2 Primitive Instructions and Simple Programs

This chapter begins our study of the robot programming language. We will start with a detailed explanation of the primitive instructions that are built into every robot's vocabulary. Using these instructions, we can instruct any robot to move through the world and handle beepers. Section 2.6 shows a complete robot program and discusses the elementary punctuation and grammar rules of the robot programming language. By the end of this chapter we will be able to write programs that instruct robots to perform simple obstacle avoidance and beeper transportation tasks.

Before explaining the primitive instructions of the robot programming language, we must first define the technical term execute: A robot executes an instruction by performing the instruction's associated action or actions. The robot executes a program by executing a sequence of instructions that are given to it by the helicopter pilot. Each instruction in such a sequence is delivered to the robot in a message, which directs one robot to perform one instruction in the program.

2.1 Changing Position

Every robot understands two primitive instructions that change its position. The first of these instructions is **move** , which changes a robot's location.

move

> When a robot is sent a **move** message it executes a **move** instruction and moves forward one block; it continues to face the same direction. To avoid damage, a robot will not move forward if it sees a wall section or boundary wall between its current location and the corner to which it would move. Instead, it turns itself off. This action, called an error shutoff, will be explained further in Section 2.7.

From this definition we see that a robot executes a **move** instruction by moving forward to the next corner. However, the robot performs an error shutoff when its front is blocked. Both situations are illustrated in Figure 2-1. Figure 2-1 shows the successful execution of a **move** instruction. The wall section is more than one half-block away and cannot block this robot's move.

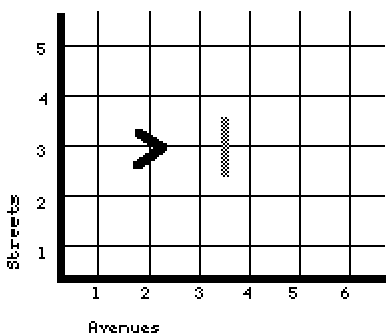

Figure 2-1 A: A Robot in the Initial Situation Before a move Instruction

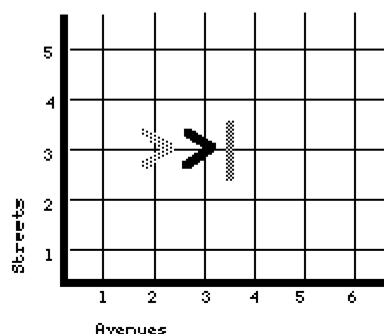

Figure 2-1 B: A Robot in the Final Situation after Executing a move Instruction

In contrast, Figure 2-2 shows an incorrect attempt to move. When this robot tries to execute a **move** instruction in this situation, it sees a wall section. Relying on its self-preservation instinct, it performs an error shutoff.

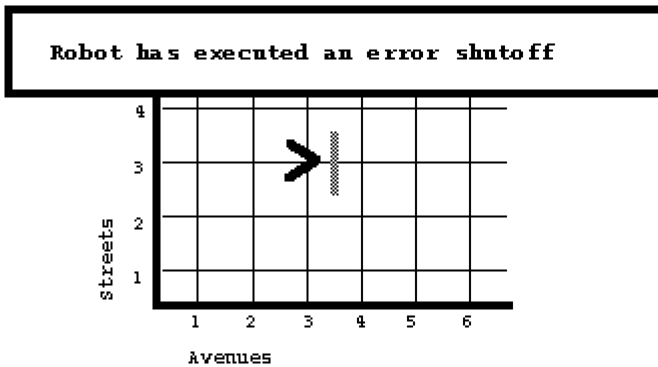

Figure 2-2 The Result of a Robot Attempting to move When Its Front Is Blocked In an Error Shutoff.

2.2 Turning in Place

The second primitive instruction that changes a robot's position is **turnLeft** . This instruction changes the direction in which the robot is facing but does not alter its location.

turnLeft

> When a robot is sent a **turnLeft** message it executes a **turnLeft** instruction by pivoting ,90 to the left. The robot remains on the same street corner while executing a **turnLeft** instruction. Because it is impossible for a wall section to block a robot's turn, **turnLeft** cannot cause an error shutoff.

A robot always starts a task on some corner, facing either north, south, east, or west. A robot cannot travel fractions of a block or turn at other than 90 angles. Although **move** and **turnLeft** change the robot's position, after executing either of these instructions, the robot still is on some corner and still is facing one of the four compass directions.

Karel-Werke's designer purposely did not provide a built-in **turnRight** instruction. Would adding a **turnRight** to the primitive instructions allow the robot to perform any task it cannot accomplish without one? A moment's thought-and the right flash of insight-shows that the **turnRight** instruction is unnecessary; it does not permit robots to accomplish any new tasks. The key observation for verifying this conclusion is that a robot can manage the equivalent of a **turnRight** instruction by executing three **turnLeft** instructions.

2.3 Finishing a Task

We need a way to tell a robot that its task is finished. The **turnOff** instruction fulfills this requirement.

turnOff

> When a robot is sent a **turnOff** message, it executes a **turnOff** instruction. It turns off and is incapable of executing any more instructions until restarted on another task. The last instruction executed by every robot in a program must be a **turnOff** instruction.

2.4 Handling Beepers

Every robot understands two instructions that permit it to handle beepers. These two instructions perform opposite actions.

pickBeeper

When a robot is sent a **pickBeeper** message, it executes a **pickBeeper** instruction. It picks up a beeper from the corner on which it is standing and then deposits the beeper in its beeper-bag. If a **pickBeeper** instruction is attempted on a beeperless corner, the robot performs an error shutoff. On a corner with more than one beeper the robot picks up one, and only one, of the beepers and then places it in the beeper-bag.

putBeeper

When a robot is sent a **putBeeper** message, it executes a **putBeeper** instruction by extracting a beeper from its beeper-bag and placing the beeper on the current street corner. If a robot tries to execute a **putBeeper** instruction with an empty beeper-bag, the robot performs an error shutoff. If the robot has more than one beeper in its beeper-bag, it extracts one, and only one, beeper and places it on the current corner.

Beepers are so small that robots can move right by them; only wall sections and boundary walls can block a robot's movement. Robots are also very adept at avoiding each other if two or more show up on the same corner simultaneously.

2.5 Robot Descriptions

All robots produced by Karel-Werke have at least the capabilities just described. As we will see, such robots are very primitive, and we might like robots with additional abilities. Therefore, we must have some way to describe those extra abilities so that the factory can build a robot to our specifications. Karel-Werke employs a simple robot programming language to describe both robot abilities and the lists of robot instructions, called programs. The formal name for the description of a robot instruction is **method**. The simple model of robot described above is called the **ur_Robot** (Foonote 1 ur is a German prefix meaning "original" or "primitive." The pronunciation of ur is similar to the sound of "oor" in "poor." class.) The specification, or interface, of the **ur_Robot** class in the robot programming language follows.

```
class ur_Robot
{
      void move(){...}
      void turnOff(){...}
      void turnLeft(){...}
      void pickBeeper(){...}
      void putBeeper(){...}
}
```

Following the model class name is a list of instructions (methods) for this kind of robot. The list is always written in braces **{** and **}**. The use of elipsis in the above {...} indicates that there are some things that belong here that are not being shown. These are the descriptions of how an ur_Robot would carry out each of these instructions.

The five methods, **move** through **putBeeper** , name actions that **ur_Robots** can perform. We defined each of these actions in the foregoing sections, and we will see many examples of their use throughout this book. The word **void** prefixes each of these methods to indicate that they return no feedback when executed. Later we will see additional methods that do produce feedback when executed, rather than changing the state of the robot as these instructions all do. The matching parentheses that follow the method names mark them as the names of actions that a robot will be able to carry out.

A sample task for an **ur_Robot** might be to start at the origin, facing east, and then walk three blocks east to a corner known to have a beeper, pick up the beeper, and **turnOff** on that corner. A complete program to accomplish this task is shown next. In this program we name the robot **Karel** , but we could use any convenient name.

```
task
{
      ur_Robot Karel = new ur_Robot(1, 1, East, 0);
            // Deliver the robot to the origin (1,1),
            // facing East, with no beepers.
      Karel.move();
      Karel.move();
      Karel.move();
      Karel.pickBeeper();
      Karel.turnOff();
}
```

Complete programs will be discussed in the next section.

2.6 A Complete Program

In this section we describe a task for a robot named Karel and a complete program that instructs it to perform the task. The task, illustrated in Figure 2-3, is to transport the beeper from 1st Street and 4th Avenue to 3rd Street and 5th Avenue. After Karel has put down the beeper, it must move one block farther north before turning off.

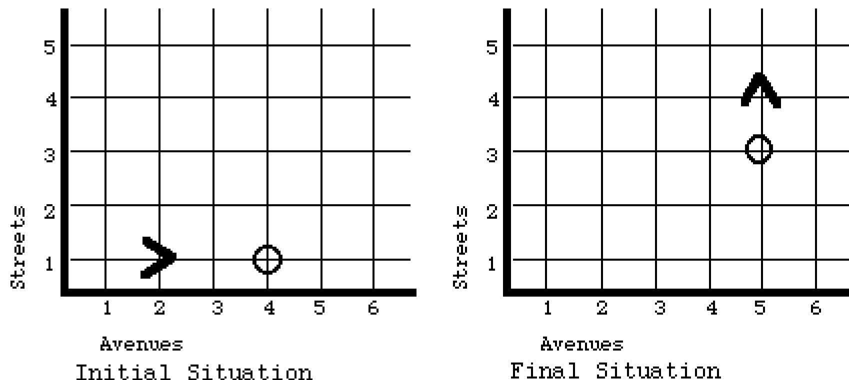

Figure 2-3 The Initial and Final Situations of Karel's task

The following program instructs Karel to perform this task. The program uses all of the methods available to robots in the **ur_Robot** class, a few new words from the robot programming vocabulary, and punctuation symbols such as the period and semicolon. We will first discuss Karel's execution of this program, and then analyze the general structure of all robot programs.

```
task
{
      ur_Robot Karel = new ur_Robot(1, 2, East, 0);
      Karel.move();
      Karel.move();
      Karel.pickBeeper();
      Karel.move();
      Karel.turnLeft();
      Karel.move();
      Karel.move();
      Karel.putBeeper();
      Karel.move();
      Karel.turnOff();
}
```

(Here is 100% Java code equivalent to the above.)

We must note that this is not the only sequence of messages that will correctly perform the stated task. Although it is obvious and direct, this is just one of many sequences that will accomplish the task.

A set of messages to one or more robots is called a task and is introduced by the special term, or reserved word, **task** . The first instruction in the main task block constructs the robot, associates the name **Karel** with it, and delivers it from the factory, ready to run, to 1st Street and 2nd Avenue, facing east with no beepers in its beeper-bag. This instruction is not, technically speaking, a message since it is not addressed to any robot. This statement can be thought of as a delivery specification. It instructs the helicopter pilot how to set up the robot when it is delivered. The delivery specification also names the specific type or class of robot that we want delivered. Here we want an **ur_Robot** .

The remaining lines of the main task block instruct Karel how to carry out the task. These messages are sent to Karel by the helicopter pilot, as described next.

2.6.1 Executing a Program

Before a program can be executed in a world, the program is read at the factory to make sure it has no errors. We will discuss errors later; for now, we will assume that our program is correct.

How is a program executed? A program execution is begun after the helicopter pilot delivers the robot to the required corner and sets it up according to the delivery specification. Here we require that the **ur_Robot Karel** be set up on 1st Street and 2nd Avenue, facing east, and with zero beepers in its beeper-bag. Then, for each additional command in the main task block, the pilot sends a corresponding electronic message to the robot named in that command. The message gives the instruction that the named robot is supposed to perform. These messages are relayed by the pilot to the robot through a special robot control satellite that hovers over the world. Since a robot can execute only one instruction at a time and since the satellite has a very limited communication capacity, only one message can be sent at a time. The pilot must wait for instruction completion before sending the next message. When the robot completes the current instruction, it sends a reply back to the pilot through the satellite indicating that the next message can be sent. Messages from the main task block are sent sequentially without omitting any messages in a strict top-to-bottom order. The pilot continues sending messages until either all messages in the main task block have been sent or the pilot attempts to send a message to a robot that has executed a **turnOff** or has performed an error shutoff.

It is also possible for robots to send messages to each other. When this occurs, the robot sending the message waits for the reply before continuing. This is to guarantee that the satellite communication channel is never overloaded.

To determine what a program does, we simulate, or trace, its execution. Simulating or tracing a robot program means that we must systematically execute the program exactly as the pilot and robots would, recording every action that takes place. We can simulate a robot program by using markers on a sheet of paper (representing robots and the world). We simulate a robot program by following the sequence of messages in the order the pilot reads them to the robot. We will discuss tracing later, but in order to become proficient robot programmers, we must understand exactly how the pilot reads the program and the robot executes it. The ability to simulate a robot's behavior quickly and accurately is an important skill that we must acquire.

Let's follow a simulation of our program. In the following simulation **(1, 4)** means 1st Street and 4th Avenue. In the following annotation we explain exactly what state the robot is left in after the execution of the instruction. Note that the symbol // (two adjacent slash characters) is used in the simulation to introduce comments into our robot programs. Each comment begins with the // mark and continues to the end of the line. These comments are ignored by the pilot and by the robots; they are included only to aid our own understanding of the program. Here we use them to explain in detail each instruction as it will be executed. We note, however, that if the program is changed in any way, the comments are likely to become invalid.

```
task
{
      ur_Robot Karel = new ur_Robot(1, 2, East, 0);
                   // A new robot named Karel is
                   // constructed and delivered to
                   // (1, 2), facing East. Karel has
                   // no beepers in its beeper-bag.

      Karel.move();         // Karel moves east
                             // to (1, 3)
      Karel.move();         // Karel moves east
                             // to (1, 4)
      Karel.pickBeeper();    // Karel picks 1 beeper,
                             // 1 beeper in bag
      Karel.move();          // Karel moves east
                             // to (1, 5)
      Karel.turnLeft();      // Karel remains on
                             // (1, 5), faces North
      Karel.move();          // Karel moves north
                             // to (2, 5)
      Karel.move;            // Karel moves north
                             // to (3, 5)
      Karel.putBeeper();     // Karel puts 1 beeper
                             // down, now 0 beepers
                             // in bag
      Karel.move();          // Karel moves north
                             // to (4, 5)
      Karel.turnOff();       // Karel remains on
                             // (4, 5) facing North
                             // and shuts off
}
```

Karel is done and we have verified that our program is correct through simulation by tracing the execution of the program.

2.6.2 The Form of Robot Programs

Now that we have seen how a robot executes a program, let's explore the grammar rules of the robot programming language. The factory and pilots pay strict attention to grammar and punctuation rules, so our time is well spent carefully studying these rules. We start by dividing the symbols in a robot program into three groups. The first group consists of special symbols. It has members such as the punctuation marks like the semicolon, the braces **{** and **}** , and the period. The next group of symbols consists of names such as robot and class names, **Karel** and **ur_Robot** . We also use names to refer to instructions, like **putBeeper** and **turnLeft** . The third and last group of symbols consists of reserved words. We have already seen a few of these like `class` and `task` .

Reserved words are used to structure and organize the primitive instructions in the robot programming language. They are called reserved words because their use is reserved for their built-in purpose. These reserved words may not be reused for other purposes in a robot program, such as robot names. To make the reading of programs easier, we may write robot programs using both upper- and lowercase letters as well as the underscore character, but we must be consistent. For example, `task` is always spelled with all lowercase letters. The robot programming language is case-sensitive, meaning that the use of upper- and lowercase letters in a word must be consistent each time the word is used. If we use the word **Task** in a robot program it would refer to something else, perhaps the name of a robot.

Since robot programs need to be read by humans as well as robots, it is helpful to be able to put explanatory material into the program itself. The language therefore permits comments to be inserted into the text of the program. As we have seen in the foregoing program, a comment begins anywhere on a line with the special symbol // (two slash marks with no space between). The comment terminates only when the line does. Anything may be put on the line following the comment symbol.

Every robot program consists of a single task for one or more robots. This main task block is introduced by the reserved word `task` and is enclosed in curly brace punctuation marks. Notice that the opening brace must be matched eventually by a closing brace. Matching pairs of braces are called delimiters, because they mark, or delimit, the beginning and end of some important entity.

If we needed specialized robots to perform various parts of the task, the class declarations of those robots would precede the task list, as would the definitions of any new instructions named in the class declarations. We will go into this in detail in Chapter 3.

The main task block itself starts with a list of definitions, called declarations. In the following program we have only one declaration, which declares that the name **Karel** will be used as the name of a robot in class **ur_Robot** . Declarations introduce new names and indicate how they will be used in the rest of the program. The declarations of robot names always end with a semicolon. We could also declare names for several different robots, even robots of different classes. The declarations of robots can best be thought of as delivery specifications to the factory. They always contain information about how the robot should be placed in the world.

Every program has one main task block. Each of the statements in the main task block is terminated by a semicolon. Most of the statements in the main task block are messages to the robots declared in the declaration list. The one exception here is the delivery instruction, which causes the factory to construct

and deliver a new **ur_Robot** named Karel to 1st Street and 2nd Avenue (we always list streets first), facing east, with no beepers in its beeper-bag. When delivered, the robot is set up and ready to receive messages sent to it. Since robots are delivered by the factory in helicopters, we don't need to be concerned about walls in the world that might impede delivery to any corner. The helicopter will be able to fly over them.

We can send messages to several different robots from the same main task block, so we need to specify which robot is to carry out each instruction. Thus, if we have a robot named Karel and want it to move, we send the message **Karel.move()** . This seems redundant here when there is only one robot, but it is required nevertheless. An instruction that causes a robot to perform one of its own instructions, such as move, is known as a message statement. The instruction named in a message statement (**move**) is called the message, and the robot (**Karel**) is the receiver of the message. Messages are the means of getting a robot to execute an instruction.

Execution always begins with the first instruction following the reserved word **`task`** . Robots are not automatically shut down at the final closing brace in a program; the **turnOff** instruction must be used for that purpose. The closing brace marks the end of the instructions that will be executed. If we reach the end of the instructions in the main task block and any robot is still on because it hasn't yet executed a **turnOff** instruction, it means that at least one **turnOff** instruction has been omitted from the program, and any robot still on will report an error.

Observe that the program is nicely indented as well as commented. It is well organized and easy to read. This style of indenting, as well as the comments, is only for the benefit of human readers. The following program is just as easily executed as the previous program.

```
   task{  ur_Robot Karel = new ur_Robot(1,2, East,0); Karel.move();
Karel.move(); Karel.pickBeeper(); Karel.move();
Karel.turnLeft(); Karel.move(); Karel.move();
Karel.putBeeper(); Karel.move(); Karel.turnOff()};
```

As this example illustrates, the importance of adopting a programming style that is easy to read by humans cannot be overemphasized.

2.7 Error Shutoffs

When a robot is prevented from successfully completing the action associated with a message, it turns itself off. This action is known as an error shutoff, and the effect is equivalent to receiving a **turnOff** message. However, turning off is not the only way such a problem could be addressed. An alternative strategy could have the robot just ignore any message that cannot be executed successfully. Using this strategy the robot could continue executing the program as if it had never been required to execute the unsuccessful instruction.

To justify the choice of executing an error shutoff, rather than just ignoring messages in such situations, consider the following: Once an unexpected situation arises-one that prevents successful execution of an instruction-a robot probably will be unable to make further progress toward accomplishing the task. Continuing to execute a program under these circumstances will lead to an even greater discrepancy between what the programmer had intended for the robot to do and what it is actually doing. Consequently, the best strategy is to have the robot turn off as soon as the first inconsistency appears.

So far, we have seen three instructions that can cause error shutoffs: **move** , **pickBeeper** , and **putBeeper** . We must construct our programs carefully and ensure that the following conditions are always satisfied.

- A robot executes a **move** instruction only when the path is clear to the next corner immediately in front of it.
- A robot executes a **pickBeeper** instruction only when it is on the same corner as at least one beeper.
- A robot executes a **putBeeper** instruction only when the beeper-bag is not empty.
- A robot executes a **turnOff** instruction at the end of each program.

We can guarantee that these conditions are met if, before writing our program, we know the exact initial situation in which the robot will be placed.

2.8 Programming Errors

In this section we classify all programming errors into four broad categories. These categories are discussed using the analogy of a motorist with a task in the real world. It should help clarify the nature of each error type. You might ask, "Why spend so much time talking about errors when they should never occur?" The answer to this question is that programming requires an uncommon amount of precision, and although errors should not occur in principle, they occur excessively in practice. Therefore we must become adept at quickly finding and fixing errors by simulating our programs.

A lexical error occurs whenever the robot program contains a word that is not in its vocabulary. As an analogy, suppose that we are standing on a street in San Francisco and we are asked by a lost motorist, "How can I get to Portland, Oregon?" If we tell the motorist, "fsdt jkhpy hqngrpz fgssj sgr ghhgh grmplhms," we commit a lexical error. The motorist is unable to follow our instructions because it is impossible to decipher the words of which the instructions are composed. Similarly, the robot executing a program must understand each word in a program that it is asked to execute.

Here is a robot program with some lexical errors:

```
taxt                  // misspelled reserved word
{
  ur_Robot Karel(1, 2, East, 0) ; // missing new...
  Karel.move();
  Karel.mvoe();       // misspelled instruction
  Karel.pick();       // unknown word
  Karel.move();
  Karel.turnright();  // unknown word
  Karel.turn-left();  // unknown word
  Karel.turnleft();   // unknown word
  Karel.move();
}
```

The last error occurs because the robot programming language is case-sensitive. The word turnLeft is not the same as turnleft.

Even if the pilot recognizes every word in a program, the program still might harbor a syntax error. This type of error occurs whenever we use incorrect grammar or incorrect punctuation. Going back to our lost motorist, we might reply, "for, Keep hundred. just miles going eight." Although the motorist recognizes

each of these words individually, we have combined them in a senseless, convoluted manner. According to the rules of English grammar, the parts of speech are not in their correct positions. We discussed the grammar rules for basic robot programs in Section 2.6.2.

The following program contains no lexical errors, but it does have syntax errors.

```
ur_Robot Karel = new ur_Robot(1,1,East,0); // declaration not in main task block
task
  Karel.move();              // missing brace
  move();                    // not addressed
                             // to any robot
  Karel.pickBeeper;          // no ()
  Karel move();              // missing period
  Karel.turnLeft()           // missing semicolon
  Karel.move();
  Karel.move();
  Karel.putBeeper();
  Karel.move();
};                           // extra semicolon
  Karel.turnOff()            // message outside
                             // task block and
                             // missing semicolon
```

If our program contains lexical or syntax errors, the factory will discover them when our program is checked there. In both cases, the factory has no conception of what we meant to say; therefore, it does not try to correct our errors. Instead, the factory informs us of the detected errors and doesn't build the robot. This action is not an error shutoff, for in this case the robot never has a chance to begin to execute the program. While discussing the next two categories of errors, we will assume that the factory finds no lexical or syntax errors in our program, so it builds the robot and the pilot delivers it and begins to execute the program.

The third error category is called an execution error. Unlike lexical and syntax errors, which are detected at the factory, the pilot can only detect these errors while the program is running or during a simulation of its execution. Execution errors occur whenever a robot in the world is unable to execute an instruction successfully and is forced to perform an error shutoff. Returning to our motorist, who is trying to drive from San Francisco to Portland, we might say, "Just keep going for eight hundred miles." But if the motorist happens to be facing west at the time, and takes our directions literally, the motorist would reach the Pacific Ocean after traveling only a few miles. At this point, the motorist would halt, realizing that he or she cannot follow our instructions to completion.

Likewise, a robot turns off if asked to execute an instruction that it cannot execute successfully. Instructing a robot to **move** when the front is blocked, to **pickBeeper** on a corner that has no beeper, and to **putBeeper** when the beeper-bag is empty are examples of execution errors, and each one results in an error shutoff.

The final error class is the most insidious, because pilots, the factory, and robots cannot detect this type of error when it occurs. We label this category of error an intent error. An intent error occurs whenever the program successfully terminates but does not successfully complete the task. Suppose our motorist is facing south when we say, "Just keep going for eight hundred miles." Even though these instructions can be successfully followed to completion, the motorist will end up somewhere in Mexico, rather than Oregon.

Here is an example of an intent error in a robot program: Beginning in the situation shown in Figure 2-4, Karel is to pick up the beeper, move it one block to the north, put the beeper down, move one more block to the north, and **turnOff** .

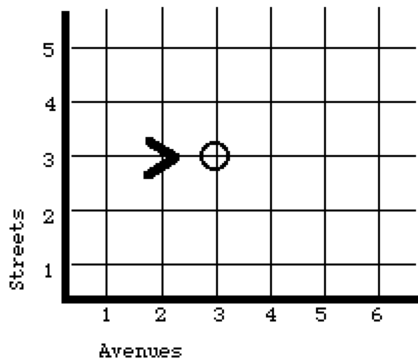

Figure 2-4 Karel's Initial Situation

```
task
{ ur_Robot Karel = new ur_Robot(3, 2, East, 0);
  Karel.move();
  Karel.pickBeeper();
  Karel.move();
  Karel.turnLeft();
  Karel.putBeeper();
  Karel.move();
  Karel.turnOff();
}
```

There are no lexical, syntax, or execution errors in this program. As far as Karel and the helicopter pilot are concerned, when the **turnOff** is executed, everything is perfect. However, look at the task and look at the program. What is the error? The task is to move the beeper one block to the north, yet Karel moved the beeper one block to the east. The intent was a northerly move, but the final result was an easterly move. The program does not satisfy the requirements of the stated task and thus contains an error of intent.

Remember that a robot does not understand the task for which we have programmed it. All that the robot can do is execute the instructions corresponding to messages we have sent it in our program. Thus, there is no way for a robot to know that the program did not accomplish what we intended. Similarly, the pilot has no way to know what we intended. He or she only knows what is actually written in the program itself.

2.8.1 Bugs and Debugging

In programming jargon, all types of errors are known as bugs. There are many apocryphal stories about the origin of this term. In one story the term bug is said to have been originated by telephone company engineers to refer to the source of random noises transmitted by their electronic communications circuits. Another story originated with the Harvard Mark I Computer and Grace Murray Hopper, later Admiral. The computer was producing incorrect answers, and when engineers took it apart trying to locate the problem, they found a dead moth caught between the contacts of a relay, causing the malfunction: the first computer bug. Other stories abound, so perhaps we shall never know the true entomology of this word.

Perhaps the term bug became popular in programming because it saved the egos of programmers. Instead of admitting that their programs were full of errors, they could say that their programs had bugs in them. Actually, the metaphor is apt; bugs are hard to find, and although a located bug is frequently easy to fix, it is difficult to ensure that all bugs have been found and removed from a program. Debugging is the name that programmers give to the activity of removing errors from a program.

2.9 A Task for Two Robots

We are not restricted to using only a single robot to perform a task. We can have as many as we like. We shall see in later chapters that robots can communicate in sophisticated ways. For now, here is a simple task for two robots.

Karel is at 3rd Street and 1st Avenue on a corner with a beeper, facing East. Carl is at the origin facing East. Karel should carry the beeper to Carl and put it down. Carl should then pick it up and carry it to 1st Street and 3rd Avenue. The beeper should be placed on this corner.

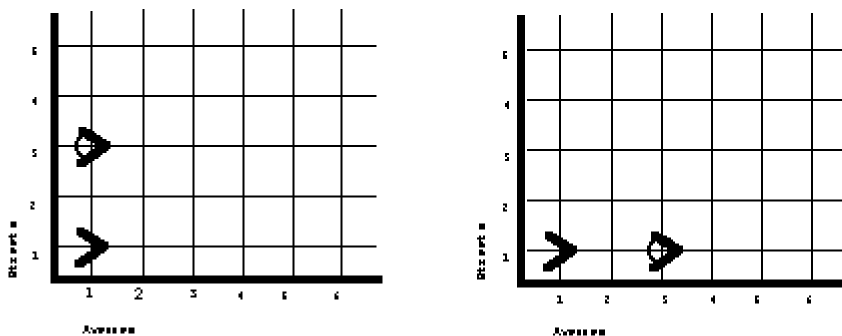

Figure 2-5. Initial and Final Situations for the Two Robot Task.

```
task
{       ur_Robot Karel = new ur_Robot(3, 1, East, 0);
        ur_Robot Carl  = new ur_Robot(1, 1, East, 0);

        Karel.pickBeeper();
        Karel.turnLeft();
        Karel.turnLeft();
        Karel.turnLeft();
        Karel.move();
        Karel.move();
        Karel.putBeeper();
        Carl.pickBeeper();
        Carl.move();
        Carl.move();
        Carl.putBeeper();
        Karel.turnOff();
        Carl.turnOff();
}
```

2.10 An infinity of Beepers

We note for completeness, though we can't use the information yet, that a robot can be delivered with infinitely many beepers in its beeper-bag. If such a robot puts down a beeper or picks a beeper, the number of beepers in the beeper-bag does not change. It is still infinity.

```
ur_Robot Karel = new ur_Robot(3, 2, East, infinity);
```

We will use this information in later chapters. It is also possible, though rare, for a corner to contain an infinite number of beepers. Since programs are finite, however, there is no way to put down or pick up an infinite number of beepers.