Karel++ Translator/Simulator

Joseph Bergin

Introduction

This system is intended to permit execution of programs that conform to the language defined in the book *Karel++: A Gentle Introduction to Object-Oriented Programming*, by Bergin, Stehlik, Roberts, and Pattis. A previous simulator has ceased to work on recent (32-bit only) Microsoft Windows systems and this is intended as a replacement for that.

More recently, a new book has been published by the same authors: *Karel J Robot: A Gentle Introduction to Object-Oriented Programming in Java*. The simulator for the new book is quite nice and is written in Java, so should run anywhere. The new book may be obtained from http://www.cafepress.com/kareljrobot. It is described, along with its simulator at http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html. This system permits the newer simulator to be used with the older book.

The Karelplusplus.jar file in this package contains a translator for the Karel++ language into a form of Java that is acceptable to the Karel J Robot simulator. You can therefore run Karel++ code (after translation) on the newer simulator.

Using the System

There are two ways to use the translator. First, you can just double click the jar file to bring up a small GUI that will let you choose (or type) the file to be translated and name the listing and world files. The Compile Button will then translate the input file into four output files as explained below.

Double clicking the jar is equivalent to typing the following command (when the working directory is current):

```
java -jar Karelplusplus.jar
```

The second way to run it is to use a command line, and enter a command such as one of the following:

```
java -cp .;Karelplusplus.jar karelplusplus.KarelCompiler myfile.r
listing.txt wld.kwld
```

or

```
java -cp .;Karelplusplus.jar karelplusplus.KarelCompiler
```

In the latter case you will be prompted for the names of the three input files. The above assumes that myfile.r contains a Karel++ program. A listing with errors (if any) will be put into the listing.txt file. Use whatever name you like for this. The third file (here wld.kwld) is the name of a world file. This file need not exist when you do the translation, but must by the time you run the simulator.

(On a UNIX or Linux system or Macintosh, the single semicolon in the above must be changed to a colon. On some versions of Java, the cp option must be spelled out as classpath. Note also that Java systems are case sensitive, even if your system is not. So karelplussplus.KarelCompiler must be capitalized exactly like this.)

Suggestions for use.
(a) put one Karel++ class in each file and translate them separately. Don't put your task in these files.
(b) put the task in a separate file and translate it.
(c) don't bother with #include. Includes wont be used and the translated file won't need them anyway.

The translator produces four files. First is a listing, including any errors. Next is a Java file that has for a name, either
(a) the name of your original file up to the first period with .java appended, provided your input file has a task block.
or
(b) the name of the last class in your file with .java appended, provided your input file does NOT have a task block.

The third and fourth files produced are a batch and a shell file for use (respectively) on windows and unix systems. Each contains commands that will use the Java compiler to translate the produced Java file AND will run it in the Karel J Robot simulator if there is a task in the input file. The latter files are called run.bat and run.sh, respectively. On unix, you need to make the file executable with chmod, of course.

Installing the System

To use this translator you should establish a working directory on your system and install the Karelplusplus.jar file into it. This jar is the translator.

You should also put the KarelJRobot.jar into the same place. This jar is the simulator and world builder. There is a user's manual for it as well, though it is directed at the Java/Karel J Robot programmer, not the Karel++ programmer.

You also need to correctly install the Java JDK from sun on your system. Do a standard installation of the JDK. Note that old versions of the JDK should be fine.

Once everything is in place you should work from your working directory. If you want to work from the command line, you should cd to this directory to give the required commands. You can also double click the translator jar and the resulting batch/shell files in most cases. The distribution of this includes a couple of Karel++ programs (with extension .r) that you can use to test the installation.

If you want a world in which your Karel++ program can run you have an additional step. If you double click the KarelJRobot.jar file it will bring up the world builder. There is a Simulator User's Manual for this on my Karel J Robot pages. You can also download a complete set of worlds representing figures in the book from the same place, though their names are keyed to the Karel J Robot book not Karel++. There is one world file included, that will work with the sample robot programs. Note that world files are just text files. We use the extension.kwld, but that is not necessary.

Notes

(1) The C++ way to get access to a method of a superclass is classname::methodname(); In this system the classname is always interpreted as the name of the most direct superclass, not a more remote ancestor class. In a rare case, this can change the meaning of your program. Java doesn't permit looking into more remote ancestors, in fact.

(2) The system does generally a poor job of recovering from syntax errors. Usually you can find the first one from the report, but it won't give a lot of information to help you fix it. It should correctly report if you misspell a method name or a class or robot name, however. In general its error reporting is not very good yet.

(3) The translator is strict about requiring braces for the parts of all if, if/else, loop, and while instructions. This is consistent with the usage in the book.

(4) In the GUI compiler there is a text field in which you can put a compile option. The J option is already entered there. When this is set, the compiler tries to show you in the listing file, the Java code that is being generated. If you use the command line version, you can give this option after the three file names (you need to list all of them) by preceding the letter J with either a / or a -. There should be no space after the "option character." Within the Karel++ file itself, you can also turn this code listing on or off by putting $J+ or $J- at the beginning of the file. Note that the option within the file overrides anything you do when running the program either with the GUI or command line version.

(5) In both versions of this program, you can run the resulting batch/shell file automatically. In the GUI version, put the letter r or R into the compile option field (alone or with the J). For the command line version append one of the following to the end of your command: /r /R -r -R. If you use the simple (prompted form) of the command line compiler you will be asked if you want to "Run the result?[yN]" If you answer with any string that starts with y (or Y) the resulting batch file will be run after compilation. The default is no. Note that in all cases, the batch/shell file is run only if your Karel++ program compiles with no errors. If you use this option, the Java and batch files are still created and written as usual for later use.

(6) Comments in the Karel++ version of your program are not carried into the Java version. The translator accepts both // and /* ... */ comment forms.

(7) The more esoteric things in Chapter 6 should work fine, but the translator is not very strict about requiring the tokens * and ->. It tries to do the right thing, however, especially with simple cases, as in the second sample below.

(8) All of the Java code produced is put into the package karel.

Sample

Here is a sample Karel++ program taken from the text. I called the file MarkPickup.r

```
class Harvester : ur_Robot
{
     void harvestTwoRows();
     void positionForNextHarvest();
     void turnRight();
     void harvestOneRow();
     void harvestCorner();
     void goToNextRow();
};

void Harvester:: goToNextRow()
{
     turnLeft();
     move();
     turnLeft();
}

void Harvester:: harvestOneRow()
{
     harvestCorner();
     move();
     harvestCorner();
     move();
     harvestCorner();
     move();
     harvestCorner();
     move();
     harvestCorner();
}

void Harvester:: harvestCorner()
{
     pickBeeper();
}

void Harvester :: positionForNextHarvest()
{
     turnRight();
     move();
     turnRight();
}

void Harvester :: turnRight()
{
     turnLeft();
     turnLeft();
     turnLeft();
}

void Harvester:: harvestTwoRows()
{
     harvestOneRow();
     goToNextRow();
     harvestOneRow();
}

task
{    Harvester Mark(2, 2, East, 0);
     Mark.move();
     Mark.harvestTwoRows();
     Mark.positionForNextHarvest();
     Mark.harvestTwoRows();
     Mark.positionForNextHarvest();
     Mark.harvestTwoRows();
     Mark.move();
     Mark.turnOff();
}
```

The command to translate it is

```
java -cp .:Karelplusplus.jar karelplusplus.KarelCompiler
MarkPickup.r  listing.txt fig3-2.kwld
```

And here is the Java that was produced. It is named MarkPickup.java.

```
package karel;
```

```
import kareltherobot.*;

class Harvester extends UrRobot{
   public Harvester(int street, int avenue, Direction direction, 
int beepers)
   {   super(street, avenue, direction, beepers);
   }
   public void harvestTwoRows(){
      this.harvestOneRow();
      this.goToNextRow();
      this.harvestOneRow();
   }
   public void positionForNextHarvest(){
      this.turnRight();
      this.move();
      this.turnRight();
   }
   public void turnRight(){
      this.turnLeft();
      this.turnLeft();
      this.turnLeft();
   }
   public void harvestOneRow(){
      this.harvestCorner();
      this.move();
      this.harvestCorner();
      this.move();
      this.harvestCorner();
      this.move();
      this.harvestCorner();
      this.move();
      this.harvestCorner();
   }
   public void harvestCorner(){
      this.pickBeeper();
   }
   public void goToNextRow(){
      this.turnLeft();
      this.move();
      this.turnLeft();
   }
}

public class MarkPickup implements Directions {
   public static void main(String [] args) {
         Harvester Mark = new Harvester(2, 2, East, 0);
```

```
          Mark.move();
          Mark.harvestTwoRows();
          Mark.positionForNextHarvest();
          Mark.harvestTwoRows();
          Mark.positionForNextHarvest();
          Mark.harvestTwoRows();
          Mark.move();
          Mark.turnOff();
}

   static {
      World.readWorld("fig3-2.kwld");
      World.setVisible(true);
      World.showSpeedControl(true);
   }
}
```

The above translation also produced a listing (not shown here) and a run.bat file with contents

```
javac -d . -classpath .;KarelJRobot.jar  MarkPickup.java
java -classpath .;KarelJRobot.jar karel.MarkPickup
```

An equivalent unix shell file was also produced: run.sh.

If you execute (double click) the batch or shell file, the Java will be compiled and the simulator will run the program in the named world. If it doesn't already exist, a folder named karel will be created in your working directory. It contains the class files produced by the Java compiler (with javac) and needed by the simulator to execute your program. It might be preferable to run the batch file from the command line (just type "run") rather than double clicking it. If your original source file had errors, the resulting Java file may also have errors and you will then see them in the command window. If you just double click, you may not see them.

Note that the GUI version names its batch files slightly differently. Since you can use it to translate several files without quitting, the batch files are named run0.bat, run1.bat, etc. for the files produced. These will overwrite earlier versions, however. The shell files are named similarly.

Known Flaws

(1) If you *declare* two methods with the same name in a class, the second only will be retained. There is a warning. If you *define* a method twice all the statements of both are run together in the result. A warning is given.

(2) If you declare two classes with the same name in one file, only the second will be retained. The declarations done in the first will be discarded. A warning is given, but additional errors will probably ensue about the discarded names not being declared when you try to later define those

methods.

(3) If you use #include, the only acceptable filenames are of the form id or id.id, where each id begins with a letter and consists of letters, digits, and underscores only. Includes aren't used here, so you can just comment them out for our purposes.

Additional Notes

(1) If the batch file doesn't run your Java system is probably not correctly installed. In particular the bin directory of the installation needs to be in your Path.

(2) This translator accepts only what is described in the Karel++ text. In particular, methods may not have parameters and the only instance fields are robots. The Karel J Robot book goes beyond these limitations, however. See, especially, Chapter 4 of that newer book.

(3) The Java file can be safely edited, perhaps to add or change the world file that is used or to add additional world options as defined in the Simulator User's Guide.

(4) The "Source File", "Listing File", and "World" buttons bring up standard dialogs. The World button is only effective, however, if the world file you want already exists. If you want to incorporate a file you have not yet created, you should just type its name into the corresponding text field.

(5) If you are using this to introduce C++, there are two additional "features" of the translator. Instead of "task" you can say "main()" or "int main()". And you can put "public:" between the robot declarations at the beginning of your class definition and the method declarations. Note that the robot declarations must come before any of the methods. This is not a complete C++ compiler, of course, and attempts only the Karel++ language with these few extensions.

(6) While you can, of course, edit the produced Java file, my intention is to make this unnecessary. If you think there are things that should be put into the Java files automatically, or under control of compiler options, please let me know. The output Java has been pretty-printed to make it easy to read.

Please send additional flaws or feature requests to the author, berginf@pace.edu. I would like a copy of any program with a common or subtle syntax error that causes this to crash with an exception, such as a null pointer exception.

Additional Example

Here is an example from Chapter 6 of Karel++. There are two classes and a task defined in PutGet.r

```
class Putter: Robot
{
     void move();
};

class Getter: Robot
{
     void move();
};


void Putter::move()
{
     Robot:: move();
     if (anyBeepersInBeeperBag())
     {
          putBeeper();
     }
}

void Getter::move()
{
     Robot :: move();
     while (nextToABeeper())
     {
          pickBeeper();
     }
}


task
{    Robot  *Karel;
     Putter Lisa(1, 1, East, 100);
     Getter Tony(2, 1, East, 0);

     loop (10)
     {
          Karel = &Lisa;       // "Karel" refers to Lisa;
          loop (2)
          {
               Karel->move();  // either Lisa or Tony
               Karel = &Tony;  // "Karel" refers to Tony
          }
     }
}
```

And here is the equivalent Java produced by the translator. It shows how if, while, and loop statements are translated:

```
package karel;

import kareltherobot.*;

class Getter extends Robot{
   public Getter(int street, int avenue, Direction direction, int
beepers)
   {   super(street, avenue, direction, beepers);
   }
   public void move(){
      super.move();
      while( this.nextToABeeper() ){
         this.pickBeeper();
      }
   }
}
class Putter extends Robot{
   public Putter(int street, int avenue, Direction direction, int
beepers)
   {   super(street, avenue, direction, beepers);
   }
   public void move(){
      super.move();
      if( this.anyBeepersInBeeperBag() ){
         this.putBeeper();
      }
   }
}

public class PutGet implements Directions {
   public static void main(String [] args) {
         Robot Karel;
         Putter Lisa = new Putter(1, 1, East, 100);
         Getter Tony = new Getter(2, 1, East, 0);
         for( int i0=0;i0<10; ++i0 ){
            Karel = Lisa;
            for( int i1=0;i1<2; ++i1 ){
               Karel.move();
               Karel = Tony;
            }
         }
}

   static {
      World.readWorld("fig3-2.kwld");
```

```
        World.setVisible(true);
        World.showSpeedControl(true);
    }
}
```