."
else
    print "There is no wall in front of me."
```

The output of this code is

```
False
Value of the wall sensor: False
There is no wall in front of me.
```

13.3 Complex logical expressions

In programming as well as in real life we often deal with logical expressions that are fairly complex. Often we use two or more simple logical expressions in one sentence, and moreover combine them with logical operations `and`, `or` or `not`.

For example, the sentence "I will go skiing on Saturday if weather is good and if Michael goes as well." includes three simple logical expressions. Let's call them for brevity

`A` = "I will go skiing on Saturday."
`B` = "The weather is good."
`C` = "Michael goes as well."

There is a logical operation `and` between the expressions `B`, `C`. The original sentence can be written briefly as

if `B and C` then `A`

We love this kind of brevity in programming. Let's say that we can predict the future and we know that the weather will be good and that Michael will go as well. Then we can translate the above condition into computer code. We also print the resulting value of `A` at the end:

```
# Let's say that weather will be good
# and Michael will join you:
B = True
C = True
# This is how you decide:
if B and C
    A = True
else
    A = False
# Print the result:
if A
    print "I will go ski on Sunday."
else
    print "I will not go ski on Sunday."
```

Output:

```
I will go ski on Sunday.
```

13.4 Truth tables

Each of these three logical operations comes with its own *truth table* that summarizes its results for different values of operands. The truth table for the logical `and` is:

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

The truth table for logical `or` is:

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

And finally, truth table for logical `not` is:

A	not A
True	False
False	True

13.5 Making random decisions

In Level 3 Karel can make random decisions via the command `rand` that returns with equal probability either `True` or `False`. Typical usage of this command is

```
if rand
    do_something
else
    do_something_else
```

For illustration, here is a nice short program that makes Karel walk randomly through the maze and look for gems:

```
# This is a simple way to
# create an infinite loop:
while True
    # Make a random number of steps forward:
    while rand
        if not wall
            go
        if gem
            get
    # Turn either right or left:
    if rand
        right
    else
        left
```

After some time (it may take a while) the robot reaches all reachable gems and collects them! The program is in fact an infinite loop, so you will have to eventually stop it via the red button.

14 Appendix - Overview of Functionality by Level

14.1 Level 0 (Section 3)

In Level 0, Karel can be guided by clicking on buttons:

- Go ... make one step forward.
- Left ... turn left.
- Right ... turn right.
- Put ... put a gem on the ground.
- Get ... pick up a gem from the ground.

14.2 Level 1 (Section 4)

In Level 1, Karel can be guided by typing the commands:

- `go` ... make one step forward.
- `left` ... turn left.
- `right` ... turn right.
- `put` ... put a gem on the ground.
- `get` ... pick up a gem from the ground.

14.3 Level 2 (Sections 5 – 10)

New commands:

- `if - else` ... condition.
- `repeat` ... counting loop (repeat an action a given number of times).
- `while` ... conditional loop (repeat an action while a condition is satisfied).
- `def` ... define a new command.

Additional new keywords:

- `wall` ... sensor that checks True if the robot faces a wall.
- `gem` ... sensor that checks True if there is a gem within the robot's reach.
- `empty` ... sensor that checks True if the robot's bag with gems is empty.
- `home` ... sensor that checks True if the robot is at home.
- `north` ... sensor that checks True if the robot faces North.

New functionality:

- Recursion (a command can make a call to itself).

14.4 Level 3 (Sections 11, 13)

New keywords:

- `print` ... print strings and variables.
- `gpsx` ... GPS coordinate in the horizontal direction.
- `gpsy` ... GPS coordinate in the vertical direction.
- `a = 0` ... create a new variable `a` and initialize it with zero (or another integer number).
  For additional ways to initialize variables see Subsection 11.5.
- `inc(a)` ... increases the value of variable `a` by one.
- `inc(a, value)` ... increases the value of variable `a` by `value`.
- `dec(a)` ... decreases the value of variable `a` by one.
- `dec(a, value)` ... decreases the value of variable `a` by `value`.
- `rand` ... random command (returns randomly `True` or `False`).
- `return` ... returns a value, to be used in functions.
- `and` ... binary logical *and*.
- `or` ... binary logical *or*.
- `not` ... unary logical *not*.
- `L[]` ... create an empty list `L`.
- `len(L)` ... length of list `L`.
- `L[i]` ... object at position `i` in list `L`. Note: `L[0]` is the first item in the list.
- `L.append(x)` ... appends `x` to the end of list `L`.
- `x = L.pop()` ... removes the last item of list `L` and assigns it to `x`.
- `del L[i]` ... removes from list `L` object at position `i`.

New functionality:

- Numerical and logical (Boolean) variables.
- Complex logical expressions.
- Functions that return values.
- Lists.

15 What next?

Congratulations, you made it through Karel the Robot! We hope that you enjoyed the textbook and exercises. If you can think of any way to improve the application Karel the Robot or this textbook, we would like to hear from you. If you have an interesting new game or exercise for Karel, please let us know as well.

You are now ready to dive into a next programming language! We would recommend Python which is a modern high-level dynamical programming language that is used in many diverse applications in business, science, engineering, and other areas.

In any case, our team wishes you good luck, and keep us in your favorite bookmarks!

Your Authors

Part II

Programming Exercises

3 Manual Mode

All exercises in this textbook are also available as Displayed Projects in NCLab. To clone them into your account, launch the File Manager, go to the menu File → Clone, and in the search box type "Karel".

3.1 First steps

Karel is returning from a long walk and his batteries are running out. Use the buttons on the left to get him home quickly!

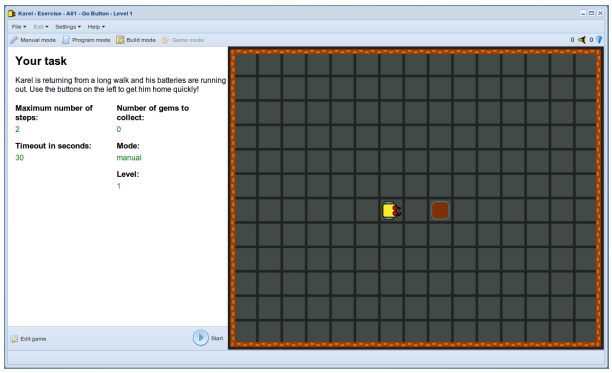


Fig. 39: Help the robot get home before his batteries run out!

Pressing Start will start the exercise, and at this time also the buttons Go, Left, Right, Put and Get appear, as shown in Fig. 40.

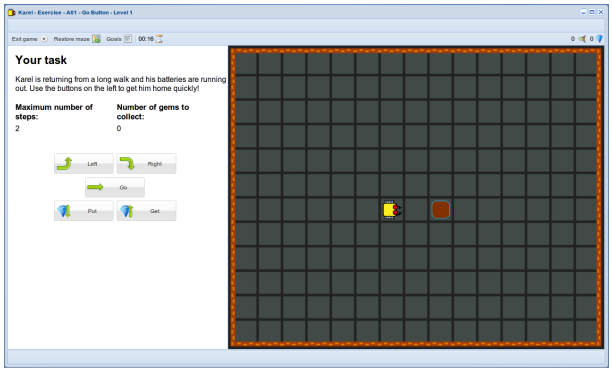


Fig. 40: Karel can be guided manually, using five buttons located in the left panel.

3.2 1st olympic games

Karel is training for Robolympic Games! Your task is to run with the robot home as fast as possible. Karel's personal record is four seconds. How fast can you be?

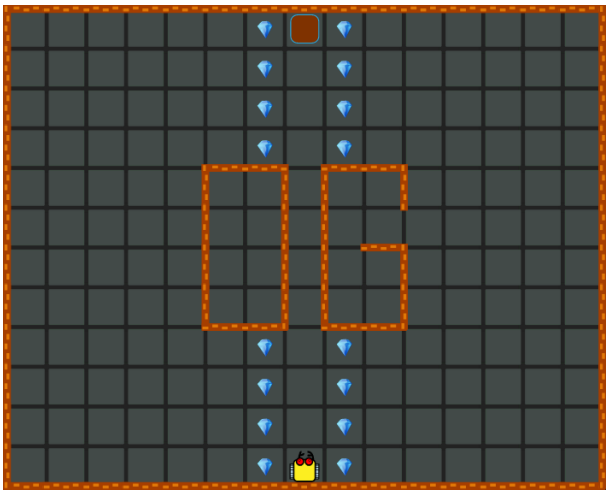


Fig. 41: Karel is training for Robolympic Games.

3.3 First gem

Today is Karel's lucky day because he is about to find his first gem. Use the buttons on the left to help the robot pick up the gem and carry it home!

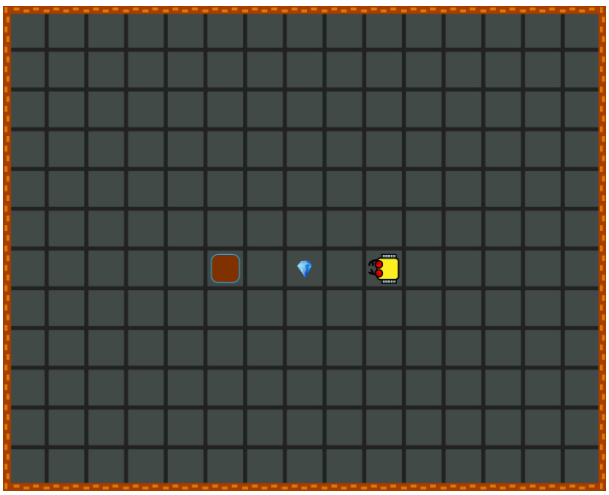


Fig. 42: Karel is about to find his first gem.

3.4 2nd olympic games

It is Robolympic season again! Run home as fast as you can, and collect all three gems on the way! Karel's personal record is 10 seconds.

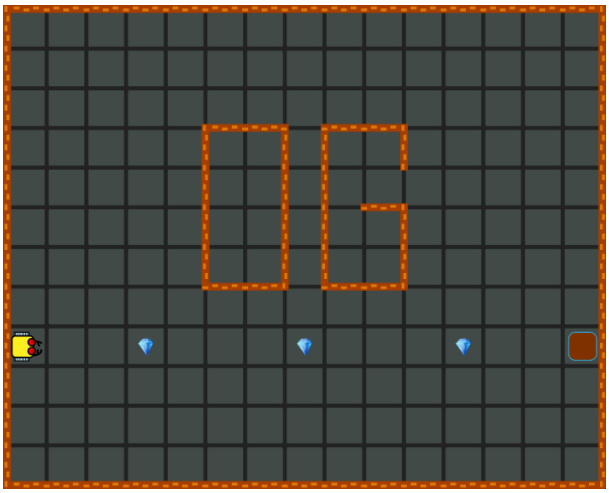


Fig. 43: Karel's second Robolympic Games.

3.5 First turn

Help the robot collect the gem and return home!

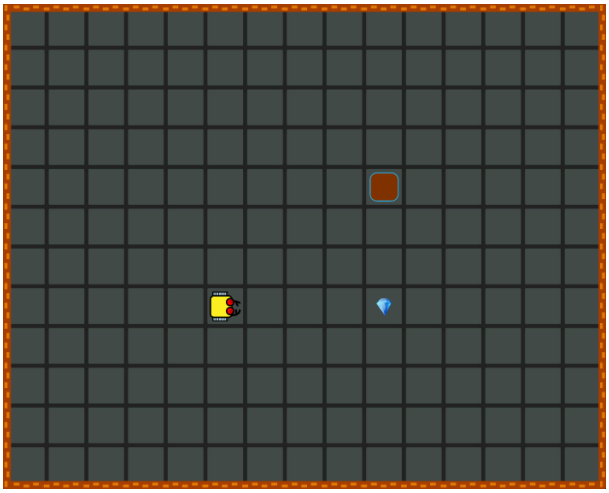


Fig. 44: Karel is about to learn how to make a left turn.

3.6 3rd olympic games

Karel is training for his third Robolympic Games. Run with him around the block and home as fast as possible. He needs to collect at least one gem on the way. Karel's personal record is 16 seconds!

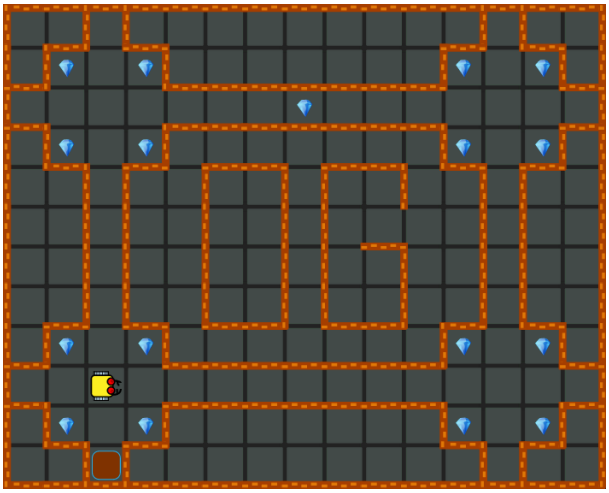


Fig. 45: Karel's third Robolympic Games.

3.7 Two gems

Pick up the two gems and get Karel home!

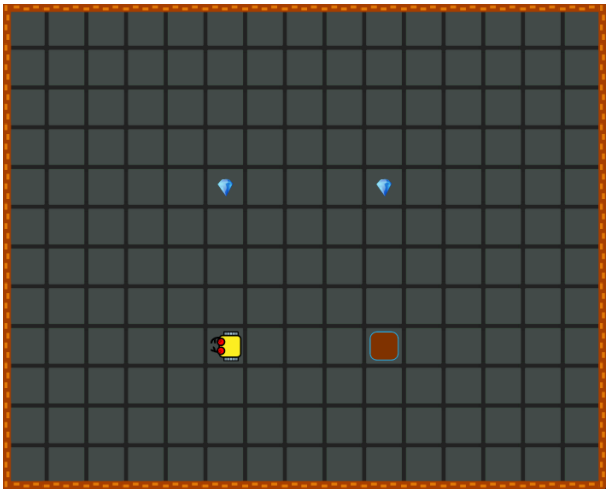


Fig. 46: Karel needs to collect two gems and get home.

3.8 4th olympic games

Last season of Karel's Robolympics Games is here! The robot needs to run home as fast as possible and bring one gem. Be careful not to crash, this is a tricky level! Karel's personal record is 26 seconds.

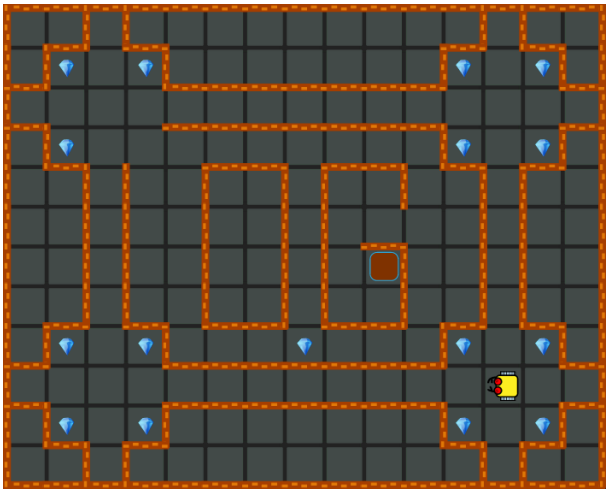


Fig. 47: Karel's fourth Robolympic Games.

3.9 Five gems

Help Karel collect five gems and get home!

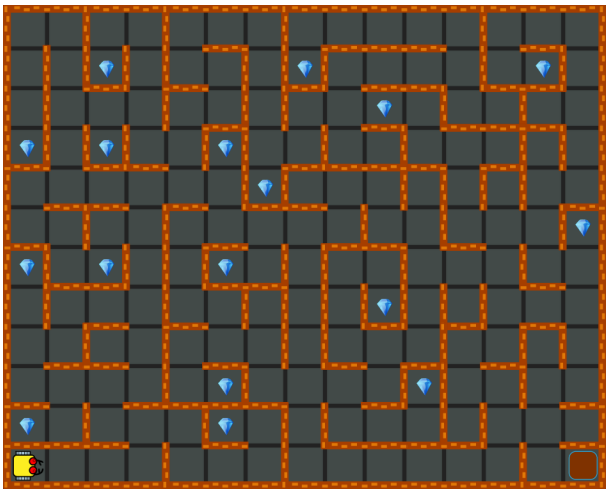


Fig. 48: Karel needs to collect five gems.

3.10 Labyrinth

This is a true labyrinth and your task is to lead Karel home. Remember - think first before going anywhere!

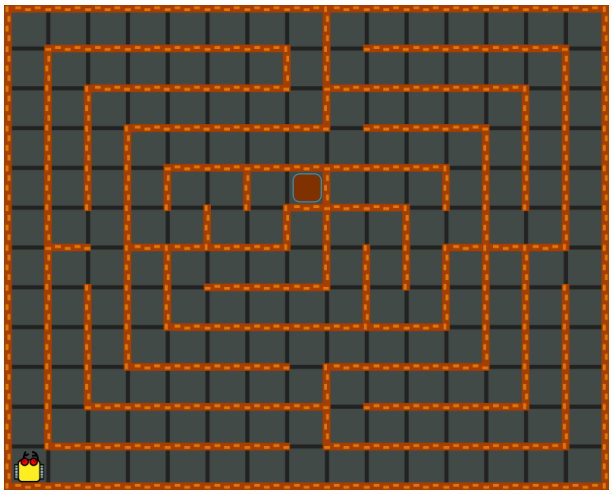


Fig. 49: Karel needs to find his way home in a labyrinth.

3.11 Diamond mine

Karel discovered an abandoned diamond mine. Use the buttons on the left to collect all gems and get back home in time!

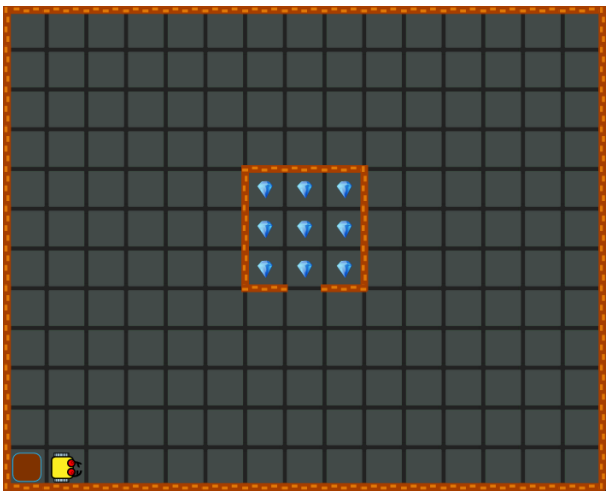


Fig. 50: Karel found an abandoned diamond mine.

3.12 Piles of gems

If gems are piled up, then a number is showing their amount. Help Karel collect all gems in this maze and return home!

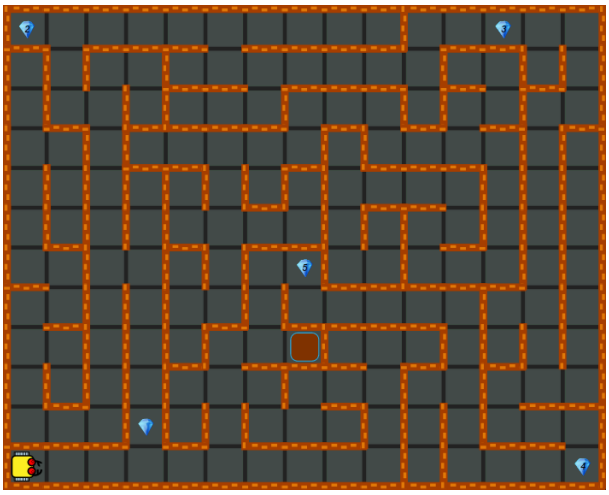


Fig. 51: If gems are piled up, then a number is showing their amount.

3.13 Gems on table

Karel has five gems in his bag. Use the buttons on the left to put the gems on the table and return home in time!

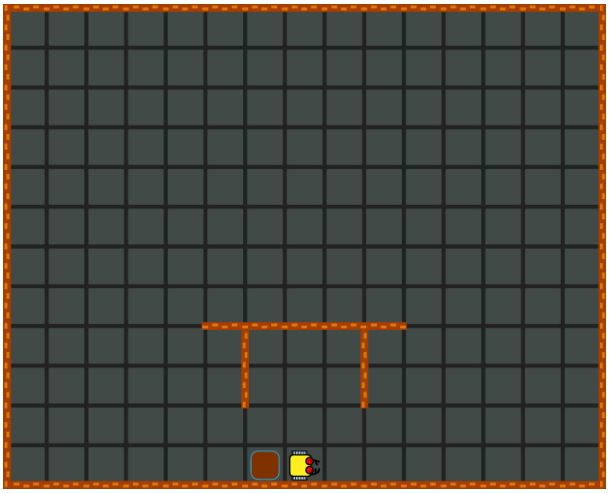


Fig. 52: Karel needs to put five gems on the table.

4 Programming Mode

4.1 First program

Write a program that gets Karel home! Remember: Always write one command per line.

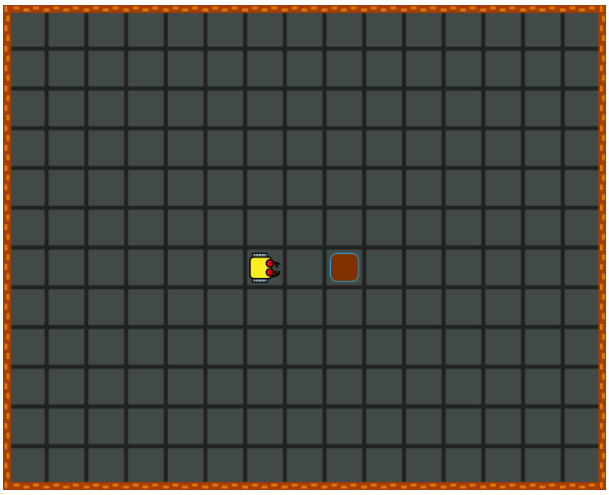


Fig. 53: Moving Karel forward via the `go` command.

4.2 Three gems

Write a program for Karel to collect all gems and get home!

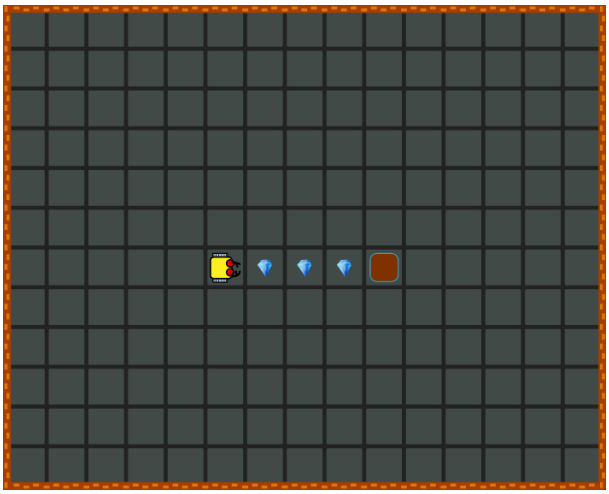


Fig. 54: Collecting gems using the `get` command.

4.3 Diagonal move

Write a program for Karel to collect the gem and return home!

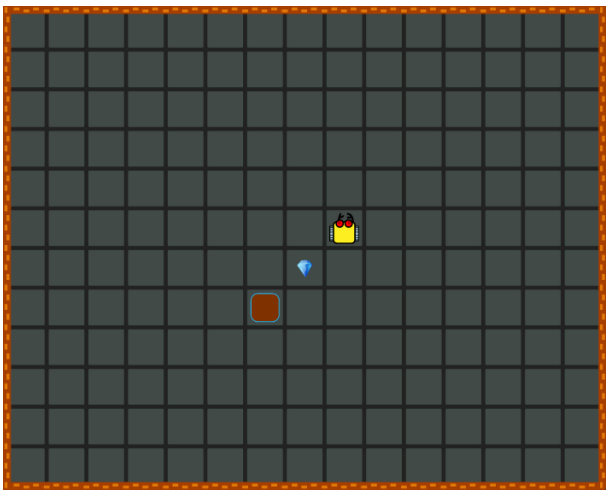


Fig. 55: Turning to the left and to the right via the `left` and `right` commands.

4.4 Five gems

Write a program for Karel to collect all five gems and get home!

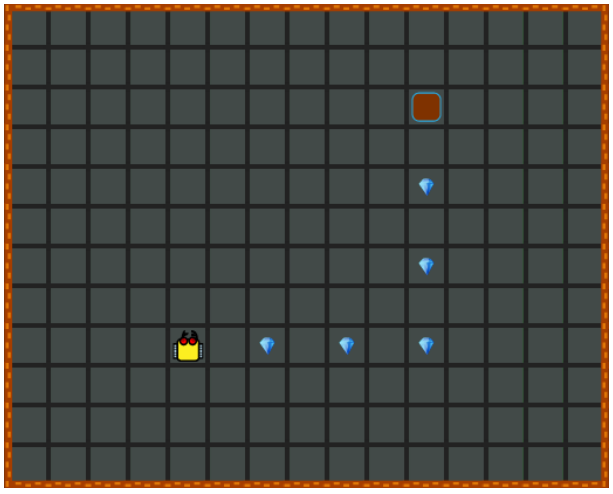


Fig. 56: Collecting five gems.

4.5 Four gems

Karel needs to collect four gems and return home!

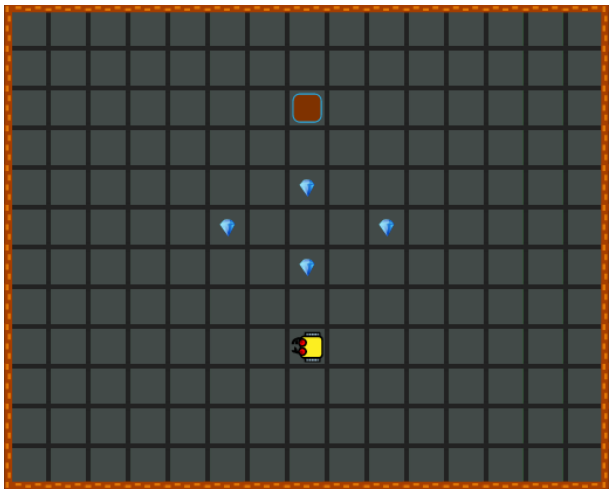


Fig. 57: Collecting four gems.

4.6 Three gems

Write a program for Karel to collect all three gems and return home!

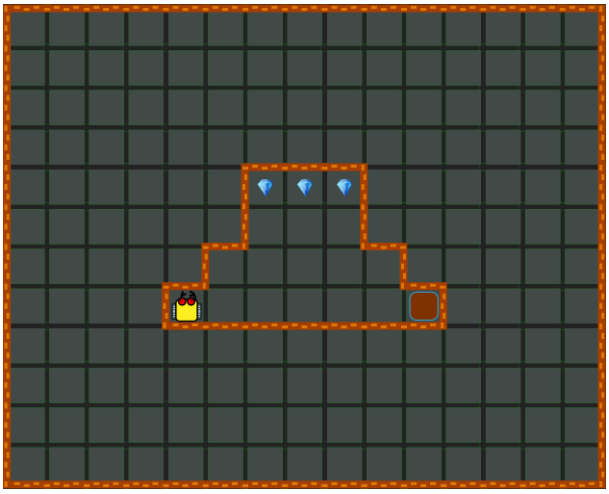


Fig. 58: Collecting three gems.

4.7 Cross

Write a program for Karel to relocate the gem to the opposite end of the cross and return home!

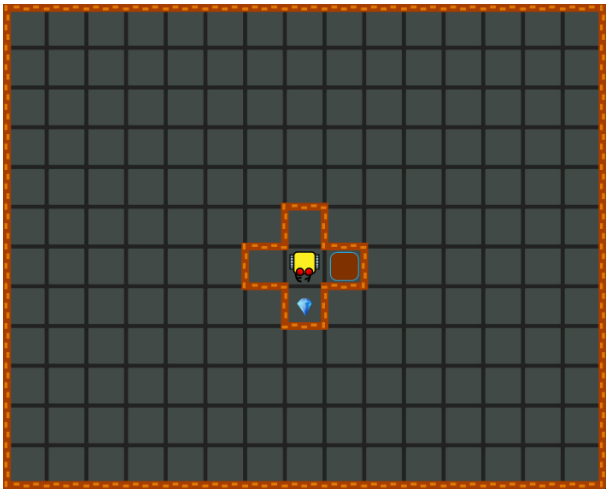


Fig. 59: Relocating a gem requires both `get` and `put` commands.

5 Counting Loop

5.1 Ten steps

Karel's home is ten steps away, so you could type `go` *ten times to get him there. However, knowing the* `repeat` *command, you can do this with only* **two lines** *of code!*

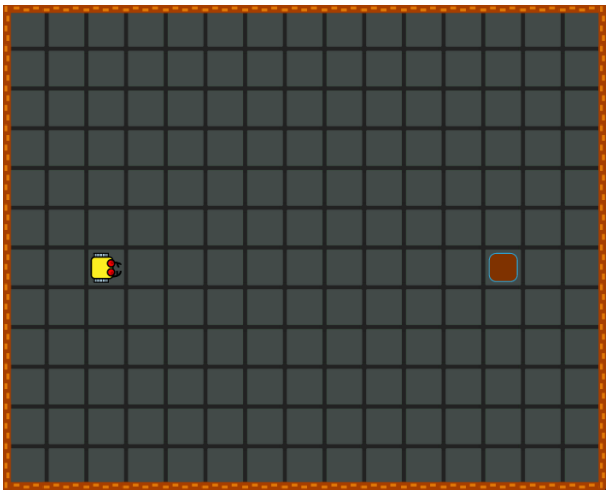


Fig. 60: Karel gets home elegantly, using only two lines of code.

5.2 Twelve gems

Karel found a pile of 12 gems! Write a program for the robot to collect all of them and get home. Your program should not be longer than **eight lines** *of code.*

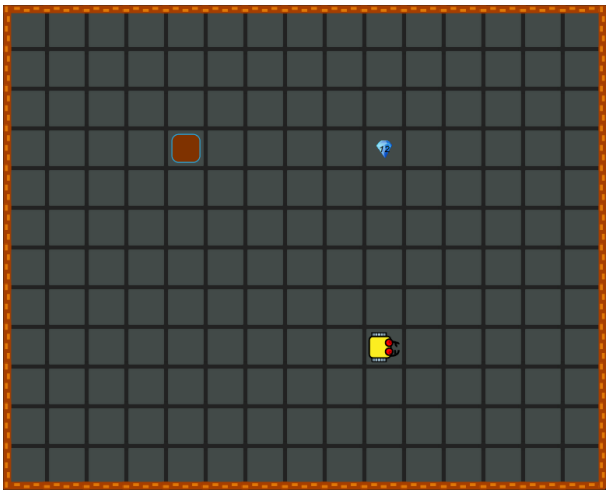


Fig. 61: Karel found a pile of 12 gems!

5.3 Feeling lucky

Karel is feeling lucky today. He wants to just step outside his house, turn around five times, then pick up one gem somewhere, and get back inside!

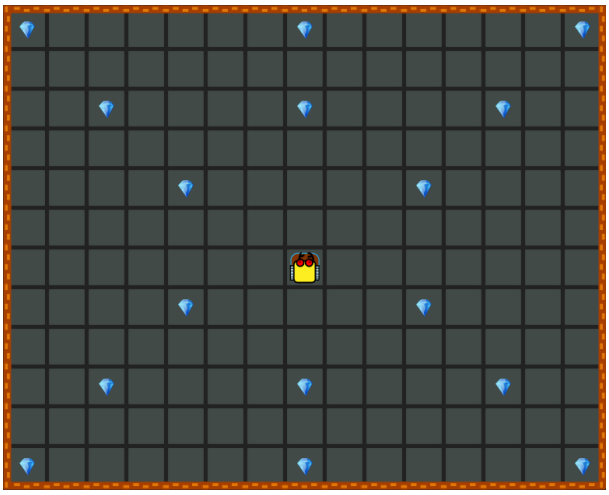


Fig. 62: Karel is feeling lucky today.

5.4 Garage sale

Karel needs to sell 10 of his oldest gems in order to make space for new ones. Write a program for the robot to step out of his garage, put 10 gems on the ground, and then turn back and get back inside! Your program should not be longer than **six lines** *of code.*

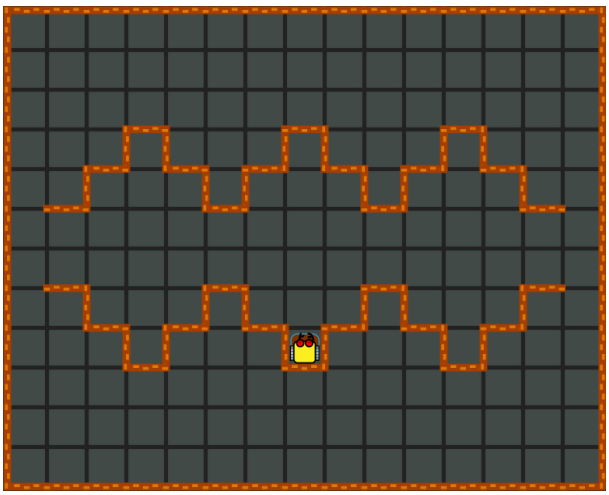


Fig. 63: Karel is getting ready for garage sale.

5.5 Diamond road

Write a program for Karel to collect all nine gems and get home! Writing one command per line, your program should not have more than **three lines** *of code.*

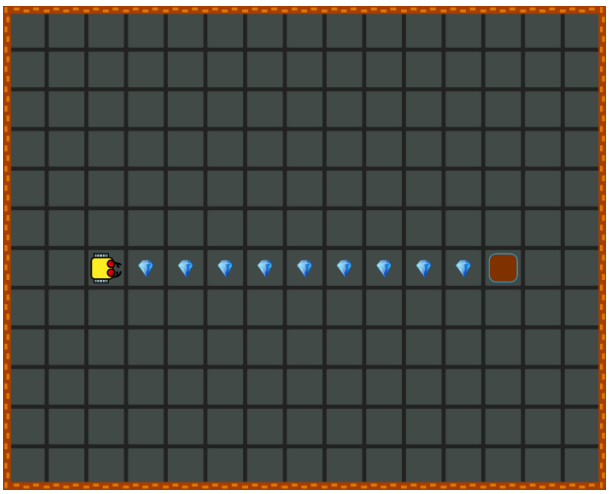


Fig. 64: Karel is going to collect nine gems.

5.6 Twenty gems

Each pile contains five gems. Write a program for Karel to collect all four piles and get back home. Writing one command per line, your code should not be longer that **seven lines**.

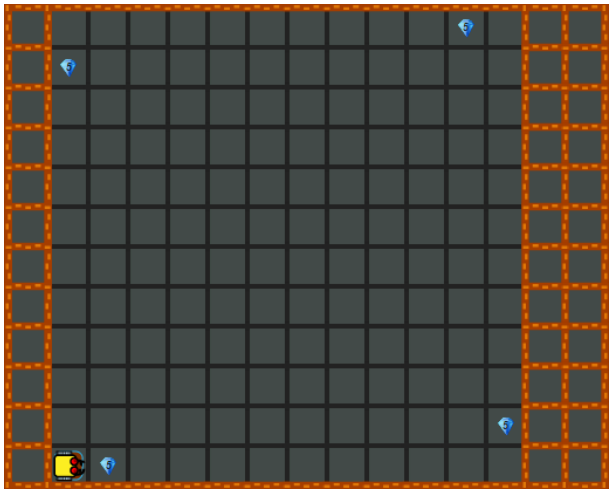


Fig. 65: Using nested counting loops to collect four piles of gems.

7 Conditions

7.1 Scattered gems

Several gems are at ramdom positions on the straight line connecting the robot and his home which is 10 steps away. There is at most one gem in each square. Write a program for Karel to collect all gems and get home. Your program should have at most **four lines**.

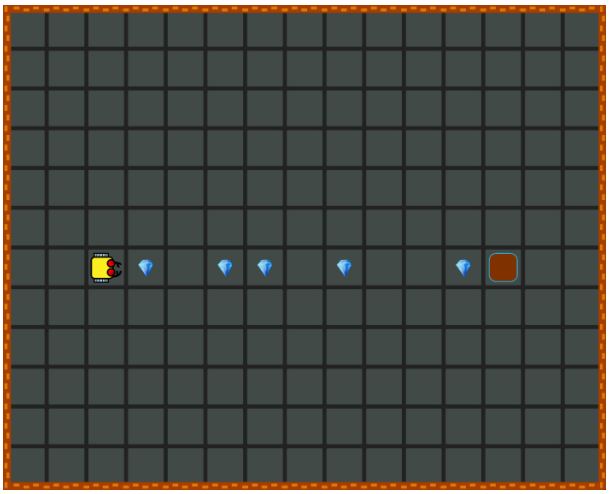


Fig. 66: Several gems are at random positions between the robot and his home.

7.2 Gems and stones

Karel is walking on a straight road that is covered with randomly placed stones and gems. His home is 14 steps away. Write a program for the robot to get home, avoiding stones and collecting all gems!

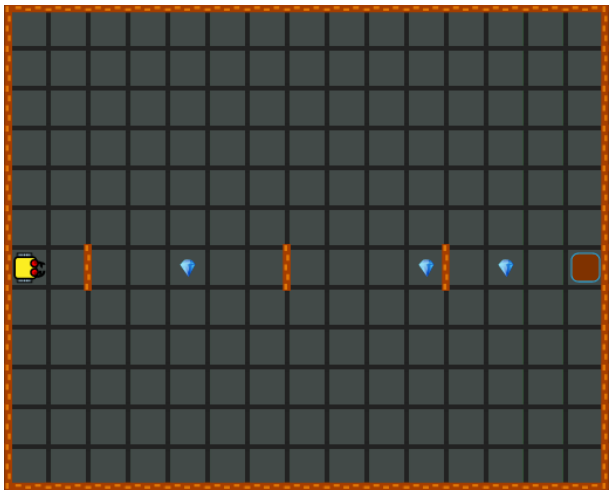


Fig. 67: Karel is walking on a stony road.

7.3 Secret chest

Karel stores all his gems in a secret chest in his cellar. Currently, some shelves are empty. Write a program for Karel to inspect all shelves and put a gem where one is missing! After that, he needs to get home as usual. Your program should have at most 7 lines.

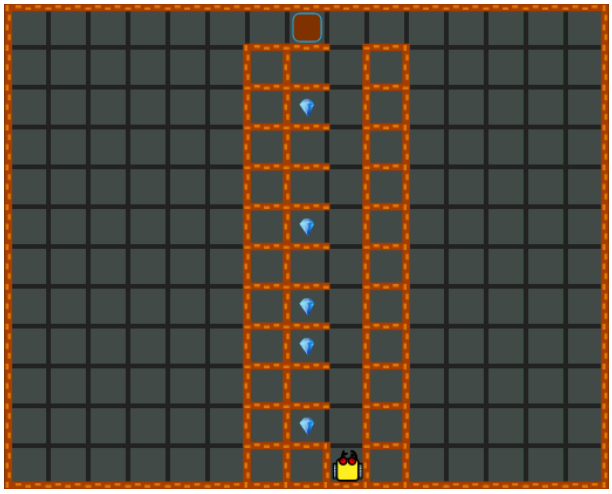


Fig. 68: Filling empty shelves with gems.

7.4 Cleaning shelves

There are 11 shelves and each one is 12 squares deep. The shelves contain random gems, always at most one gem per square. Write a program for Karel to collect all gems and return home!

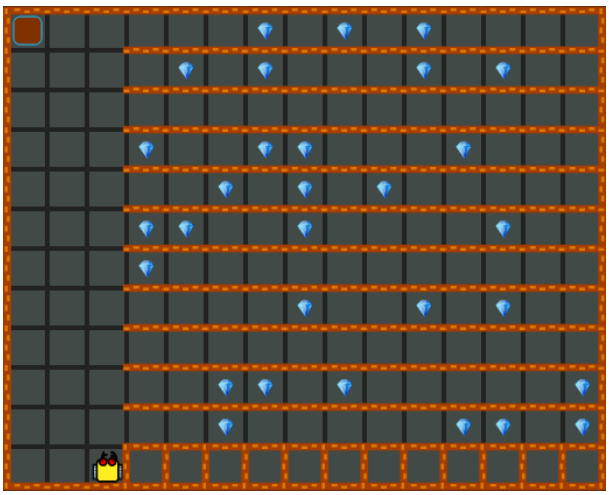


Fig. 69: Collecting gems from 11 shelves.

8 Conditional Loop

8.1 Southwest

Karel stands at a random position in the maze, facing a random direction. The maze is empty – no walls or gems. Write a program for the robot to get to his home which is in the south-west corner! Not counting comments, your program should have at most **nine lines**.

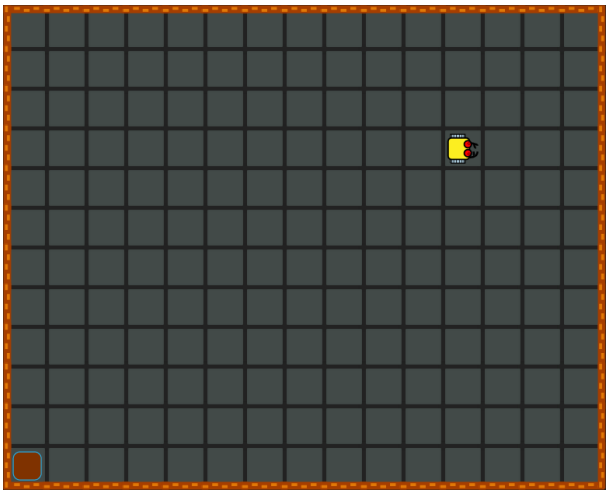


Fig. 70: Karel only knows that his home is in the SW corner of the maze.

8.2 Hide and seek

Karel's friends are hiding. To find them, he has to straight ahead and whenever he finds a gem, he has to collect it, turn left, and keep walking. Eventually, they said, he will get to the place where they are. Write a program for Karel to find his friends!

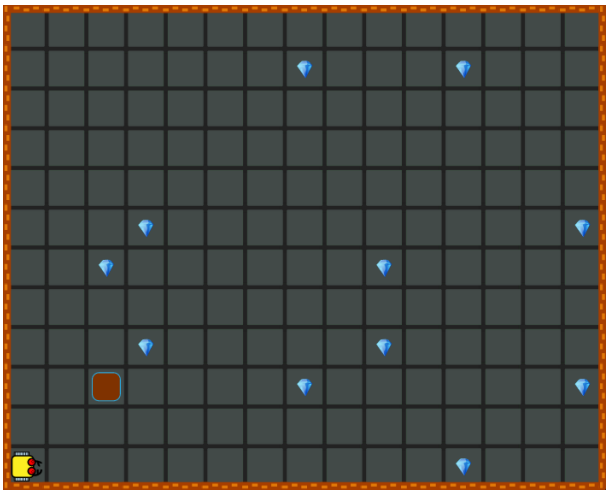


Fig. 71: Karel plays hide-and-seek with his friends.

8.3 Other side

Karel stands next to a straight wall of random length. The wall is not touching the border of the maze. The robot's home is somewhere on the other side of it but he does not know exactly where. Write a program for the robot to get there!

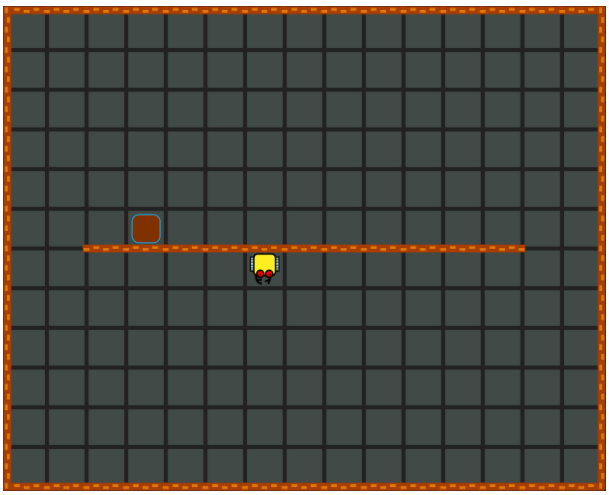


Fig. 72: Karel knows that his home is somewhere on the other side of the wall.

8.4 Spiral

Write a program for Karel to get home. Your program should not have more than **four lines**.

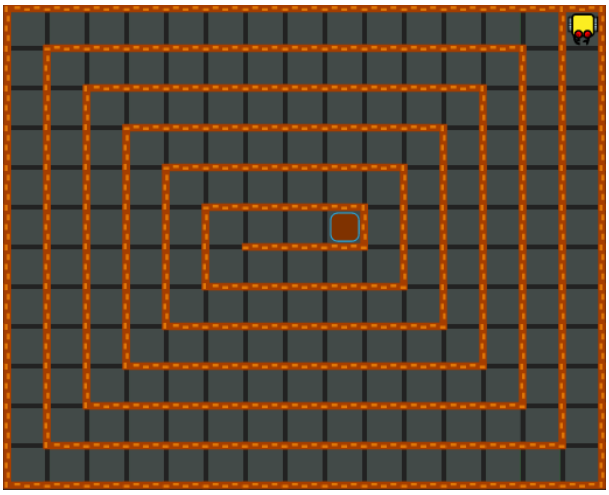


Fig. 73: This time, Karel's home is in a spiral maze.

8.5 Spiral II

Write a program for Karel to get home, collecting all the randomly placed gems on the way. There may be multiple gems in some squares. Your program should not have more than **six lines**.

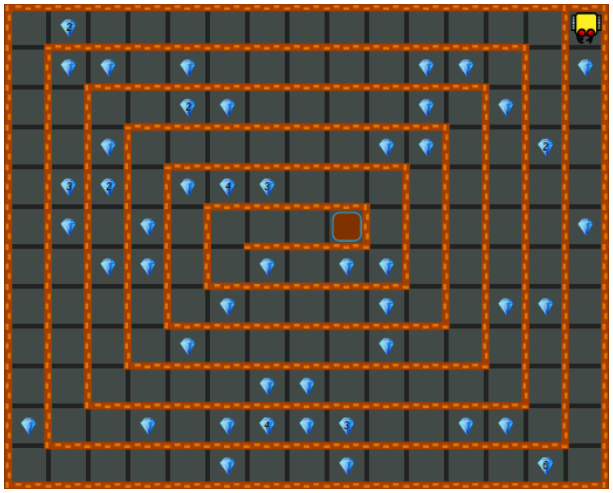


Fig. 74: Karel needs to get home, collecting all the gems on the way.

8.6 Skyline

Karel stands in the south-west corner of the maze, facing east. There is an array of randomly high poles and between some of them, there is a gem on the ground. Write a program for the robot to collect all gems and get home!

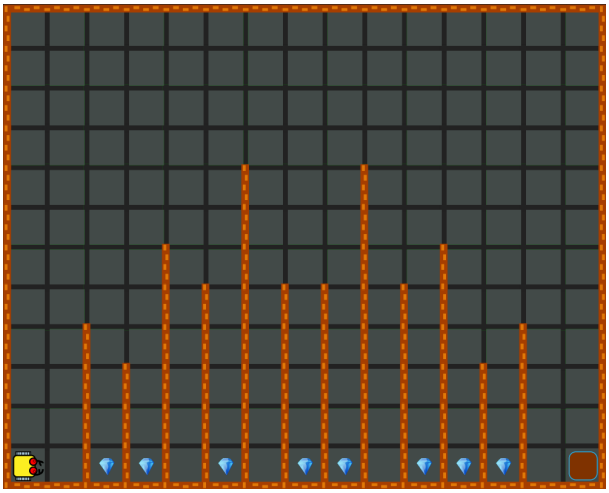


Fig. 75: This time Karel faces an array of randomly-high poles.

9 Custom Commands

9.1 Four stars

This time Karel has to collect four stars of gems!

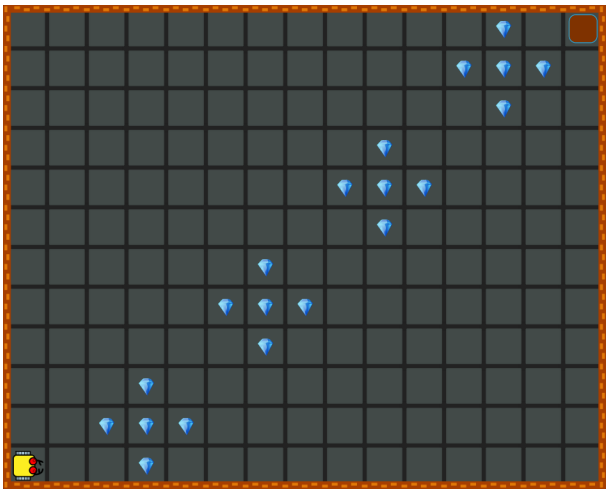


Fig. 76: Karel needs to collect four stars of gems.

9.2 Haul 36

Write a program for the robot to move the 6x6 square of gems to the opposite corner of the maze. The task is finished when the robot is back home.

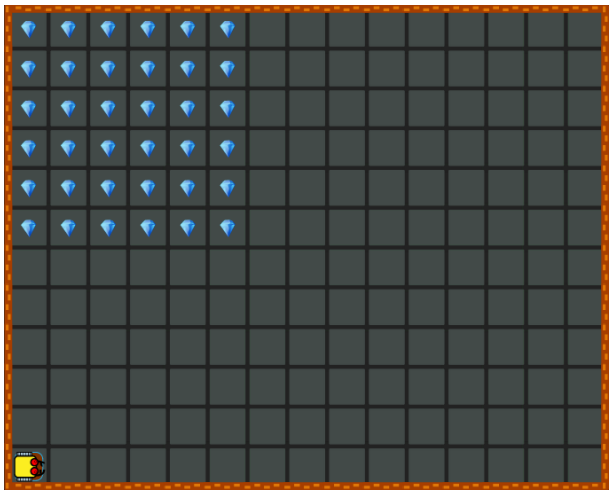


Fig. 77: Karel needs to move the gems to the opposite corner of the maze.

9.3 Egg hunt

Write a program for Karel to search all cells and collect all eggs (gems) that he can find. The task is finished when the robot is back home.

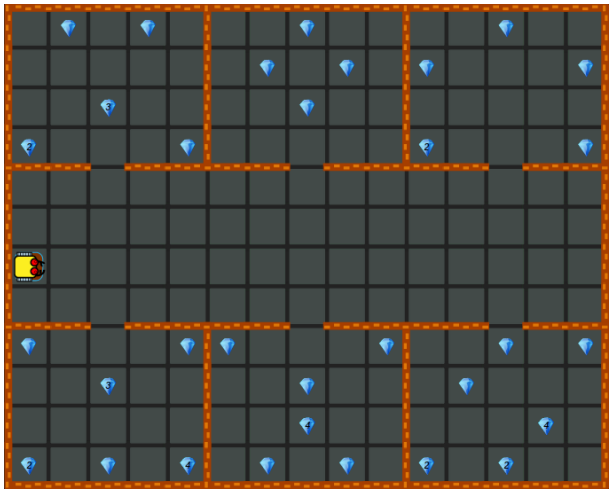


Fig. 78: Easter is here, Karel is on egg hunt!

9.4 Blind carpenter

Karel is a blind carpenter who needs to install windows (gems) into a newly built house. All he knows is that the house is a rectangle and that each window is exactly one tile large, But he can't see where the openings for the windows are. Install the windows and return home!

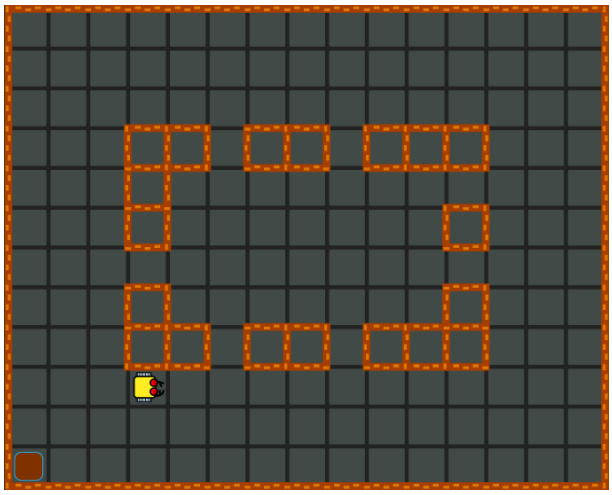


Fig. 79: This time Karel installs windows.

9.5 Pirate ship

Karel is on a pirate ship! Write a program for him to collect all gems and run away (to his home) before the pirates are back. Here Karel needs to be extremely efficient to survive. Therefore, there should be no repeating parts whatsoever in your program!

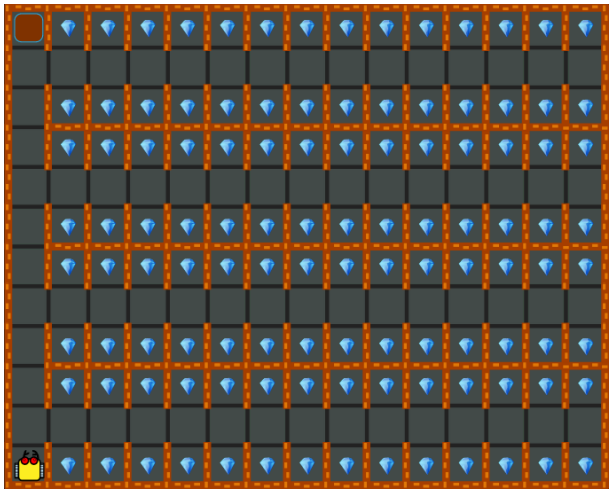


Fig. 80: Karel found a pirate treasure.

9.6 Diamond staircase

Write a program for Karel to climb the staircase, collect all gems, and get home. The number of steps in the staircase is random.

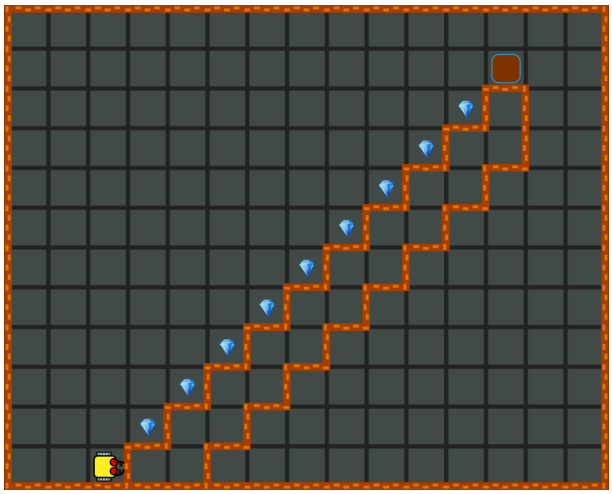


Fig. 81: This time Karel has to do some climbing.

9.7 Plucking flowers

Write a program for Karel to pluck all flowers that grow at the fence of his garden (gems), and get back home! Note two levels of repetition.

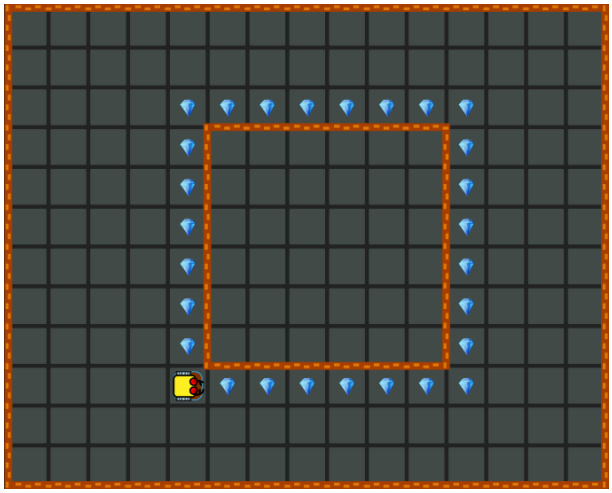


Fig. 82: Karel is plucking flowers along his fence.

9.8 Gems for friends

Karel has three gems in his bag, that he wants to give to his three friends R2, D2 and Marvin who live close by. Write a program for Karel to put a gem on the ground in the middle of each friend's home, and then return to his own home.

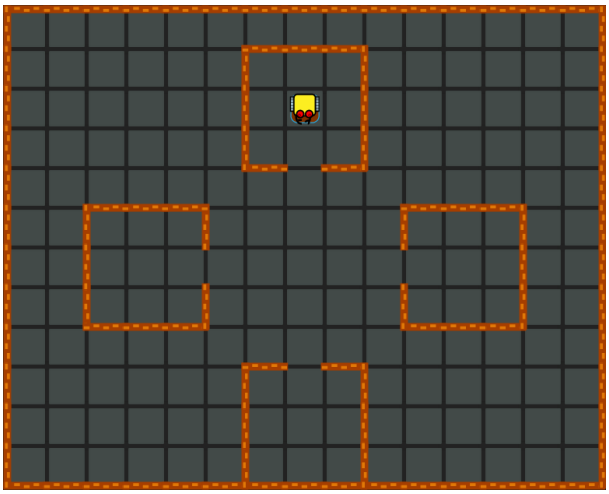


Fig. 83: Karel is going to give gems to his three friends R2, D2 and Marvin.

9.9 Diamond rectangle

Karel stands in front of a diamond rectangle of unknown dimensions. Write a program for the robot to walk around the rectangle, collect all gems, and get home!

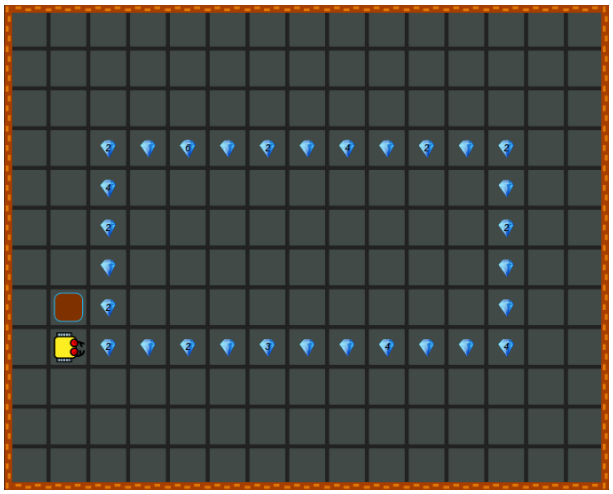


Fig. 84: This time Karel does not know the size of the rectangle.

9.10 Gem jam

In this maze, gems are distributed randomly along the walls. Otherwise the maze is empty. Karel's home is in the south-west corner, and the robot stands on the right of it, facing east. Write a program for Karel to collect all the gems and return home!

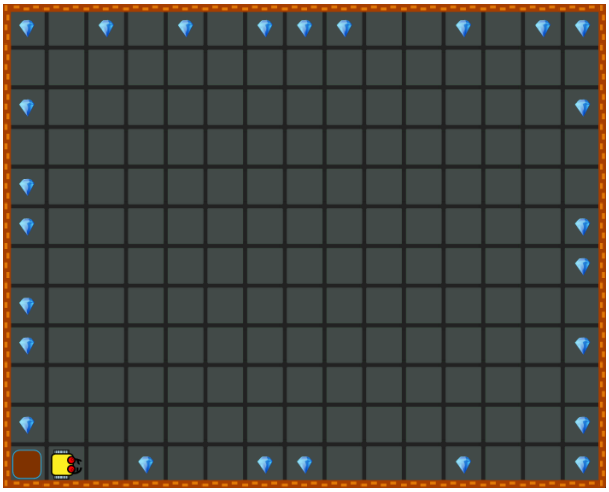


Fig. 85: Gem Jam!

9.11 The matrix

Write a program for Karel to collect all gems and enter his home!

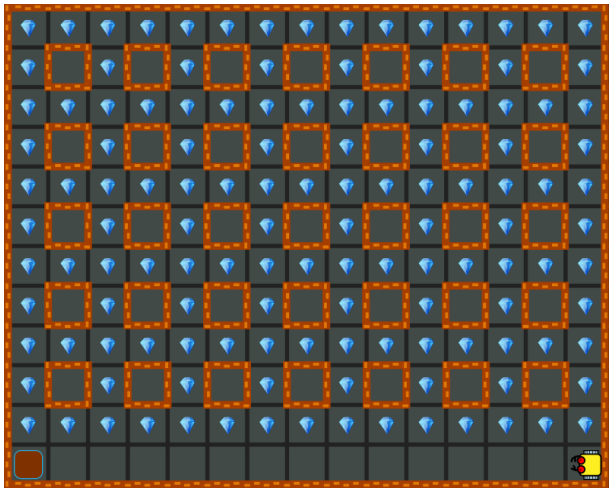


Fig. 86: The matrix.

9.12 Alcatraz

Karel is escaping from the Alcatraz prison! At the moment he is in an underground labyrinth with many random tunnels but only one of them leads to his freedom. Use the right-hand rule to find your way out!

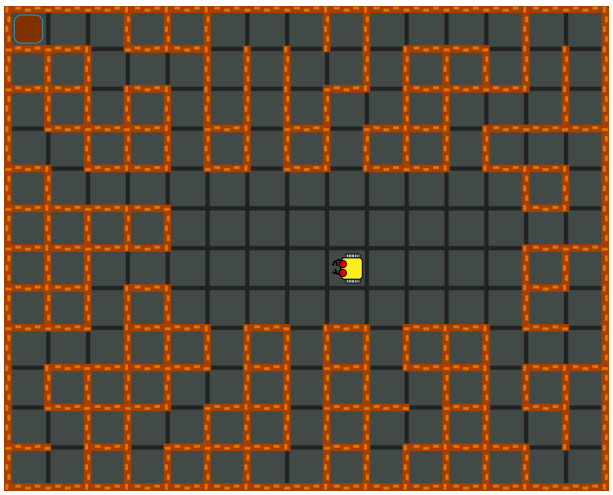


Fig. 87: Karel is escaping from the Alcatraz prison.

9.13 Border patrol

Write a program for Karel to check the perimeter of the maze using the right-hand rule. Do not forget to pick up all gems that you find on the way.

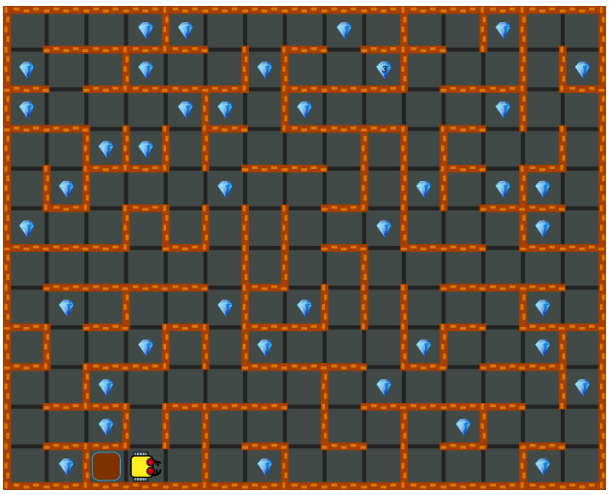


Fig. 88: Border Patrol.

9.14 Ariadne's thread

In an ancient Greek legend, princess Ariadne saved the life of her beloved Theseus by giving him a thread that he used to avoid getting lost in the maze of king Minos and kill a feared beast Minotaurus. Karel uses a similar technique - he leaves behind him a chain of gems that helps him to safely find his way back home. Your program needs to work for an arbitrary maze. You can assume that the string of gems is continuous and that it does not contain any loops.

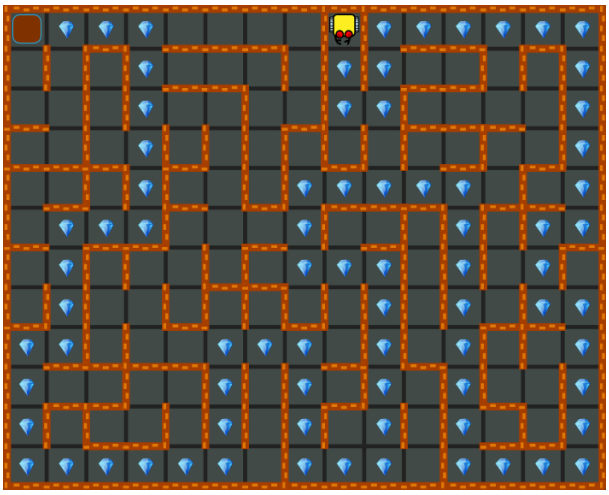


Fig. 89: Maze of king Minos, home of the beast Minotaurus.

10 Recursion

10.1 Cheese please

Karel's favorite way to eat cheese is to peel one edge at a time. His brick of cheese is represented by a rectangle of gems. The rectangle has random dimensions and the robot's initial position is at one of the corners, as shown in Fig. 90. Write a recursive algorithm for the robot to eat all the cheese and return home!

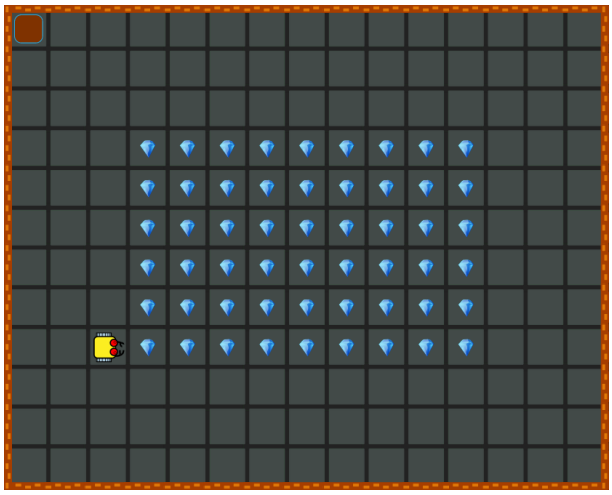


Fig. 90: Karel is eating cheese.

10.2 Speleologist

Karel stands at the entrance to a cave and wants to explore it. Write a recursive program for the robot to descend the staircase to the bottom, return back, and enter his home! The number of steps in the staircase is random.

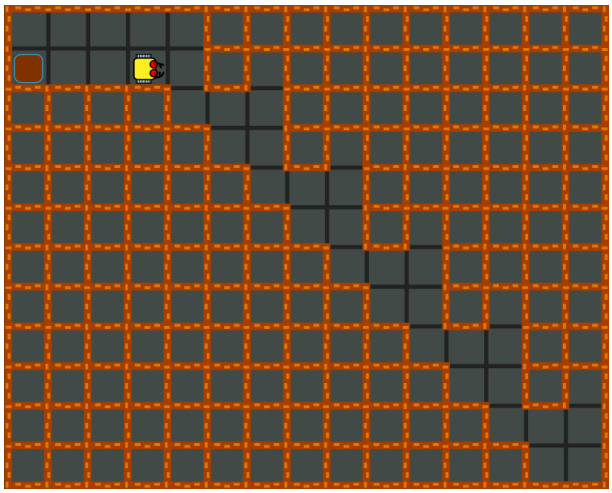


Fig. 91: Karel became a speleologist.

10.3 Homage to lemmings

Karel has many gems in his bag, and he needs to build a heap shown in Fig. 92 before he can enter his home. Use recursion.

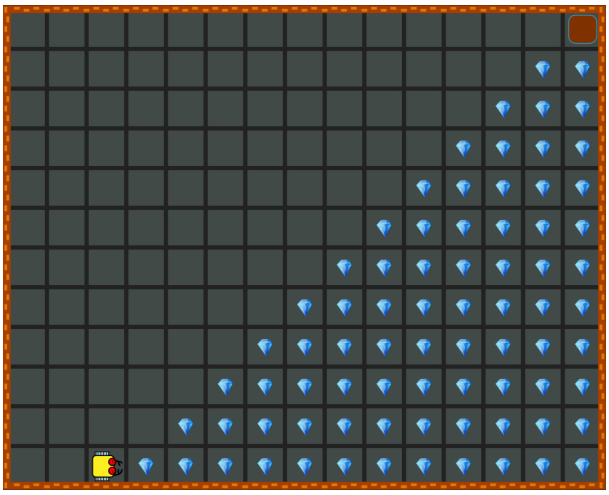


Fig. 92: Karel needs to build a heap of gems before he can get home.

10.4 Diamond tree

Karel stands under a diamond tree, with his home next to him on the the right. The tree is random – at any point it can have a branch in the north-west or in the north-east direction (or in both). Write a recursive algorithm for the robot to collect all gems from the tree and get home!

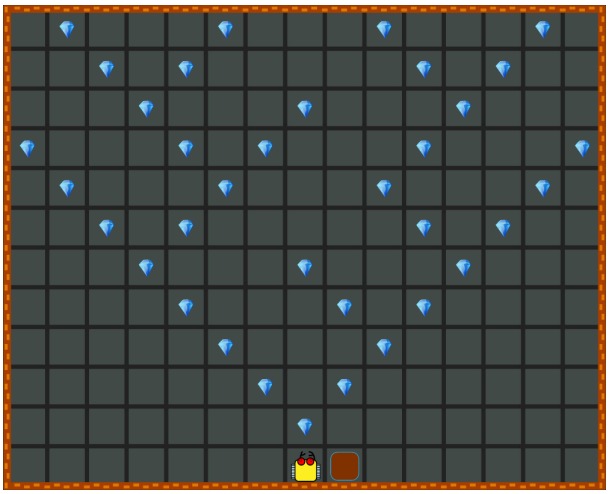


Fig. 93: Karel is traversing a random diamond tree.

11 Variables and Functions

11.1 Accounting

Karel has an unknown number of gems in his bag. Write a function `accounting` *for the robot to count the gems and return their number. Then call the function and print the result.*

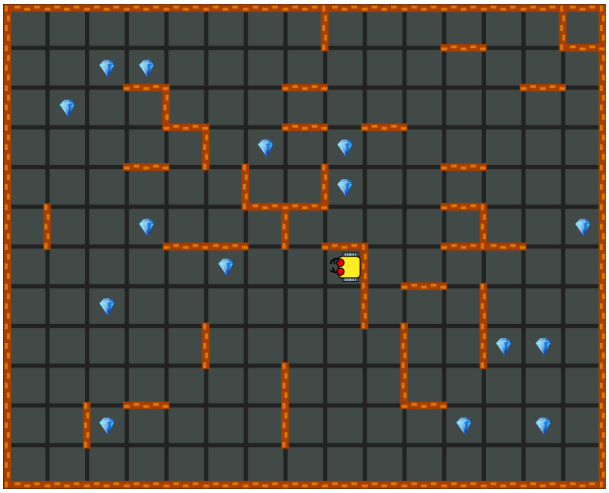


Fig. 94: Karel is counting his gems.

12 Lists

12.1 Tape measure

Karel stands in a rectangular room with random dimensions. Write function `measure` *that returns a list [x, y] where 'x' and 'y' are the horizontal and vertical sizes of the room, respectively. Call the function and print the result.*

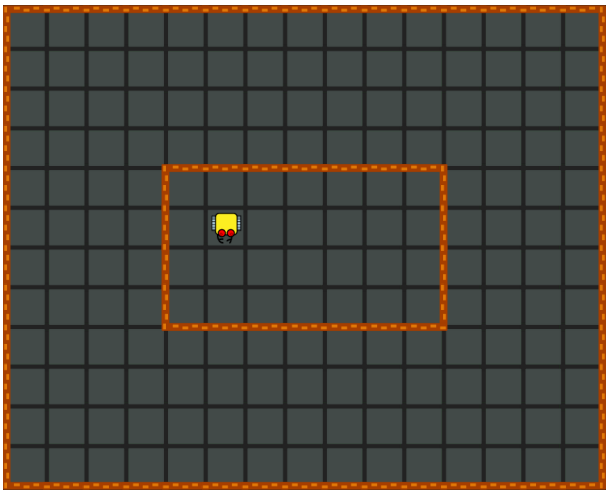


Fig. 95: Karel is measuring his room.

12.2 Reconnaissance

Karel stands in a rectangular room with random dimensions, that contains gems at random positions. There is at most one gem in each square. Write function `gems` *that returns a list of pairs [x, y] where 'x' and 'y' are the GPS coordinates of each gem. Call the function and print the result.*

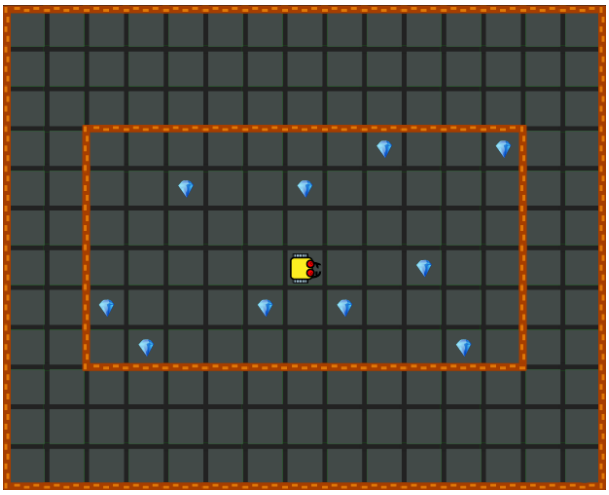


Fig. 96: Karel is locating gems.

12.3 Orchard

Trees in Karel's orchard are represented by piles of gems. The position of trees as well as the number of apples per tree are random. Initial position of the robot is in the south-west corner, facing east. There are no walls to crash into. Write a procedure `orchard` *for the robot to return a list of triplets [x, y, n] with one triplet per tree, where 'x' and 'y' are the GPS coordinates and 'n' the number of apples on the tree.*

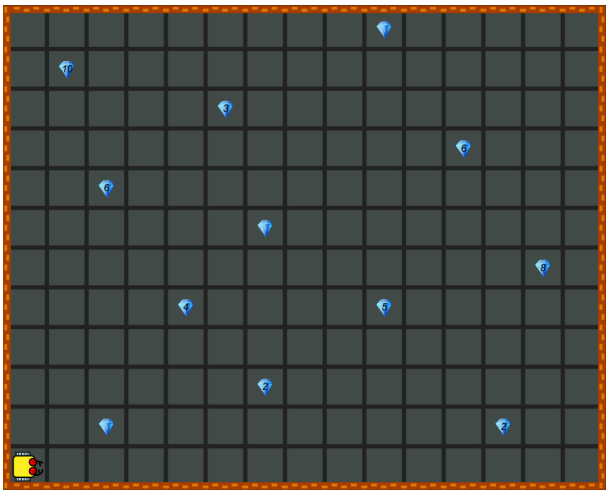


Fig. 97: Karel is locating trees and counting apples in his orchard.

12.4 New carpet

Karel needs new carpet! His apartment has random but simple shape – wherever the robot stands, when he goes north, south, east or west he will always reach exterior wall. The initial position of the robot is at the west wall of the bottom row, facing east. Write a function `carpet` *that returns a list of triplets [x, y, n] with one triplet for each row, starting with the bottom one, where 'x' and 'y' are the GPS coordinates of the left-most square, and 'n' is the size of the row.*

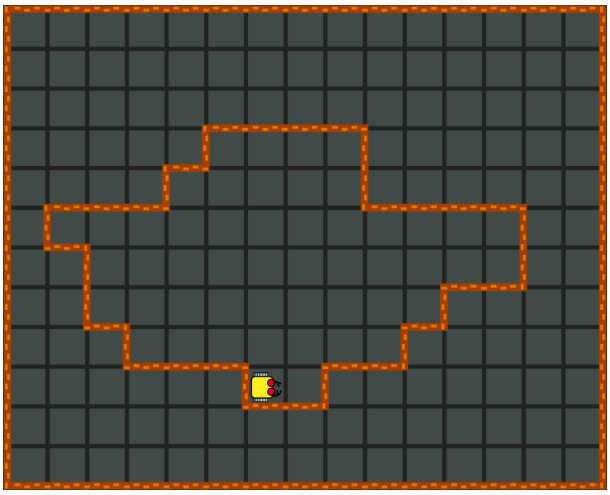


Fig. 98: Karel is measuring his apartment before buying new carpet.

13 Logic and Randomness

13.1 Reading numbers

Each 3x5 box contains a random integer number between 0 and 9. Write a function "readnumber" where Karel enters a box, determines the number in it, and returns it. Initial position of the robot is at the entrance of the box, facing north. The final position of the robot should be the same. Use this function to determine the numbers in the three boxes as shown in the screenshot.

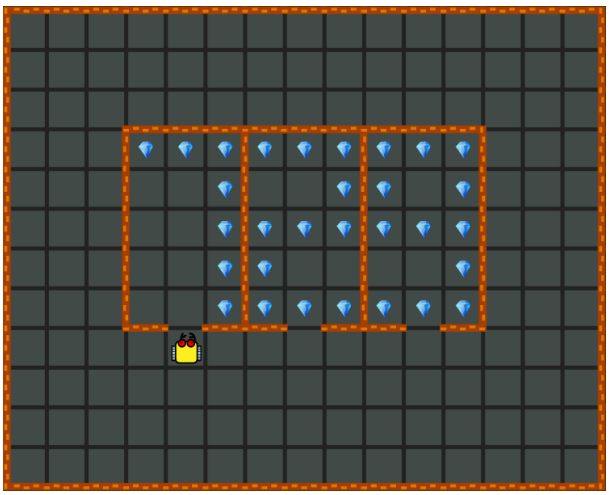


Fig. 99: Karel is learning to read numbers.

13.2 Writing numbers

There is a pile of gems at the entrance of each box. The number of gems in each pile is random between 0 and 9. Write a function "writenumber" where Karel counts the gems in the pile, enters the box, and renders the number. Then use the function to render numbers in three boxes as shown in the screenshot.

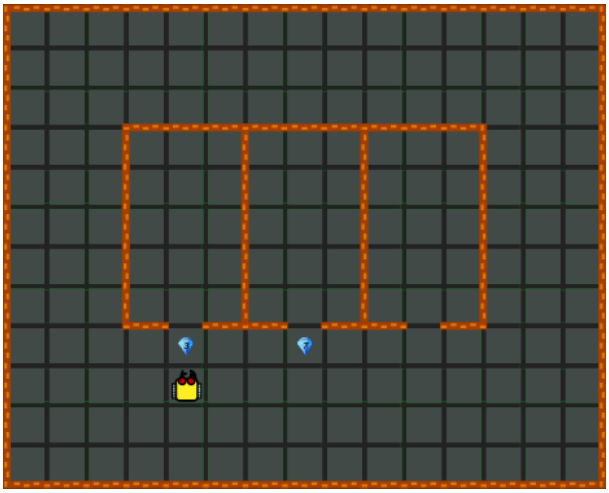


Fig. 100: Karel is learning to write numbers.

13.3 Adding numbers

This problem can be solved with the help of the results of the previous two exercises. The two upper boxes contain randomly generated numbers between 0 and 9. Karel stands at the entrance of the first box, facing north. Write a function `addnumbers` *where Karel will enter each box, determine the number in it, and then add the numbers together and return the result. Also render the result in the double-box below. Use the functionality developed in the previous two tasks!*

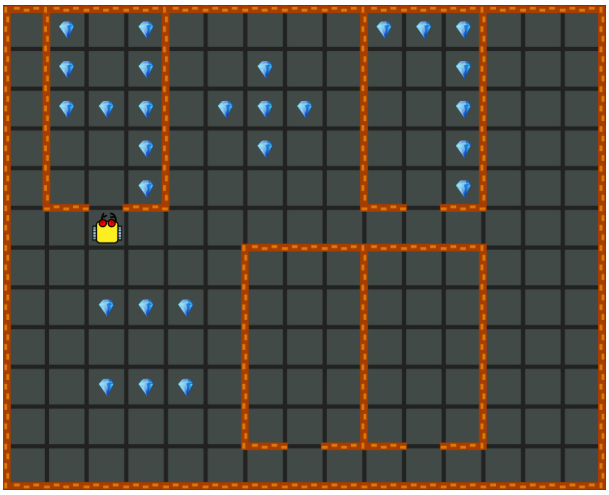


Fig. 101: Karel is adding randomly generated numbers.

14 Advanced Programs

If the previous exercises have not been challenging enough, then here is something for you. Enjoy!

14.1 Eight queens

Karel stands on an 8 x 8 chess board along with eight queens (gems). Recall that a chess queen dominates its row, its column, as well as both diagonals that pass through her position. Nothing may stand in these fields or she will destroy it. Currently, some queens are threatening each other. Write a program for Karel to correct the positions of the queens in such a way that none of them is threatened. You can assume that each column and each row contains exactly one queen.

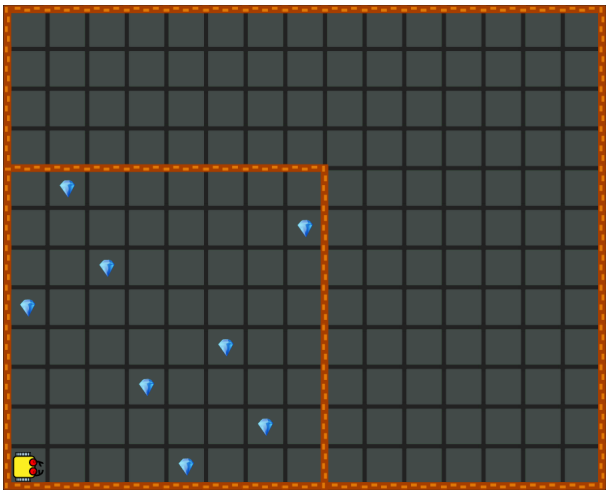


Fig. 102: The eight queens problem.

14.2 Bubble sort

Karel stands in front of 9 piles of gems. Each pile contains a random number of gems between 1 and 9. Two or more piles can be equally large. Sort the piles in such a way that the smallest pile is on the left and the largest on the right. Hint – look up the BubbleSort algorithm on Wikipedia.

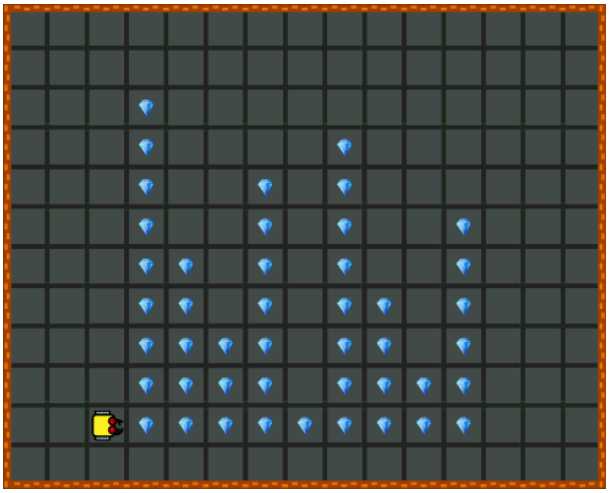


Fig. 103: BubbleSort algorithm.

14.3 Land surveyor

Karel took a job as land surveyor and his first task is to measure the size of a fenced property. However, there is a big dog inside and Karel is scared of dogs. If he fails to measure the size of the property, he will lose his job. Hence, you need to write a program for the robot to calculate the size of the property without actually entering it. You can assume that the fence does not touch the boundary of the maze and that Karel is free to walk along the entire length of the fence. The property has a random shape and the robot is standing next to the fence.

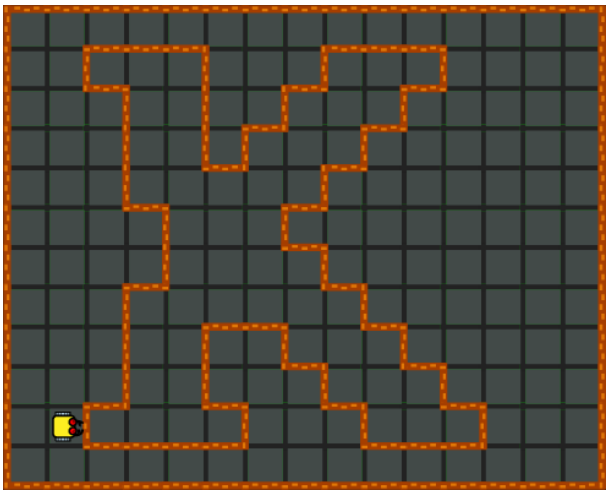


Fig. 104: Karel needs to measure the size of the property without entering it.

14.4 Loopy loop

Write a program for Karel to decide whether the walls in the maze form a closed loop or not. You can assume that the robot can freely walk along all walls and that they do not touch the maze's boundary.

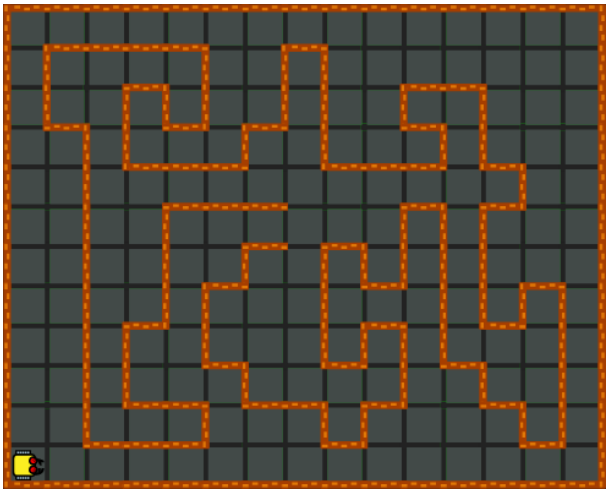


Fig. 105: Karel needs to decide whether the curve is open or closed.

14.5 Espionage

Karel works for the Secret Service. The word is out that enemy spies may be hiding in the maze. The rumor has it that they are hiding in a place that is completely surrounded with walls and thus not reachable from any direction. Hence, your task is to write a program for the robot to traverse the entire maze and decide whether all positions are reachable or not. Return a list containing all unreachable positions (the list should be empty if there are none). This list will be used by an assault team to eliminate all enemy forces.

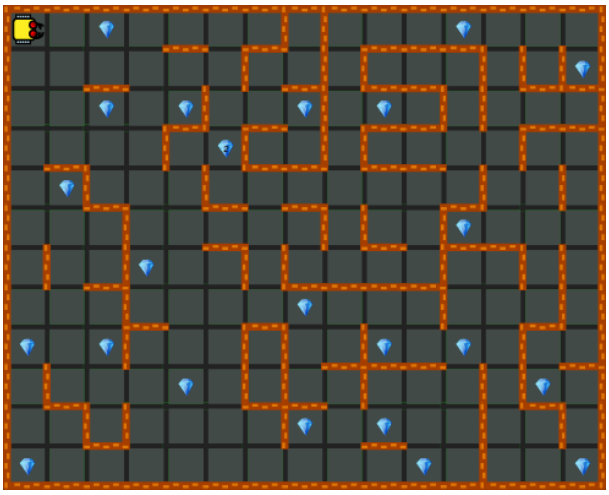


Fig. 106: Karel is looking for hidden spies.

Part III

Review Questions

1 Introduction

For all review questions in this book: None, one, or multiple choices may be correct.

1. Name the university where Karel the Robot was created.
   A1 MIT
   A2 Princeton
   A3 Harvard
   A4 Stanford
2. What is the programming language that you should learn after Karel?
   A1 C
   A2 Python
   A3 Haskell
   A4 Fortran
3. What is the largest conceptual difference between Karel and standard procedural programming languages such as Python, C, C++ or Fortran?
   A1 Karel knows only five commands.
   A2 Karel can only do simple algorithms.
   A3 Karel does not know math.
   A4 Karel has only five sensors.
4. Where does Karel live and what objects does he like to collect?
   A1 Karel lives in a maze and he likes to collect gems.
   A2 Karel lives in a cave and he likes to collect gold nuggets.
   A3 Karel lives on an island and he likes to collect coconuts.
   A4 Karel lives in a marketplace and he likes to collect coins from the ground.
5. What is the command that moves the robot one step forward?
   A1 step
   A2 forward
   A3 go
   A4 move
6. What is the command that turns the robot to the left?
   A1 left_turn
   A2 turn_left
   A3 turnleft
   A4 left
7. What is the command that turns the robot to the right?
   A1 right_turn
   A2 right
   A3 turnright
   A4 turn_right

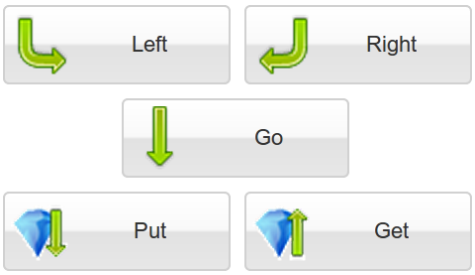
8. What is the command to pick up a gem from the ground?
   A1 pick
   A2 pick_gem
   A3 get
   A4 get_gem
9. What is the command to drop a gem on the ground?
   A1 put
   A2 drop_gem
   A3 drop
   A4 put_gem
10. What sensor helps the robot avoid crashing into walls?
   A1 wall_ahead
   A2 wall
   A3 crash
   A4 danger
11. What sensor helps the robot detect gems?
   A1 detect_gem
   A2 see_gem
   A3 gem
   A4 near_gem
12. What sensor does the robot use to check whether he has any gems in his bag?
   A1 bag_full
   A2 bag_empty
   A3 has_gems
   A4 empty
13. What sensor helps the robot detect his orientation?
   A1 south
   A2 north
   A3 east
   A4 west
14. What sensor helps the robot detect whether he is at home?
   A1 at_home
   A2 finished
   A3 home
   A4 stop
15. What is the most important skill in computer programming?
   A1 Writing short programs.
   A2 Designing great algorithms.
   A3 Writing programs quickly.
   A4 Knowing many programming languages.

2 Launching Karel

1. What are the four modes of the Karel application?
   A1 Manual mode, Programming, Designer, Games.
   A2 Mode 1, Mode 2, Mode 3, Mode 4.
   A3 Beginner mode, Intermediate mode, Advanced mode, Expert mode.
   A4 Karel has no modes.
2. What is the difference between Levels 1 and 2?
   A1 In Level 2 the robot has more sensors.
   A2 In Level 2 one can only guide the robot with the mouse.
   A3 In Level 2 the robot has new objects in the maze.
   A4 In Level 2 programs can contain conditions, loops, and custom commands.
3. In which level does the user start working with logical and numerical variables?
   A1 Level 1.
   A2 Level 2.
   A3 Level 3.
   A4 Level 4.
4. What is new in Level 3?
   A1 Parallel programming.
   A2 Object-oriented programming.
   A3 Scientific computing.
   A4 Variables and lists.
5. In which mode can the user build custom mazes?
   A1 Manual mode.
   A2 Programming.
   A3 Designer.
   A4 Games.

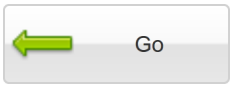
3 Manual Mode

1. How can Karel be switched to Manual mode?
   A1 Through Settings in main menu.
   A2 By pressing the Programming button twice
   A3 By pressing the Manual mode button.
   A4 Through the File menu.
2. Look at the arrows and decide which direction the robot is facing!



A1 North
A2 South
A3 East
A4 West

3. Look at the button and decide which direction the robot is facing!

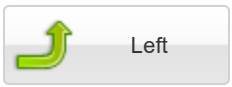


A1 North
A2 South
A3 East
A4 West

4. Which direction is the robot facing after this button is pressed?

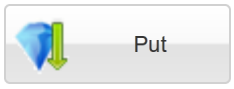
A1 North
A2 South
A3 East
A4 West

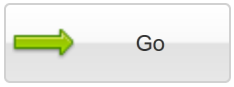
5. What is the function of the following button?

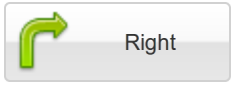


A1 Make one step forward, turn left, and make one step forward.
A2 Make one step forward and turn left
A3 Step aside towards East and then make one step forward.
A4 Turn left.

6. The robot is facing North. Which button will make him face East?

A1 

A2 

A3 

A4 

7. The robot is facing South. Which two buttons do you need to press to make him move one step ahead and turn East?

A1 

A2 

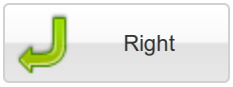
A3 

A4 

8. Select one or more correct statements from the four options below!

A1 All these buttons will turn the robot 90 degrees to the right.
A2 Two of the buttons will turn the robot to the left, the other two will turn him to the right.
A3 The last button on the right will turn the robot to the left.
A4 The last button on the right will turn the robot to the right.

9. After you press the following button, the robot will:



A1 Turn left and face West.
A2 Turn right and face West.
A3 Turn right and face East
A4 Turn right and face South.

10. The robot is facing South. Which two buttons will make him rotate 180 degrees?

A1 

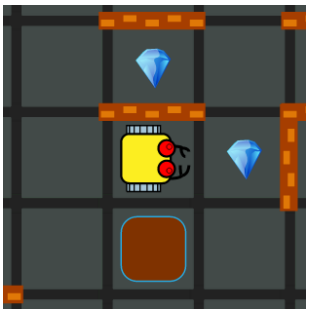
A2 

A3 

A4 

4 Programming Mode

1. Where do we enter programs for Karel?
   A1 In Microsoft Word.
   A2 In Wordpad.
   A3 We upload them from hard disk.
   A4 In code cells.
2. How are Karel's programs run?
   A1 By clicking on the green arrow in the menu.
   A2 By clicking on the green arrow under the code cell.
   A3 By clicking on Run in File menu.
   A4 By clicking on Run in Edit menu.
3. The robot's initial situation is as shown in the image. His bag with gems is empty. Read the following program and select one or more correct statements!



```
go
right
get
go
right
go
```

A1 The robot will return home with no gems in the bag.
A2 The robot will return home with one gem in the bag.
A3 The robot will not return home and he will have no gems in the bag.
A4 The robot will not return home and he will have one gem in the bag.

4. The robot's initial situation is the same as in the previous question. He has no gems in his bag. Read the following program and select one or more correct statements from the four options below!

```
go
get
left
go
left
go
get
right
right
go
right
go
put
go
left
go
```

A1 The robot will return home with two gems in the bag.
A2 The robot will return home with only one gem in the bag.
A3 The robot will not return home and he will have two gems in the bag.
A4 The robot will not return home and he will have only one gem in the bag.

5. What is an *algorithm*?
   A1 Very short program.
   A2 Sequence of instructions for the robot written using the correct commands.
   A3 Logical mistake in our program.
   A4 Sequence of logical steps written using human language.
6. What is a *program*?
   A1 Sequence of logical steps written using human language.
   A2 Algorithm that does not contain any mistakes.
   A3 Algorithm that is translated from human language to programming language.
   A4 Very long algorithm.
7. If the robot does something unexpected, what is the most probable reason for that?
   A1 Something is wrong with the computer.
   A2 Internet connection is too slow.
   A3 Web browser needs upgrading.
   A4 Our algorithm contains a logical mistake.
8. What of the following is a *logical* mistake?
   A1 Mistake in an algorithm that causes the robot to do something unexpected.
   A2 Opening Karel through the Programming menu instead of through File Manager.
   A3 Typing a command in a wrong way, such as `lft` instead of `left`.
   A4 Learning Fortran.
9. What of the following is a *syntactical* mistake?
   A1 Writing `left` instead of `right`.
   A2 Having two empty characters between commands.
   A3 Mis-spelling a command.
   A4 Mistake that causes the robot to do something unexpected.
10. What of the following will cause an error message?
    A1 Turning the robot four times to the left.
    A2 Putting two or more gems on each other.
    A3 Using the `put` command while the robot's bag is empty.
    A4 Returning home without any gems.
11. What do we mean by *debugging*?
    A1 Apologizing after we ask a friend too many questions.
    A2 Playing an algorithm in our head prior to writing a program.
    A3 Writing a new program after the previous one did not work.
    A4 Looking for mistakes in a program that does not work.

5 Counting Loop

1. What command should we use to make the robot repeat something a given number of times?
   A1 `repetition`
   A2 `loop`
   A3 `for`
   A4 `repeat`
2. Why should we always write one command per line?
   A1 To keep the code readable.
   A2 To make the code look longer.
   A3 Because all programming languages require that.
   A4 Because it speeds up communication with server.
3. What is the *body* of a `repeat` loop?
   A1 Body of a loop is the number that follows the `repeat` command.
   A2 The command on the first line following the `repeat` command.
   A3 One or more commands that follow the `repeat` command and are indented.
   A4 One or more commands that follow the `repeat` command and are not indented.
4. Why does the body of a loop need to be indented?
   A1 To make clear where it begins and where it ends.
   A2 It does not have to be indented, the indentation is optional.
   A3 Because the code is visually nicer.
   A4 Because the code is easier to read.
5. Which program(s) will rotate Karel 360 degrees?
   A1
   ```
   left
   left
   left
   left
   ```
   A2
   ```
   repeat 2
       right
       right
   ```
   A3
   ```
   repeat 4
       left
   ```
   A4
   ```
   repeat 2
       repeat 2
           right
   ```
6. Karel needs to pick up 5 gems from the ground, walk 10 steps forward, and put the 5 gems on the ground again. Which one(s) of the following programs will do that?

A1
```
repeat 5
    get
    repeat 10
        go
        repeat 5
            put
```
A2
```
repeat 5
    get
repeat 10
    go
repeat 5
    put
```
A3
```
repeat 5
    get
    repeat 10
        go
    repeat 5
        put
```
A4
```
repeat 5
get
repeat 10
go
repeat 5
put
```

6 Working with Code and Text Cells

1. How can a new text cell be added?
   A1 By clicking on *Add text cell* under an existing cell.
   A2 By clicking on *Add text cell* in the Edit menu.
   A3 By clicking on *New* in File menu.
   A4 By clicking on *Clone* in File menu.
2. How can one add a new code cell?
   A1 Click on *Add new code cell* in the Edit menu.
   A2 Click on *New* in File menu.
   A3 Click on *Clone* in File menu.
   A4 Click on *Add code cell* under an existing cell.
3. When should we use multiple code cells?
   A1 When our program contains more than one command.

A2 When we want to run parts of the program separately.
A3 When a command is repeated multiple times.
A4 When the program is longer than 10 lines.

4. What is the best way to erase all text from a cell?
   A1 Close Karel, logout, and login again.
   A2 Restart Karel.
   A3 Remove the code cell and add a new one in its place.
   A4 Click on *Clear cell* under the cell.
5. How can a cell be collapsed?
   A1 Click on *Remove cell* under the cell.
   A2 Click on *Collapse cell* under the cell.
   A3 Click on *Collapse* in File menu.
   A4 Click on the bracket on the right of the cell.
6. How can a cell be removed?
   A1 Click on *Add code cell* under the cell.
   A2 Click on *Remove cell* under the cell.
   A3 Click on *Clear cell* under the cell.
   A4 Switch to Designer and back.
7. How can we evaluate all code cells at once?
   A1 Through *Expand all cells* in Edit menu.
   A2 Click on the red button in the main menu.
   A3 Click on *Run cell* under the last code cell.
   A4 Click on the green arrow button in the main menu.
8. How should a running program be stopped?
   A1 Click on the green arrow button in the main menu.
   A2 Close the main Karel window.
   A3 Click on the red button in the main menu.
   A4 Click on *stop* under the last code cell.
9. How can we evaluate just one selected code cell?
   A1 Copy and paste the contents of all code cells into the first one, and remove them.
   A2 Click on the green arrow button in the main menu.
   A3 Click on the green arrow button under the code cell.
   A4 Click on *Remove cell* under the code cell.

7 Conditions

1. When does the `gem` sensor check true?
   A1 There is at least one gem in the maze.
   A2 There is at least one gem in front of the robot.

A3 There are one or more gems under the robot.
A4 The robot's bag contains at least one gem.

2. When does the `north` sensor check true?
   A1 The robot's home is North of him.
   A2 The robot is North of his home.
   A3 The robot faces South.
   A4 The robot faces North.
3. When does the `empty` sensor check true?
   A1 The robot's bag is empty.
   A2 There are no gems where the robot stands.
   A3 There are no gems in the maze.
   A4 There are no gems in the robot's home.
4. When does the `home` sensor check true?
   A1 Robot's home is right in front of the robot.
   A2 The robot is inside his home.
   A3 Robot's home is straight ahead of the robot.
   A4 The robot needs to go home to drop all gems that he collected.
5. When does the `wall` sensor check true?
   A1 The robot will reach a wall with one or more steps.
   A2 There is a wall on the robot's right-hand side.
   A3 There is a wall right in front of the robot.
   A4 There is a wall on the robot's left-hand side.
6. When can Karel see from where he stands whether a wall is two steps ahead?
   A1 Never.
   A2 If his home is not in the way.
   A3 If no gem is in the way.
   A4 If no other wall is in the way.
7. When can the robot check without turning whether a wall is on his right?
   A1 Any time, there is the sensor `right` for that.
   A2 Any time, there is the sensor `wall` for that.
   A3 Only if he is not at home.
   A4 Never.
8. Can Karel check from where he stands whether a gem is one step away?
   A1 Yes, there is the sensor `gem` for that.
   A2 No.
   A3 Yes but the gem must be in front of him.
   A4 Yes but not when he is at home.
9. Can he check from where he stands whether his home is one step away?
   A1 Yes but his home must be in front of him.
   A2 Yes, there is the sensor `home` for that.

A3 No.
A4 Yes but not when he is at home.

10. Can the robot check whether he has at least one gem in the bag?
A1 No.
A2 Yes, he can use the `gem` sensor.
A3 No, the sensor gem only works for one gem.
A4 Yes, he can use the `empty` sensor.

11. Which program(s) make Karel check whether he stands on a gem, and if so, to pick it up?
A1
```
if gem
    get
```
A2
```
if not empty
    get
```
A3
```
if get
    gem
```
A4
```
if empty
    get
```

12. Which program(s) will turn Karel to the North, regardless the direction he is facing?
A1
```
repeat 4
    if not north
        left
```
A2
```
if not north
    left
else
    right
```
A3
```
if south
    repeat 2
        right
```
A4
```
if not north
    right
```

8 Conditional Loop

1. What is the difference between the `repeat` and `while` loops?
A1 Body of the `while` loop does not have to be indented.
A2 The `while` loop only can be used when the number of repetitions is known a priori.
A3 The `repeat` loop can do 100 repetitions maximum.

A4 The `repeat` loop only can be used when the number of repetitions is known a priori.

2. Which program will always turn Karel to face West?

A1
```
repeat 4
    left
```
A2
```
while not west
    right
```
A3
```
while not north
    left
left
```
A4
```
while not north
    right
left
```

3. The maze does not contain any gems and any walls except for the ones that form the outer rectangular boundary. The robot stands in the south-west corner facing East. Which program will make the robot walk along the boundary of the maze and bring him back to the original position?

A1
```
repeat 4
    while not wall
        go
```
A2
```
repeat 4
    while not wall
        go
    left
```
A3
```
repeat 4
    while not wall
        go
    right
```
A4
```
repeat 4
    while not wall
        left
    go
```

9 Custom Commands

1. Should we, or should we not always try to split a big task into smaller ones?

A1 No, because it makes the resulting program longer.
A2 Yes, because then we can only solve the simple tasks.
A3 Yes, because then we can ask help solving some tasks.

A4 Yes, because the big task becomes simpler to solve.

2. Should we, or should we not replicate computer code?
   A1 Yes, because the resulting program is faster.
   A2 Yes, because the resulting program is simpler.
   A3 No, because our code would become prone to errors.
   A4 No, because the Karel language does not allow that.
3. When should a new command be defined?
   A1 Whenever the same command is repeated on two consecutive lines in the code.
   A2 Whenever the algorithm contains an action that is repeated in more than one situation.
   A3 Whenever the code contains an `if - else` statement.
   A4 If the code is longer than 100 lines.
4. Which command(s) will empty the robot's bag without causing an error?
   A1
```
def emptybag
    while gem
        put
```
   A2
```
def emptybag
    while not gem
        put
```
   A3
```
def emptybag
    while empty
        put
```
   A4
```
def emptybag
    while not empty
        put
```
5. Which command(s) will turn Karel to the South no matter which direction he is facing?
   A1
```
def turnsouth
    while north
        repeat 2
            left
```
   A2
```
def turnsouth
    while not north
        right
    repeat 2
        left
```
   A3
```
def turnsouth
    while not north
        right
    repeat 2
        right
```

A4
```
def turnsouth
    while not north
        left
    repeat 2
        left
```

10 Recursion

1. There are three commands `A`, `B`, `C`. Identify all cases of recursion in the four options below!
   A1 `C` calls `B`, `B` calls `A`, `C` calls `A`.
   A2 `C` calls `B`, `B` calls `C`, `C` calls `A`.
   A3 `C` calls `B`, `B` calls `A`, `A` calls `C`.
   A4 `C` calls `B`, `B` calls `A`, `A` calls `B`.
2. What do we mean by *base case* in recursion?
   A1 The initial state of the robot before executing a recursive program.
   A2 Recursion where a single command calls itself directly.
   A3 Branch of a conditional statement that makes a recursive call.
   A4 Branch of a conditional statement that does not make a recursive call.
3. What happens if base case is not present?
   A1 Recursion turns into an infinite loop.
   A2 The program throws an error.
   A3 The program will end after the first call to the recursive command.
   A4 The program does not throw an error but the robot will do nothing.
4. When should recursion be used?
   A1 The algorithm contains multiple conditional statements.
   A2 The algorithm contains multiple repetitions.
   A3 After solving part of the problem, the remaining task is similar to the original problem.
   A4 Our program contains multiple new commands.
5. Which of the four recursive commands below turn(s) the robot to the North without causing an infinite loop?
   A1
```
def turn_north
    right
    turn_north
```
   A2
```
def turn_north
    if not north
        right
    turn_north
```

A3
```
def turn_north
    if not north
        right
        turn_north
```

A4
```
def turn_north
    right
    if not north
        turn_north
```

11 Variables and Functions

1. What are *variables* used for in programming?
   A1 To generate random data.
   A2 To make programs shorter.
   A3 To store useful information.
   A4 To create variable mazes.
2. What values do the read-only variables `gpsx` and `gpsy` have when Karel's position is as shown in Fig. 107?

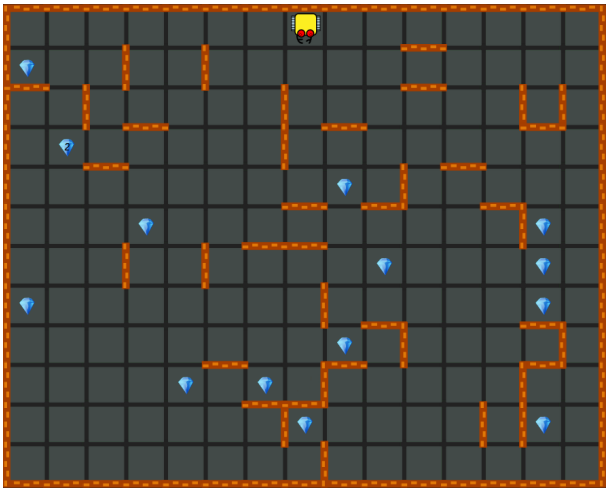


Fig. 107: Karel is reading his GPS device again.

A1 `gpsx` is 7 and `gpsy` is 12.
A2 `gpsx` is 8 and `gpsy` is 0.
A3 `gpsx` is 8 and `gpsy` is 12.

A4 `gpsx` is 7 and `gpsy` is 11.

3. What values will the variables `gpsx` and `gpsy` have after Karel executes the following program? He starts from the position shown in Fig. 107.

```
repeat 5
    while not wall
        go
        if gem
            get
    left
```

A1 `gpsx` is 7 and `gpsy` is 7.
A2 `gpsx` is 0 and `gpsy` is 10.
A3 `gpsx` is 9 and `gpsy` is 7.
A4 `gpsx` is 9 and `gpsy` is 11.

4. Which of the following programs are invalid?

A1
```
a = gpsx
inc(a)
```
A2
```
b = 7
dec(b)
```
A3
```
gpsx = 0
inc(gpsx)
```
A4
```
e = 1
e = gpsx
e = gpsy
```

5. Which of the following programs are invalid?

A1
```
def count_steps
    n = 0
    while not wall
        go
        inc(n)
    return n
```
A2
```
def count_steps
    n = 0
    while not wall
        go
        inc(n, 1)
    print "Number of steps made =", n
```

A3
```
def count_steps
    n = 0
    while not wall
        go
        inc(n)
```
A4
```
def count_steps
    while not wall
        go
        inc(n)
    return n
```

12 Lists

1. Which of the following are correct ways to define a list?
   A1 `L = []`
   A2 `L = [1, 2, 35]`
   A3 `L = [20, "Monday", [5, 15]]`
   A4
   ```
   word = "Monday"
   list = [5, 15]
   L = [20, word, list]
   ```
2. What is the correct way to store the length of list `L` in variable `m`?
   A1 `m = L`
   A2 `m = length(L)`
   A3 `m = len(L)`
   A4 `m = L[0]`
3. List `Y` is defined as `[1, 4, 9]`. Which of the following codes will throw an error?
   A1 `print Y[0]`
   A2 `a = L[1]`
   A3 `b = L[2]`
   A4 `print L[3]`
4. Which of the following are correct ways to append `100` to a list `X`?
   A1 `append(X, 100)`
   A2 `X.append(100)`
   A3 `append(100, X))`
   A4 `100.append(X)`
5. List `Z` is defined as `["January", "February", "March"]`. Which of the following codes will remove the last item from `Z` and assign it to the variable `month`?
   A1 `month = Z.pop()`
   A2 `month = Z[2]`
   A3 `month = del Z[2]`
   A4 `month = Z.pop(2)`

13 Logic and Randomness

1. Let's introduce a variable `my_age` that stores your age in years. This variable is:
   A1 Numerical.
   A2 Logical.
   A3 Both numerical and logical.
   A4 Neither numerical nor logical.
2. What is a *logical expression*?
   A1 Expression that makes sense.
   A2 Expression that is always true.
   A3 Expression that is either true or false.
   A4 Expression that cannot be false.
3. A is `True` and B is `False`. What is then the value of (A *and* B) ?
   A1 `False`.
   A2 `True`.
   A3 It can be both `True` and `False`.
   A4 Neither `True` nor `False`.
4. A is `True` and B is `False`. What is then the value of (A *or* B) ?
   A1 `False`.
   A2 `True`.
   A3 It can be both `True` and `False`.
   A4 Neither `True` nor `False`.
5. A is `True` and B is `False`. What is then the value of (A *and not* (A *or* B)) ?
   A1 `False`.
   A2 `True`.
   A3 It can be both `True` and `False`.
   A4 Neither `True` nor `False`.
6. A is `True` and B is `True`. What is then the value of (A *and not* (A *and not* B)) ?
   A1 `False`.
   A2 `True`.
   A3 It can be both `True` and `False`.
   A4 Neither `True` nor `False`.