

Karel the Robot

Extending the Primitive Commands

We know how to write programs using Karel's primitive commands

```

move
turnleft
pickbeeper
putbeeper
turnoff

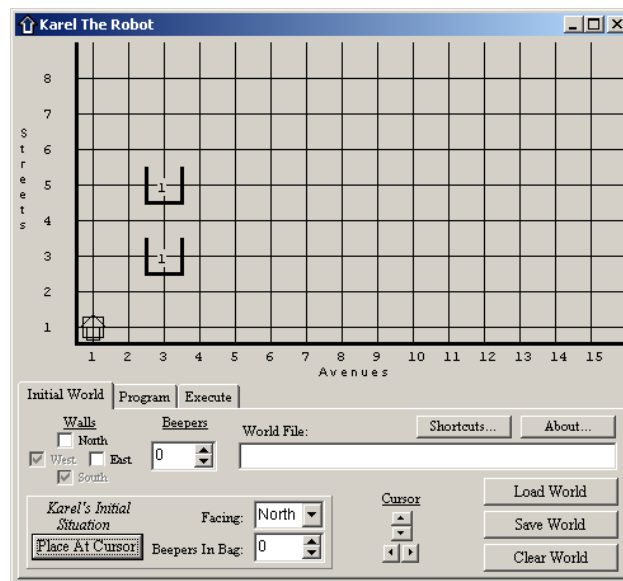
```

We know how to navigate between Karel's World view, Karel's Program view and Karel's Execution (or Run) view.

We know how to Compile a program, which means to translate it from Karel's programming language, which humans can understand as well as Karel into machine code that the computer inside Karel understands.

AND, we immediately saw that writing programs with only the basic 5 primitives is very tedious. It is hard to keep track of exactly what we are asking Karel to do.

We have already seen how Karel can go out into the world and retrieve beepers while going around obstacles in its path. In this example Karel is to go out into the world and retrieve the beepers found in two boxes, bring the beepers home and deposit them at the origin. The World view looked something like this:



As we saw in our first example, it can be painful to keep track of all the move instructions and all the turnleft commands. This is true even if we are using two screen images of Karel one for the World and one for the program.

In this lesson we will learn how we can simplify our coding, making it easier for the programmer to write and to understand. Karel will still only understand the five basic commands but we can expand Karel's vocabulary - well sort of!!!

We began the programming process with a problem statement:

Problem Statement: Karel's world contains two boxes, each with one beeper. Go out into Karel's world, retrieve the beepers from the boxes, return to the origin take beepers out of beeper bag.

Karel the Robot

Extending the Primitive Commands

We then defined the end result of the program (Output) and the starting state of our program (Input)

Define the Output

Return to Origin with two beepers in beeper bag
Place two beepers at Origin

Define the Input

Start at Origin, facing North
There are NO beepers in the beeper bag

Working with a problem statement, output and input we defined the initial version of our algorithm, our initial solution to Karel's problem.

Karel starts at origin with no beepers
Move Karel until one block North of first beeper box
Turn right
Move to above first beeper box
Turn to face down
Pick up beeper
Turn around
Move one block
Turn left
Move 1 block
Turn right
Move 2 blocks
Turn right
Move 1 block
Turn to face down
Move one block
Pick up beeper
Turn around
Move one block
Go back to origin
Put down two beepers
Turn off.

Looking over this algorithm, we recognized that few if any of these statements were at a level that Karel could understand. Our next task was to translate the initial algorithm into Karel's primitive commands.

Our initial program looked something like this:

```
beginning-of-program
  beginning-of-execution
    move;
    move;
    move;
    turnleft;
    turnleft;
    turnleft;
    move;
    move;
    turnleft;
    turnleft;
    turnleft;
    move;
    pickbeeper;
    turnleft;
    turnleft;
    move;
    turnleft;
```

Karel the Robot

Extending the Primitive Commands

```

move;
turnleft;
turnleft;
turnleft;
move;
move;
turnleft;
turnleft;
turnleft;
move;
turnleft;
turnleft;
turnleft;
move;
pickbeeper;
turnleft;
turnleft;
move;
turnleft;
move;
move;
turnleft;
move;
move;
move;
move;
move;
turnleft;
turnleft;
putbeeper;
putbeeper;
turnoff;
end-of-execution
end-of-program

```

While this certainly solved Karel's problem statement, the constant repetition of commands to do simple tasks such as turn right or turn around made the process very tedious.

Defining New Instructions:

Since Karel is very good at following instructions, his designers included the concept of creating new instructions which simplifies the program for his human handlers but from Karel's perspective still uses just the 5 basic instructions.

Here is the structure of a Karel program. Notice that Karel's basic instructions are located between the Beginning-of-Execution and End-of-Execution. The definitions of new terms are placed between the Beginning-of-program and the Beginning-of-Execution.

BEGINNING-OF-PROGRAM

[Definitions

BEGINNING-OF-EXECUTION

[Instructions

END-OF-EXECUTION

END-OF-PROGRAM

Karel the Robot

Extending the Primitive Commands

Creating a new instruction is not a difficult task, and it is not complex. It **does** require care and attention to detail. The benefit is that the program can be much more natural and easy to understand.

To create a new instruction for Karel we use this command:

```
DEFINE-NEW-INSTRUCTION <new name here> AS
    begin
    <instruction(s);>
    end;
```

The **begin** and **end** indicate where the definition starts and finishes. **For every begin, there must be a corresponding end.**

Let's try a simple new definition:

```
DEFINE-NEW-INSTRUCTION turnright AS
    begin
        turnleft;
        turnleft;
        turnleft;
    end;
```

Our Karel programming language now contains the five primitives and the new instruction "turnright". The following version of Karel's two box program uses turnright instead of three turnleft commands.

```
beginning-of-program
    define-new-instruction turnright as
    begin
        turnleft;
        turnleft;
        turnleft;
    end;
beginning-of-execution
    move;
    move;
    move;
    turnright;
    move;
    move;
    turnright;
    move;
    pickbeeper;
    turnleft;
    turnleft;
    move;
    turnleft;
    move;
    turnright;
    move;
    move;
    turnright;
    move;
    turnright;
    move;
    pickbeeper;
```

Karel the Robot

Extending the Primitive Commands

```

turnleft;
turnleft;
move;
turnleft;
move;
move;
turnleft;
move;
move;
move;
move;
move;
turnleft;
turnleft;
putbeeper;
putbeeper;
turnoff;
end-of-execution
end-of-program

```

Not only is this program shorter than the earlier version but it is easier to understand. Definitions are not standardized. The programmer can create them however the programmer chooses. For example, the programmer might have decided to define the term turn-180 (which means turn-around) and then use turn-180 to define turnaround. The basic program would be unchanged, but the definitions section would contain two definitions one of which uses the other. This is perfectly legal. The definitions section would look like:

```

beginning-of-program
define-new-instruction turn-180 as
begin
  turnleft;
  turnleft;
end;
define-new-instruction turnright as
begin
  turn-180;
  turnleft;
end;
beginning-of-execution

```

Notice that the definition of turnright makes use of the turn-180 command. The most important concept here is that a definition must be defined before it is used. So turn-180 had to be defined before turnaround.

Using the turn-180 definition we can simplify our initial program further making it easier to read and understand. From Karel's point of view nothing has changed. Karel is still using only his primitive commands. From the programmer's view the programming process is becoming much simpler.

```

beginning-of-program
define-new-instruction turn-180 as
begin
  turnleft;
  turnleft;
end;
define-new-instruction turnright as
begin
  turn-180;
  turnleft;
end;
beginning-of-execution
  move;

```

Karel the Robot

Extending the Primitive Commands

```

move;
move;
turnright;
move;
move;
turnright;
move;
pickbeeper;
turn-180;
move;
turnleft;
move;
turnright;
move;
move;
turnright;
move;
turnright;
move;
pickbeeper;
turn-180;
move;
turnleft;
move;
move;
turnleft;
move;
move;
move;
move;
move;
turn-180;
putbeeper;
putbeeper;
turnoff;
end-of-execution
end-of-program

```

Reducing Repeated Instructions:

Looking at the program above, there is still a lot of repetition.

The Iterate (which means repeat) command can be used to simplify code where instructions are repeated over and over again. In the above program, if we were to bring Karel home, our instructions would look like this.

```

beginning-of-program
  define-new-instruction turn-180 as
  begin
    turnleft;
    turnleft;
  end;
  define-new-instruction turnright as
  begin
    turn-180;
    turnleft;
  end;
beginning-of-execution
  move;
  move;
  move;
  turnright;
  move;

```


Karel the Robot

Extending the Primitive Commands

```

move;
turnright;
move;
pickbeeper;
turn-180;
move;
turnleft;
move;
turnright;
move;
move;
turnright;
move;
turnright;
move;
pickbeeper;
turn-180;
move;
turnleft;
move;
move;
turnleft;
move;
move;
move;
move;
turn-180;
putbeeper;
putbeeper;
turnoff;
end-of-execution
end-of-program

```



Notice, there are a few places in the code where commands are repeated. The Iterate command can be used to repeat a set of instructions (or just one instruction) a number of times.

The Iterate command has the following structure:

```

Iterate <some number> times
begin
  Instruction(s);
end;

```

The Iterate command can be used within a define a new instruction command or directly within the code. Here is an example of the program we have been using where the Iterate command replaces the many repetitions of the move command.

Looking at the compiled version of the code we can see that Karel's designers recognized the Iterate command as a repeat or loop and show us this visually in the compiled code. There is an oval grouping the iterate statements together. Anytime you repeat something in your code you are having the computer perform a loop.

Karel the Robot Extending the Primitive Commands

```

1 beginning-of-program
2  ┌ define-new-instruction turn-180 as
3  │ begin
4  │ ┌ turnleft;
5  │ └ turnleft;
6  │ └ end;
7  └ define-new-instruction turnright as
8  │ begin
9  │ ┌ turn-180;
10 │ └ turnleft;
11 │ └ end;
12 └ beginning-of-execution
13   ┌ iterate 3 times
14   │ begin
15   │ ┌ move;
16   │ └ end;
17   │ └ turnright;
18   │ ┌ move;
19   │ └ move;
20   │ └ turnright;
21   │ ┌ move;
22   │ └ pickbeeper;
23   │ └ turn-180;
24   │ ┌ move;
25   │ └ turnleft;
26   │ ┌ move;
27   │ └ turnright;
28   │ ┌ move;
29   │ └ move;
30   │ └ turnright;
31   │ ┌ move;
32   │ └ turnright;
33   │ ┌ move;
34   │ └ pickbeeper;
35   │ └ turn-180;
36   │ ┌ move;
37   │ └ turnleft;
38   │ ┌ move;
39   │ └ move;
40   │ └ turnleft;
41   │ ┌ iterate 5 times
42   │ │ begin
43   │ │ ┌ move;
44   │ │ └ end;
45   │ │ └ turn-180;
46   │ │ └ putbeeper;
47   │ │ └ putbeeper;
48   │ │ └ turnoff;
49   └ end-of-execution
50 end-of-program

```

The diagram highlights two loops in the code. The first loop, labeled "LOOP", is defined by the "iterate 3 times" block (lines 13-16) and includes the following instructions: `move;` (line 15), `turnright;` (line 17), `move;` (line 18), `move;` (line 19), `turnright;` (line 20), `move;` (line 21), `pickbeeper;` (line 22), `turn-180;` (line 23), `move;` (line 24), `turnleft;` (line 25), `move;` (line 26), `turnright;` (line 27), `move;` (line 28), `move;` (line 29), `turnright;` (line 30), `move;` (line 31), `turnright;` (line 32), `move;` (line 33), `pickbeeper;` (line 34), `turn-180;` (line 35), `move;` (line 36), `turnleft;` (line 37), `move;` (line 38), and `move;` (line 39). The second loop, also labeled "LOOP", is defined by the "iterate 5 times" block (lines 41-44) and includes the following instructions: `move;` (line 43) and `turn-180;` (line 45).