Turing: A JAVA application for Turing machines

Timothy Murphy
<tim@maths.tcd.ie>
School of Mathematics, Trinity College Dublin

February 28, 2013

Contents

Abstract

The "Turing" application documented here reads in the specification of a Turing machine from a file, and then accepts input which is processed and output as it would be by the machine. We follow Donald Knuth's "Literate Programming" recipe, in which application and documentation are combined in a single "web" file.

1 Introduction

The application begins by reading in the specification of a Turing machine $\mathcal{T}$ as a sequence of quadruples

$$(i, b, a, j),$$

where i and j number the input and output states, b is the bit (0 or 1) under the scanner, and a is the action to be taken, from the choice

$$A = \{\texttt{Noop}, \texttt{Swap}, \texttt{Left}, \texttt{Right}, \texttt{Read}, \texttt{Write}\}.$$

The rules are read in from a file, with one quadruple per line. (Anything on a line beyond the quadruple is regarded as a comment, and is ignored.)

The order of the quadruples does not matter. Nor need all possible quadruples (given the number of states) be listed; those omitted are assumed to be of the form $(i, b, \texttt{Noop}, 0)$, q_0 being both initial and halting state.

Having read in the rules, the application looks for input data through the standard input. This processed as it is read, and the output is sent to the standard output.

2 The `Turing` class

Our application is written in JAVA. The definition or specification of a JAVA class can be divided into 5 parts (not all of which are necessarily, or even usually, present), namely

Instance data — the 'fields' which are defined for each object in the class, and which in turn define that object; as for example if T is an object in the *Turing* class then the field $T.tape$ holds the tape of the machine in question.

Instance methods — the routines which act on objects and their fields, as for example $T.run(\,)$ describes how the machine is to run.

Constructors — the code describing how an object in the class is *created*.

Class methods — routines not specific to a particular object in the class, as for example the *main* routine required in any JAVA application.

Class data — data shared by all the objects in the class, as for example, the constant definitions of the actions *Noop*, *Swap*, *Read*, *Write*, *Left* and *Right*.

2
```
import java.io.*;
import java.net.*;
import java.util.*;
public class Turing extends Thread
{
  ⟨ Turing instance data 8 ⟩
  ⟨ Turing instance methods 34 ⟩
  ⟨ Turing constructors 14 ⟩
  ⟨ Turing class data 5 ⟩
  ⟨ Turing class methods 41 ⟩
}
```

¶ Although a Java file can only define 1 public class, it can include other subsidiary classes, which are, however, only visible within the file.

3 ⟨ Subsidiary classes 17 ⟩

2.1 `Turing` class data

Although Java does not support **typedef**s, it may help readability if we introduce a few 'pseudo-types' with the help of *ctangle*.

4
```
format Bit int
format State int
format Action int
format Time int
#define Bit ≡ boolean
#define State ≡ int
#define Action ≡ int
#define Time ≡ int
```

¶ It is convenient to represent the 6 possible actions numerically.

5 ⟨ *Turing* class data 5 ⟩ ≡
```
final static int Noop = 0;
final static int Swap = 1;
final static int Read = 2;
final static int Write = 3;
final static int Left = 4;
final static int Right = 5;
public final static String actionName[] = {"Noop", "Swap", "Read",
    "Write", "Left", "Right", };
```
See also chunk 6.

This code is used in chunk 2.

¶ We define a number of boolean variables, which can be set by program switches. Otherwise they take the default values below.

6 ⟨ *Turing* class data 5 ⟩ +≡
```
public static boolean debugging = false;
public static boolean interactive = false;
public static boolean numericIO = false;
```

2.2 `Turing` instance data

¶ The machine 'progresses' in discrete steps, which we may think of as occurring at times $t = 0, 1, \ldots$.

8 ⟨ *Turing* instance data 8 ⟩ ≡
```
Time time;
```
See also chunks 9, 10, 11, 12, and 13.

This code is used in chunk 2.

¶ At each moment the machine is in a state

$$q = q(t) \in Q = \{q_0, \ldots, q_{N-1}\},$$

where $N = numStates$. We identify state q_i with its label i. Thus, in effect,

$$Q = \{0, 1, \ldots, numStates - 1\}.$$

9 ⟨ *Turing* instance data 8 ⟩ +≡
```
State state;
int numStates;
```

¶ Each Turing machine has an associated *tape*, that is, an object in the *Tape* class defined below.

10 ⟨ *Turing* instance data 8 ⟩ +≡
Tape tape;

¶ The rules defining the Turing Machine are organised as a Java *ArrayList*, with rule (i, b, a, j) in position $2i + b$. (Recall the $b = 0$ or 1.)

11 ⟨ *Turing* instance data 8 ⟩ +≡
static *ArrayList* < *Rule* > *rules* = **new** *ArrayList* < *Rule* > ( );

¶ Finally, the machine communicates with the 'outside world' through an input stream and an output stream.

12 ⟨ *Turing* instance data 8 ⟩ +≡
Reader in;
Writer out;

¶ By default, input and output consist of sequences of 0's and 1's. But if the *numericIO* switch is given, then these are both converted to sequences of natural numbers, following the code

$$n \mapsto \overbrace{11\ldots1}^{n\text{ 1's}}0.$$

For example the input

111101111110

is coded as

$$4, 6$$

13 ⟨ *Turing* instance data 8 ⟩ +≡
int *inBuf* = 0;
int *outBuf* = 0;

2.3 `Turing` constructors

An object in a JAVA class must be explicitly created, by using one of the *constructor* methods provided. The constructor methods all have the same name as the class — in our case *Turing*.

To construct a *Turing* we must be given an array of *Rule*s. The input and output character streams may also be specified.

14 ⟨ *Turing* constructors 14 ⟩ ≡

```
public Turing(ArrayList < Rule > rules, Reader in, Writer out)
{
  this.in = in;
  this.out = out;
  tape = new Tape( );
}
```

See also chunks 15 and 16.

This code is used in chunk 2.

¶ If no input and output streams are specified, these are assumed to be *System.in* and *System.out*, the standard Java I/O streams.

15 ⟨ *Turing* constructors 14 ⟩ +≡

```
public Turing(ArrayList < Rule > rules)
{
  this(rules, new InputStreamReader(System.in), new
      OutputStreamWriter(System.out));
}
```

¶ We give a default constructor for a 'bare' Turing machine, whose rules are presumably read in later. (We shall not use this constructor.)

16 ⟨ *Turing* constructors 14 ⟩ +≡

```
public Turing( )
{
  this(null);
}
```

3 The `Rule` class

Each rule consists of a quadruple

$$(inStat, bit, action, outState)$$

eg

$$(3, 1, \texttt{Left}, 5),$$

indicating that if the machine is in state q_3 with bit 1 under the scanner then the scanner will move left along the tape, and the state will change to q_5.

17 ⟨ Subsidiary classes 17 ⟩ ≡

```
class Rule {
  Bit bit;
  State inState;
  Action action;
  State outState;
  ⟨ Rule constructors 18 ⟩
  ⟨ Rule methods 26 ⟩
}
```

See also chunks 27 and 28.

This code is used in chunk 3.

3.1 `Rule` constructors

Each line in the 'rule file' contains a quadruple, as described above. We use the Java *StringTokenizer* routine to abstract the rule from the line.

Empty lines, and lines starting with #, are ignored. Otherwise each line must contain a valid rule. If it does not, it 'throws a *BadRuleException*'.

18 ⟨ **Rule** constructors 18 ⟩ ≡

```
Rule(String line)
throws BadRuleException {
  ⟨ Parse line 19 ⟩
  ⟨ Get inState 20 ⟩
  ⟨ Get bit 21 ⟩
  ⟨ Get action 22 ⟩
  ⟨ Get outState 23 ⟩
}
```

See also chunks 24 and 25.

This code is used in chunk 17.

19 ¶ ⟨ Parse line 19 ⟩ ≡

```
StringTokenizer st = new StringTokenizer(line, "(,)␣\t");
if (st.countTokens( ) ≠ 4) throw new BadRuleException(line);
```

This code is used in chunk 18.

20 ¶ ⟨ Get inState 20 ⟩ ≡

```
inState = Integer.parseInt(st.nextToken( ));
```

This code is used in chunk 18.

21 ¶ ⟨ Get *bit* 21 ⟩ ≡

```
int b = Integer.parseInt(st.nextToken());
if (b ≡ 0) bit = false;
else if (b ≡ 1) bit = true;
else throw new BadRuleException(line);
```

This code is used in chunk 18.

22 ¶ ⟨ Get *action* 22 ⟩ ≡

```
String act = st.nextToken().toLowerCase();
if (act.equals("noop")) action = Turing.Noop;
else if (act.equals("swap")) action = Turing.Swap;
else if (act.equals("read")) action = Turing.Read;
else if (act.equals("write")) action = Turing.Write;
else if (act.equals("left")) action = Turing.Left;
else if (act.equals("right")) action = Turing.Right;
else throw new BadRuleException(line);
```

This code is used in chunk 18.

23 ¶ ⟨ Get *outState* 23 ⟩ ≡

```
outState = Integer.parseInt(st.nextToken());
if (inState < 0 ∨ outState < 0) throw new BadRuleException(line);
```

This code is used in chunk 18.

¶ A second constructor requires all the fields to be given.

24 ⟨ **Rule** constructors 18 ⟩ +≡

```
Rule(State inState, Bit bit, Action action, State outState)
{
  this.inState = inState;
  this.bit = bit;
  this.action = action;
  this.outState = outState;
}
```

¶ Our final constructor gives an empty rule, with fields to be filled in.

25 ⟨ **Rule** constructors 18 ⟩ +≡

```
Rule()
{}
```

26 ¶ ⟨ **Rule** methods 26 ⟩ ≡

```
public String toString( )
{
  StringBuffer buf = new StringBuffer( );
  buf.append('(');
  buf.append(inState);
  buf.append(',');
  buf.append(bit ? '1' : '0');
  buf.append(',');
  buf.append(Turing.actionName[action]);
  buf.append(',');
  buf.append(outState);
  buf.append(')');
  return buf.toString( );
}
```

This code is used in chunk 17.

¶ If an error occurs when reading in the rules we throw a *BadRuleException*.

27 ⟨ Subsidiary classes 17 ⟩ +≡

```
class BadRuleException extends Exception
{
  public static final long serialVersionUID = 24362462L;
  public BadRuleException( )
  {
    super@&er( );
  }
  public BadRuleException(String s)
  {
    super@&er(s);
  }
}
```

4 The Tape class

We implement the tape as two JAVA *Stack*s, corresponding to the two halves of the tape ($i < 0$ and $i > 0$), together with an 'accumulator bit' corresponding to the visible 'square' $i = 0$.

28 ⟨ Subsidiary classes 17 ⟩ +≡

```
class Tape {
```

```
  Bit accumulator;
  private Stack < Boolean > leftTape;
  private Stack < Boolean > rightTape;
  ⟨ Tape instance methods 30 ⟩
  ⟨ Tape constructors 33 ⟩
}
```

4.1 `Tape` instance methods

30 ¶ ⟨ **Tape** instance methods 30 ⟩ ≡

```
void left( )
{
  rightTape.push(accumulator);
  if (leftTape.empty( )) leftTape.push(false);
  accumulator = (leftTape.pop( ));
}
void right( )
{
  leftTape.push(accumulator);
  if (rightTape.empty( )) rightTape.push(false);
  accumulator = (rightTape.pop( ));
}
```

See also chunk 31.

This code is used in chunk 28.

¶ We include a routine for 'printing out' the contents of the tape. This is a little awkward, since we have to pop the bits off the stacks to view them, saving them on a temporary stack so that we can restore them.

31 ⟨ **Tape** instance methods 30 ⟩ +≡

```
public String toString( )
{
  Stack < Boolean > tmpStack = new Stack < Boolean > ( );
  StringBuffer buf = new StringBuffer( );
  while (¬leftTape.empty( )) {
    BooleanO = leftTape.pop( );
    tmpStack.push(O);
  }
  while (¬tmpStack.empty( )) {
    BooleanB = tmpStack.pop( );
```

```
    buf.append(B.booleanValue( ) ? '1' : '0');
    leftTape.push(B);
  }
  buf.append('*');
  buf.append(accumulator ? '1' : '0');
  buf.append('*');
  while (¬rightTape.empty( )) {
    Boolean B = rightTape.pop( );
    buf.append(B.booleanValue( ) ? '1' : '0');
    tmpStack.push(B);
  }
  while (¬tmpStack.empty( )) {
    Boolean O = tmpStack.pop( );
    rightTape.push(O);
  }
  return "Tape:␣" + buf.toString( );
}
```

4.2 `Tape` constructors

To construct a tape we must create the two *Stack*s constituting its left and right halves.

33 ¶ ⟨ **Tape** constructors 33 ⟩ ≡

```
public Tape( )
{
  leftTape = new Stack < Boolean > ( );
  rightTape = new Stack < Boolean > ( );
  accumulator = false;
}
```

This code is used in chunk 28.

5 Running the machine

Since our *Turing* class extends the *Thread* class, we must provide instance methods *start* and *run*,

34 ⟨ *Turing* instance methods 34 ⟩ ≡

```
public void start( )
{
```

```
    time = 0;
    state = 0;
  }
```
See also chunk 35.

This code is used in chunk 2.

¶ The *run* routine is the heart of our Turing machine. Essentially, it puts the machine into an infinite loop, reading the rules and taking the appropriate action. The machine exits from the loop if and when it re-enters state 0.

35 ⟨ *Turing* instance methods 34 ⟩ +≡
```
  public void run( )
  {
    ⟨ Buffer input and output 36 ⟩
    int col = 0;
    Rule r = rules.get(0);
    while (true) {
      if (debugging) {
        ⟨ Print out rule being followed 38 ⟩
      }
      ⟨ Take appropriate action 37 ⟩
      ⟨ Move to new state 39 ⟩
      ⟨ Exit if state = 0 40 ⟩
      time++;
    }
  }
```

36 ¶ ⟨ Buffer input and output 36 ⟩ ≡
```
  BufferedReader input = new BufferedReader(in);
  PrintWriter output = new PrintWriter(out, true);
```
This code is used in chunk 35.

37 ¶ ⟨ Take appropriate action 37 ⟩ ≡
```
  int b = tape.accumulator ? 1 : 0;
  char ch = '␣';
  System.out.println("rule:␣" + r);
  System.out.println("accumulator:␣" + tape.accumulator);
  switch (r.action) {
  case Noop: break;
```

```
case Swap: tape.accumulator = ¬tape.accumulator;
  break;
case Left: tape.left( );
  break;
case Right: tape.right( );
  break;
case Read:
  if (numericIO) { }
  else {
    try {
    loop:
      while (true) {
        ch = (char) in.read( );
        if (ch ≡ '0' ∨ ch ≡ '1') break loop;
      }
    }
    catch(IOExceptione)
    {
      System.out.println("Read␣past␣end␣of␣input");
      System.exit(1);
    }
    tape.accumulator = (ch ≡ '1');
  }
  break;
case Write: System.out.println("accumulator:␣" + tape.accumulator);
  if (numericIO) { }
  else {
    output.print(tape.accumulator ? '1' : '0');
    if (++col ≥ 256) {
      output.println( );
      col = 0;
    }
  }
  break;
}
```

This code is used in chunk 35.

38 ¶ ⟨ Print out rule being followed 38 ⟩ ≡

```
System.err.println(tape);
```

This code is used in chunk 35.

39 ¶ ⟨ Move to new *state* 39 ⟩ ≡

```
int index = 2 * r.outState;
if (tape.accumulator) index++;
r = rules.get(index);
```

This code is used in chunk 35.

40 ¶ ⟨ Exit if *state* = 0 40 ⟩ ≡

```
if (r.inState ≡ 0) {
  output.println( );
  System.exit(0);
}
```

This code is used in chunk 35.

6 The main routine

Every JAVA application must have at least one class method, namely the *main*( ) program. As we see, this simply reads in the rules for a Turing machine, creates the machine, and sets it running.

41 ⟨ *Turing* class methods 41 ⟩ ≡

```
public static void main(String[]args)
{
  ⟨ Process any options 42 ⟩
  if (argc ≠ 1) {
    usage( );
    System.exit(1);
  }
  ⟨ Open rules file 43 ⟩
  Rule defaultRule = new Rule(0, false, 0, 0);
  for (int i = 0; i < 32; i++) {
    rules.add(defaultRule);
  }
  ⟨ Read in rules 44 ⟩
  int numRules = rules.size( );
  System.out.println("numRules: " + numRules);
  TuringT = new Turing(rules);
  T.run( );
}
```

See also chunk 45.

This code is used in chunk 2.

42 ¶ ⟨ Process any options 42 ⟩ ≡

```
int argc = args.length;
int optCount = 0;
for ( ; optCount < argc ∧ args[optCount].charAt(0) ≡ '-'; optCount++)
  {
  if (args[optCount].length( ) ≡ 1) continue;
  switch (args[optCount].charAt(1)) {
  case 'd': debugging = true;
    break;
  case 'n': numericIO = true;
    break;
  default: usage( );
    System.exit(1);
  }
}
argc −= optCount;
```

This code is used in chunk 41.

43 ¶ ⟨ Open rules file 43 ⟩ ≡

```
BufferedReader rulesFile = null;
try {
  rulesFile = new BufferedReader(new FileReader(args[optCount]));
}
catch(IOException e)
{
  System.err.println("Cannot␣open␣rules␣file:␣" + args[optCount]);
  System.exit(1);
}
```

This code is used in chunk 41.

44 ¶ ⟨ Read in rules 44 ⟩ ≡

```
String line = null;
int lineno = 0;
int index;
Rule r = null;
while (true) {
  try {
    line = rulesFile.readLine( );
  }
```

```
    catch(IOExceptione)
    { }
    if (line ≡ null) break;
    lineno++;
    line = line.trim( );
    if (line.isEmpty( )) continue;
    if (line.charAt(0) ≠ '(') continue;
    try {
      r = new Rule(line);
      index = 2 ∗ r.inState;
      if (r.bit) index++;
      rules.set(index, r);
      System.out.println("rule[" + index + "]: " + r.toString( ));
    }
    catch(BadRuleExceptione)
    {
      System.err.println("(line " + lineno + ")");
    }
    if (debugging) System.err.println(r);
  }
```

This code is used in chunk 41.

¶ We define one other class method, to print out a *usage* message.

45 ⟨ *Turing* class methods 41 ⟩ +≡

```
  public static void usage( )
  {
    System.err.println("Usage: Turing [switches] rulesFile ");
    System.err.println("switches: -d (debug)");
    System.err.println("switches: -i (interactive)");
  }
```

7 Cweb and the 'j' switch

Knuth's original '*web*' format ([?]) was tied to Pascal. Later Knuth and Levy developed '*cweb*' (knuth3) to provide output in C. Since Java is a dialect of C, *cweb* only requires minor modifications to output Java. These are contained in the change files *ctang−java.ch*, *cweav−java.ch* and *comm−java.ch*, all of which can be found at `ftp://ftp.maths.tcd.ie/pub/TeX/javaTeX`. If *ctangle* and *cweave* are compiled with these change files (as

eg by modifying the *cweb Makefile* by changing the line '`TCHANGES =`' to '`TCHANGES` = *ctang* − *java.ch*', and similarly for `WCHANGES` and `CCHANGES`). then the '+*j*' switch can be used to output JAVA, eg

```
% ctangle +j Turing.w
```

produces the file *Turing.java* which can then be compiled in the usual way

```
% javac -O Turing.java
```

and then run by

```
% java Turing
```

On the other hand, this document was produced by

```
% cweave -j Turing.w
```

producing the LaTeX file *Turing.tex* which can be processed in the usual way

```
% latex Turing
% xdvi Turing
% dvips Turing
```

8 Administrivia

The file *Turing.w* from which this document is derived can be retrieved from `ftp://ftp.maths.tcd.ie/pub/TeX/javaTeX/javaweb/examples/Turing.w`.

For simplicity this work is published subject to the GNU GPL Licence. Essentially this allows the work to be freely copied and used, provided the original file *Turing.w* is made available. Changes should preferably be made through a change file — perhaps *Turing.ch*.

Index

List of Refinements

⟨ Buffer input and output 36 ⟩ Used in chunk 35.
⟨ Exit if *state* = 0 40 ⟩ Used in chunk 35.
⟨ Get *action* 22 ⟩ Used in chunk 18.
⟨ Get *bit* 21 ⟩ Used in chunk 18.
⟨ Get *inState* 20 ⟩ Used in chunk 18.
⟨ Get *outState* 23 ⟩ Used in chunk 18.
⟨ Move to new *state* 39 ⟩ Used in chunk 35.
⟨ Open rules file 43 ⟩ Used in chunk 41.
⟨ Parse line 19 ⟩ Used in chunk 18.
⟨ Print out rule being followed 38 ⟩ Used in chunk 35.
⟨ Process any options 42 ⟩ Used in chunk 41.
⟨ Read in rules 44 ⟩ Used in chunk 41.
⟨ Subsidiary classes 17, 27, 28 ⟩ Used in chunk 3.
⟨ Take appropriate *action* 37 ⟩ Used in chunk 35.
⟨ **Rule** constructors 18, 24, 25 ⟩ Used in chunk 17.
⟨ **Rule** methods 26 ⟩ Used in chunk 17.
⟨ **Tape** constructors 33 ⟩ Used in chunk 28.
⟨ **Tape** instance methods 30, 31 ⟩ Used in chunk 28.
⟨ *Turing* class data 5, 6 ⟩ Used in chunk 2.
⟨ *Turing* class methods 41, 45 ⟩ Used in chunk 2.
⟨ *Turing* constructors 14, 15, 16 ⟩ Used in chunk 2.
⟨ *Turing* instance data 8, 9, 10, 11, 12, 13 ⟩ Used in chunk 2.
⟨ *Turing* instance methods 34, 35 ⟩ Used in chunk 2.