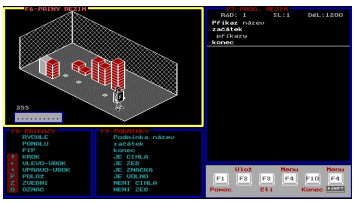
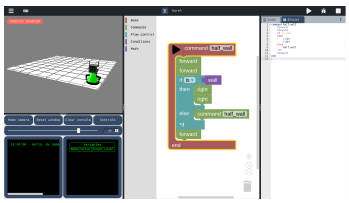
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

Karel 3D – Application for Teaching of Programming

Vojtěch Čoupek





Abstract
The paper discusses the problem of teaching the basics of programming to the upper primary school and secondary school students. Firstly, it introduces Karel programming language, which is a tool that has been used since 1981, and some of its most important versions. Afterwards, the current trends employed in teaching the basic understanding of programming languages like block programming will be mentioned. Then, it presents a number environments used for this purposes and discusses their strong and weak points. The main goal is to create a new modern environment based on Karel programming language with up to date features that allow to teach programming in a playful and entertaining way.

Keywords: Teaching of programming – pedagogical programming language – block programming – web application – syntax checking – 3D graphics – JavaScript – Karel language – primary school – secondary school – beginners in programming

Supplementary Material: Aplication website — GitHub repository — Demonstration video

xcoupe01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In the world where almost everything is driven by computers, we need people who can understand them, manage them and create programs that make these computers useful. These people begin as students; therefore, they need to start with something simple. We cannot expect anyone who has not got any experience in the field of programming to understand modern powerful programming languages like C++ or JavaScript. The complexity unnecessary for the beginners, the inability to visualize their actions and the language barrier makes them really unsuitable for teaching the basics of programming.

That is why there are several specialized languages that can check these boxes. One of them is Karel programming language, which was designed by professor Richard E. Pattis and published in a book in 1981 (see [1]). This language allows one to control a robot in a closed environment. This environment is a two dimensional grid consisting of right to left streets and top to down avenues. The robot can move in this environment, ask simple questions and place beepers if it has any in its backpack. Robot's actions are programmed via the textual representation of Karel language. There are several projects based on this idea, such as a Czech translation for the robot in 2D room by Oldřich Jedlička [2].

One of the most important versions of environments that use the same idea as professor Pattis described was the project of Andrej Blaho and his team. They developed Karel in 3D for the operating system MS-DOS (visualized in the teaser images – first from

left). This project exerted a great influence on the author because this Karel was used to teach him the basics of programming. More about it can be learned from the Czech site created by Karel Klíma [3]. But due to its implementation for the MS-DOS operation system, it has its limitations on modern machines and often leads to poor workability. It is not user friendly, for example has no mouse support, which is really important for those beginners. Yet, it is still used in some schools in the Czech Republic because it offers great possibilities to demonstrate and teach basic code structures and programming logic. Karel will be thoroughly analyzed in Section 2.

Apart from Karel there are, of course, other solutions. As you already know, Karel is a relatively old idea and many new great projects have appeared since the 1980s. One of the most popular environments is Scratch[4]. Scratch is a free to use multi-platform tool developed by the MIT media lab which teaches programming through creating one's own animated 2D characters, or even through creating their own simple 2D game. Unlike Karel, Scratch does not use the text representation. Instead of the text, Scratch uses a modern block-based representation, which is really convenient for the users, because it enables the syntax control by the blocks in the process of program creation. Also the syntax is visualized by the blocks connection so the user understands which blocks can be connected, as well as the meaning of the connection. The blocks are also color-coded so that similar functions represented by the blocks have a similar color. This approach to defining the code is really popular nowadays in languages for beginners and a lot of environments are based on it.

The goal of this project is to create a new modern version of the environment for Karel language and make it more accessible for users. Apart from preserving the original function and options, it enhances its capabilities and supplements it with modern elements that are widely used and popular. Therefore, I designed and developed a web-based application, which can be run and used on many different devices including PCs, laptops and tablets. In its current condition, it is a replica of Andrej Blaho's team version for MS-DOS but extended by the block programming, truly 3D graphics, variables support and a professional text code editor. My application mentioned above contains both approaches to defining the code and the users can choose which one they prefer. A multilingual support is in progress. Currently, there is the Czech and English version in both block-based and source-code-based representation.

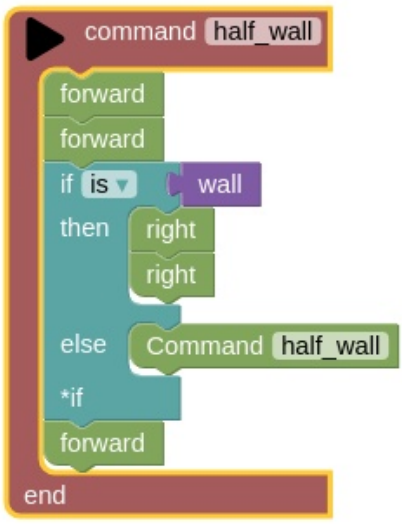


Figure 1. Example of a code defined with blocks

2. About robot Karel

This section will discuss the possibilities of Karel language and its limitations. Later in Section 3.4, an example of Karel programming language will be demonstrated in the new environment. The following structures are applicable in the application proposed by this paper. As mentioned before, Karel is a robot that can interact with a closed environment. It is important to note that the robot's face determines the block in front of it, which is essential for the robot's interaction.

2.1 Commands

Commands make it possible for the robot to interact with its surroundings. Robot is placed in a square shaped room created from blocks similar to a chess board. Karel can be only on one of these blocks in the room and the robot cannot escape the room. If the block in front of it is missing, it is defined that there is a wall.

Firstly, it can turn using commands `right` and `left`. Karel can move forward with the command `forward` if there is a space in front of it. The robot is able to move only to the blocks inside the room, climb one brick at a time and it does not mind falls. Karel can place bricks in front of it, but the bricks must be laid on a ground block or another brick. Karel cannot place and pick a brick more than one brick below its position and more than ten bricks above its position. As mentioned before, the robot can step on the bricks and use them to move vertically in the room.

In addition to bricks, the robot can place a mark on the block where it stands. The mark is always on the top of the block, even if Karel is adding bricks to the same block. Karel can mark its current location by the command `mark` and remove the mark

by `unmark`.

Lastly, the user can alter the speed in which Karel executes the commands by using keywords `faster` and `slower`. The robot can also make a beep sound by using the command `beep`.

2.2 Conditions

Apart from interacting with the room, Karel can also sense what is around. In the code, we use conditions which can, at any given time and position, answer simple yes or no questions. There are four basic conditions:

wall – says true if there is a wall in front of Karel.
brick – says true if there are one or more bricks on the block in front of Karel.
mark – says true if there is a mark on the same block as is Karel's current location.
vacant – says true if Karel can make a step forward on the block in front of it.

These conditions can be used in branching and looping control structures, described in Subsection 2.3.

2.3 Cycles and Decision Making Structures

Block-structured programming is based on decision making structures and jumps, so Karel also provides some basic ones. To close a code in these structures, most programming languages use brackets, but Karel language uses an asterisk and a type of the cycle to close as shown in the following examples. In the block of a code that is nested by this cycle, users are free to use other cycles and commands.

The simplest one is `do` cycle, which repeats the nested code a given number of times. The structure looks like this:

```
do [number] times
    [block of code]
*do
```

A slightly more advanced structure is `while` cycle, which repeats its nested code until the given condition is true.

```
while is/not [condition]
    [block of code]
*while
```

The last one is the decision making `if` structure. It takes the condition and executes its nested commands if the condition is true. There is an optional `else` branch, which will be executed if the condition is false. So this structure can be one of following:

```
if is/not [condition]
then
    [block of code]
*if

# or with the else branch

if is/not [condition]
then
    [block of code]
else
    [block of code]
*if
```

In the last example, there is a line starting with `#`. Users can employ this to create comments of their code and the interpreter will ignore these lines. They will turn green to indicate that this part of a code is the command.

2.4 User-Defined Commands as Conditions

Karel language allows users to create their own commands and conditions. In fact, it is required to make some part of a code executable. The difference between a command and condition is that the condition can output true or false value while the command cannot output anything. The commands can be created by the following structure:

```
command [identifier]
    [block of code]
end
```

If the user wants to create a custom condition, he or she will use this structure:

```
condition [identifier]
    [block of code]
end
```

The identifier must be unique among all other user defined commands and conditions and cannot be one of the language key words. After the command or condition is defined, it is possible to use them as commands or conditions in other parts of the code. Karel also supports recursions, so you can find the center of the wall by the following example:

```
command find_center_wall
    forward forward
    if is wall
    then
        left left
    else
        find_center_wall
    *if
    forward
end
```

If the condition is created, it requires special commands `true` and `false` to define the outcome of the condition but they do not end the execution of the

condition execution block. It is recommended that the robot ends in the same location and the room is the same as it was at the beginning of the execution of the condition. Those conditions can then be used in the `while` and `if` structures.

2.5 Expressions and Variables

Expressions and variables are not accessible in the original language; they are implemented in the revised environment. The integrated expressions support operations with positive integer numbers listed below:

+ – addition
− – subtraction
***** – multiplication
/ – integer division
% – modulo
() – brackets

There are also comparison operators which return one if they are true or zero if not. These operators contain:

> – greater than
>= – greater than or equal
< – less than
<= – less than or equal
== – equal to
!= – not equal to

Expressions can be used to define the number of repetitions in `do` cycle or to replace conditions in structures `if` and `while`. When replacing the condition, the expression is considered as true if the outcome is not zero and false if the outcome is zero.

Also the variable support is added so that the user can use variables instead of numbers in the expressions. It is possible to create variables with the global or local scope. As expected, the global variables are accessible anywhere from the code and can be used to pass values between commands and conditions, whereas local variables can be used only in the command or condition where they are defined.

To define the global variable, a user must employ the following syntax outside any command or condition declaration:

```
global [variable name] = [expression]
```

Defining a variable with the same name as any command or condition will cause error. When defined, the variable value can be set in any command or condition with the same syntax or used in other expressions.

Similarly, the local variables can be defined by the following syntax in any command or condition:

```
local [variable name] = [expression]
```

Defining the local variable with the same name as any command, condition or global variable is not allowed. There can be an optional keyword `variable` after the keywords `global` or `local`.

3. The New Version

This section will give the reader a slightly deeper insight into the technical aspects of the new version, name some valuable libraries used by the application, talk briefly about the user interface and, in the end, show some simple example exercise in the new Karel.

The project was designed for the web environment, so any user on any device could run Karel. The main programming language employed in this project is JavaScript for the functional parts combined with markup language HTML and style sheet language CSS that define the user interface.

3.1 Syntax Checking and Interpretation

The main thing that the application needs to do is to check and interpret the user-defined code. Instead of checking purely the given code, it is firstly split into tokens, which are stored and make up the internal representation of the defined code. After that step, it is ready to be checked.

For that purpose, the textual representation uses predictive parsing based on LL1 method, which checks the code by trying to generate the user code with the given rules that Karel language must follow. More details about this method can be found in Czech paper by Miroslav Novotný [5]. For that, the internal token representation is used, and the tokens are further processed and saved into internal lists. If any misunderstandings or errors are found, they are stored, and, at the end of syntax checking, shown as a tooltip in the editor.

If no error is detected, the application can proceed to the interpretation. The interpreter takes the newly created internal representation and, using those tokens, iterates the array through the defined code. The cursor in the text editor is moving with the interpreter so the user can tell what part of the code is currently being processed.

3.2 Used Libraries

To make this all happen, some libraries were used to tackle certain problems regarding the visualization of the room with the robot and the ways in which the user can enter the code. Here are the tools which are employed by the application:

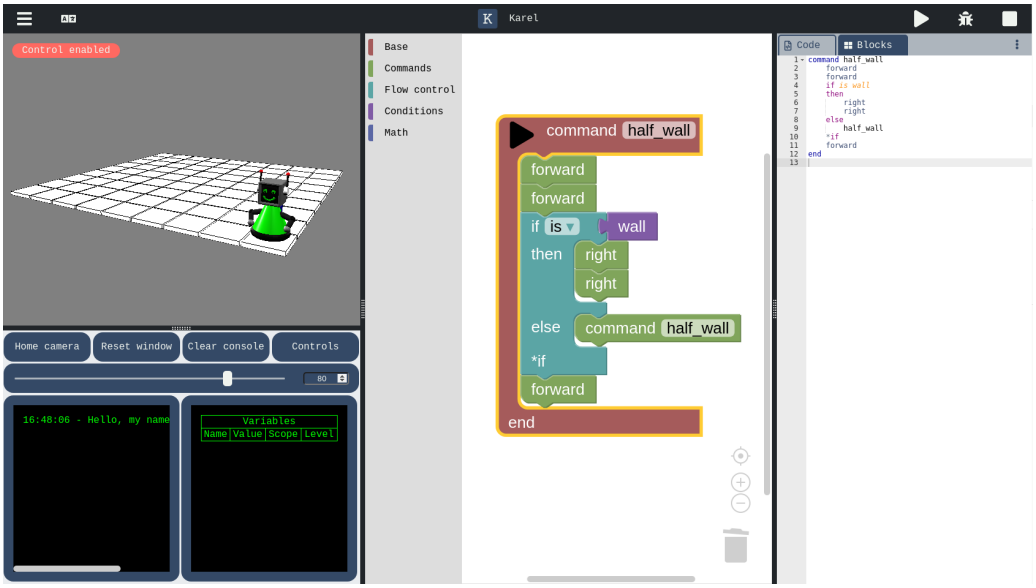


Figure 2. User interface of the new version of Karel

Three.js – The main graphical library of the project. It takes care of visualizing the room, loading objects into the room, lighting it, ect.

ACE – As the user writes the code, this text code editor handles the code highlighting and other needed functions. It is one of the most popular code editor libraries used in many known apps (for example Overleaf).

Blockly – The block code editor allows the user to define a code by blocks. This library is frequently used by the beginner programming languages like Scratch.

Split.js – It can make the user interface modular by allowing resizable multiple windows (or in HTML `divs`).

jQuery – It is a very popular JavaScript library with its extension jQuery UI employed in this program to realize popup dialogs.

3.3 User Interface

This section will briefly discuss the current user interface visualized in Figure 2. From the left to right there are the following parts:

Room visualization – users can see the room and the robot here, the camera of the visualization can be controlled via mouse. If the users activate this room by clicking on it (the room focus indicator goes red), then they can control the robot directly with the keyboard using the following keys:

- W – step forward.
- A – turn left.
- D – turn right.
- P – place one brick in front of the robot.
- Z – pick up one brick in front of the robot.
- O – mark the unmarked position below the robot or unmark the marked position below the robot.
- I – remove a block in front of the robot from the room or bring back the removed block.

Blockly editor – users can create their own programs using the block representation. In the tool bar on the right, there are categories in which they can find blocks they might need. If they nest some commands in one of the blocks from `Base` category, the run button is available, which can start the execution of the nested code.

Text code editor – users can create programs using the text code representation. If toggled, it can also visualize the current code defined by the blockly editor in the read only mode. The code is color highlighted and if any error is found during the checking, it is displayed here by underlining the word and error message in the left gutter.

All these parts of the application are resizable, so the users can grab and drag the splitters to hide some unwanted editors or to adjust the environment as they wish.

On the top, there is the main application bar, which contains, from left to right:

Main menu – from this menu, the users can access functions like changing the room size, saving and loading application states, ect.

Languages – selects the language of the application.

Run – executes the code selected in the text editor.

Debug – runs the selected code in the debug mode.

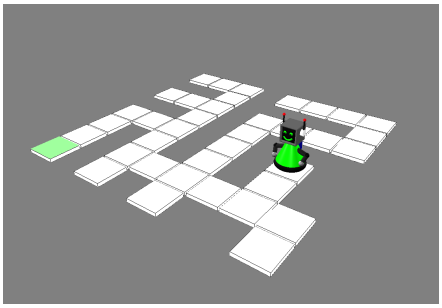


Figure 3. Karel in the maze, the task is to get to the green marked block.

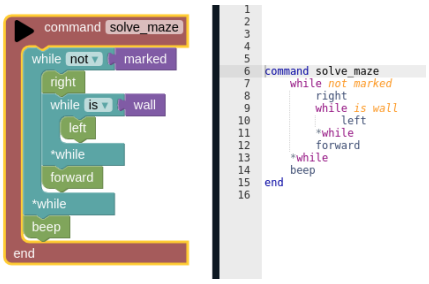


Figure 4. The maze solving code written in a text and block representation

Stop – stops any currently executing function.

Finally, below the room visualization, there is a quick action menu, which also shows the current state of the application, such as variable values and output terminal. The user can control the environment behavior, e.g. interpretation speed, from here.

3.4 What Karel Can Do

In this section, a simple example exercises will be shown. The first task is to get the robot from its current position to a marked position in the maze visualized in Figure 3.

To solve this problem, it is necessary to create our own command, which can be named `solve_maze`. We need to stop this command if there is a mark on Karel´s current position, which can be accomplished by using `while` cycle with the condition `not mark` to run its nested command when the robot in not standing on the mark.

One of the algorithms that can solve the maze is that the person in the maze places his/her left or right hand on the wall and goes forward by always touching the wall with his/her hand. Karel can do the same. In the following example, the right hand will be chosen. This can be done by turning the robot right, then turning left, until there is a wall in front of it, and then going forward. When the robot arrives at the desired marked position, it will beep to let the user know that the robot has finished its task. The code can look like the one seen in Figure 4 and it is demonstrated in the demonstration video. The reader can see the

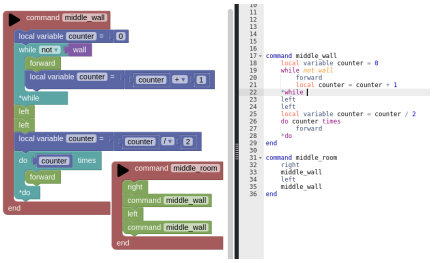


Figure 5. The text and block representation of the code, which can find the middle of the room

example solution in the application here.

The second example utilizes the variable extension of Karel language. The task is to move the robot from the initial position in the corner closest to the camera to the center of the room. To make this exercise possible, it is necessary to have a room with the odd number of blocks in both dimensions. The given solution will also work for rooms with the even number of blocks, but the robot will end on one of the four blocks around the center of the room. In the example, there is a room with nine by nine dimensions. The solution using recursion was already presented in Figure 1, but now another approach with variables will be shown.

A simple algorithm to solve this problem is to move across the room and count every step till there is no wall in front of Karel. Then the robot turns around, divides the counted steps by two and makes the calculated number of steps. Afterwards, the program turns the robot to the left and repeats the first part of the algorithm. Now the robot is in the middle of the room. The reader can see the example solution in Figure 5 or try it in the application here.

If the user clicks on the middle button in the top main bar, he or she will be taken to the main page of the information system built around the application. In the current implementation, only the Czech version of the system is available. This page includes some information about the project and controls, but mainly there are other exercises to solve and check following the proposed solutions. If needed, they can be loaded in Czech and then translated into English by switching the language in the application. The exercises are arranged from the easiest to the harder ones, but the list is not complete and more exercises are to be added.

There is one more function provided by the information system – the users can create their own account. When logged in, users can save the state of the application to the cloud or load any saved state from the cloud. These options are added to the main menu in the application. Login is accessible only from the information system main page on the top right corner.

4. Conclusions

To sum up the project, I think that the new Karel can be a useful tool for teaching the beginner students the basics of programming. It is based on a solid foundation laid by many successful students and teachers. The new version employs many modern technologies that the students can encounter later in their instruction and which make the study easier and more enjoyable, such as block programming or syntax highlighting.

The further development will involve the creation of more exercises for users and polishing and improving the information system. I hope that it will facilitate the education of many successful and talented programmers.

Acknowledgements

I would like to thank my supervisor, Ing. Zbyněk Křivka, Ph.D., for his help and advice during the development of the project. Also I would like to thank my secondary school teacher, Mgr. Roman Ondrůšek, and my father, Mgr. Petr Čoupek, who taught me how to program, introduced me to Karel language and made the first ever prototype with me.

References

[1] Richard E. Pattis. *Karel The Robot: A Gentle Introduction to the Art of Programming*. Wiley, 2 edition, 1995.

[2] Oldřich Jedlička. Robot Karel: vývojové prostředí. online, 2006. http://karel.oldium.net/.

[3] Karel Klíma. Karel. online, 2007. http://karel.webz.cz/.

[4] Mitchel Resnick. Scratch. online, 2005. https://scratch.mit.edu.

[5] Ing. Miroslav Novotný. Syntaktická analýza založená na stavových gramatikách. online, 2015. https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=115409.