Today's Lecture: Programs that Print Themselves

Reading:

Today: Something Fun: Programs that print themselves, viruses, Trojan horses, worms, etc.

I. A program that prints itself.

A. Figure 1 shows the source code for a Java program that prints itself. It's on the course web-page at
http://www.ugrad.cs.ubc.ca/~cs421/notes/12.02/PrintMyself.java

1. Kozen has another example (see *Kozen*, p. 288).

2. Kozen's program is short and clever.

3. I deliberately wrote mine to be as straightforward as I could make it – no tricks.

4. Both work for the same reason.

B. The main idea

1. Store the text for the program in the array `s`.

2. Access this array twice:

a. The first time to print out the declaration of the array itself.

b. The second time to print the rest of the program.

C. The method `myString` computes the string that is the source code for the program.

1. The first two `buf.append` invocations correspond to the first two lines of the program (declaring the class and declaring `s`).

2. The first `for`-loop prints the strings that initialize `s`.

a. This is the first use of the data in `s`.

b. The method `fix` takes care of the special sequences: `\"`, `\\` and `\n`.

c. The `if(i < s.length()-1)` statement handles whether or not there should be a comma after the string. Commas separate strings in the initializer. Every string except for the last one is followed by a comma.

3. Between the two `for`-loops, the `myString` inserts a line with `};` to end the declaration of `s` and a blank line.

a. I could have put these things into `s` as well.

b. I wrote it the way that I did so that `s` neatly matches the program structure.

c. The text in `s` corresponds exactly to the source for the methods `fix`, `myString` and `main`. I didn't break-up any functions or declarations.

4. At the end, `myString` appends the final `}` for the class.

D. The method `fix` takes care of special characters in the strings in `s`.

1. The characters `\`, `"` and newline are special in Java string constants.

2. They are represented in the source code with the sequences `\\`, `\"` and `\n`.

3. The method `fix` examines each character, `c`, of its argument.

a. If `c` is one of these special characters, `fix` replaces it with the corresponding escape sequence.

b. Otherwise, `fix` just copies the character directly into its return string.

c. Note that these escape sequences use the `\` character, which must be escaped itself. Writing this part of the method required carefully counting the backslashes.

E. The method `main` is very simple – it just gets the string for the source code of the program from the `myString` method and prints it.

```
class PrintMyself {
  static String[] s = new String[] {
    "  static String fix(String u) {                              \n",
    "    StringBuffer buf = new StringBuffer();                   \n",
    "    for(int i = 0; i < u.length(); i++) {                    \n",
    "      char c = u.charAt(i);                                  \n",
    "      if(c == '\"') buf.append(\"\\\\\\\"\");                   \n",
    "      else if(c == '\\\\') buf.append(\"\\\\\\\\\");             \n",
    "      else if(c == '\\n') buf.append(\"\\\\n\");                \n",
    "      else buf.append(c);                                    \n",
    "    }                                                        \n",
    "    return(buf.toString());                                  \n",
    "  }                                                          \n",
    "                                                             \n",
    "  public static String myString() {                          \n",
    "    StringBuffer buf = new StringBuffer();                   \n",
    "    buf.append(\"class PrintMyself {\\n\");                     \n",
    "    buf.append(\"  static String[] s = new String[] {\\n\");   \n",
    "    for(int i = 0; i < s.length; i++) {                      \n",
    "      buf.append(\"    \\\"\" + fix(s[i]) + \"\\\"\");            \n",
    "      if(i < s.length-1) buf.append(\",\");                    \n",
    "      buf.append(\"\\n\");                                     \n",
    "    }                                                        \n",
    "    buf.append(\"  };\\n\");                                   \n",
    "    buf.append(\"\\n\");                                       \n",
    "    for(int i = 0; i < s.length; i++)                        \n",
    "      buf.append(s[i]);                                      \n",
    "    buf.append(\"}\\n\");                                      \n",
    "    return(buf.toString());                                  \n",
    "  }                                                          \n",
    "                                                             \n",
    "  public static void main(String[] args) {                   \n",
    "    System.out.print(myString());                            \n",
    "  }                                                          \n"
  };

  static String fix(String u) {
    StringBuffer buf = new StringBuffer();
    for(int i = 0; i < u.length(); i++) {
      char c = u.charAt(i);
      if(c == '"') buf.append("\\\"");
      else if(c == '\\') buf.append("\\\\");
      else if(c == '\n') buf.append("\\n");
      else buf.append(c);
    }
    return(buf.toString());
  }

  public static String myString() {
    StringBuffer buf = new StringBuffer();
    buf.append("class PrintMyself {\n");
    buf.append("  static String[] s = new String[] {\n");
    for(int i = 0; i < s.length; i++) {
      buf.append("    \"" + fix(s[i]) + "\"");
      if(i < s.length-1) buf.append(",");
      buf.append("\n");
    }
    buf.append("  };\n");
    buf.append("\n");
    for(int i = 0; i < s.length; i++)
      buf.append(s[i]);
    buf.append("}\n");
    return(buf.toString());
  }

  public static void main(String[] args) {
    System.out.print(myString());
  }
}
```

Figure 1: A program that prints out its own source code

F. This approach to writing the program is quite general.

1. We could add other methods, fields, etc. to the class.

2. We can invoke `myString` as many times as we want. For example, if we had a method, `halt`, that supposedly solves the halting problem –

a. We could include the source code for `halt` in this class.

b. We would modify the strings that initialize `s` to include the `halt` function.

c. We could then write:

```
public static void main(String[] args) {
  if(halt(myString, ""))
    while(true);
  else System.exit(0);
}
```

This would give us the usual contradiction.

II. Gödel's Theorem, one more time

A. Just as we can make a mapping between programs and strings, we can make a mapping between statements in Peano arithmetic and the non-negative integers. In other words, for any formula that you can write in PA, there is an integer that corresponds to that formula.

1. For example, we could write formulas using the ASCII character set.

2. We can interpret each character as a digit in a number represented in base 256.

3. Consider the string "2+3".

a. It is a formula that when evaluated produces the integer 5.

b. Noting that the ASCII codes for '2', '+' and '3' are 50, 43, and 51 respectively, the formula itself corresponds to the integer 3287859 (i.e. $(256^2) * 50 + 256 * 43 + 51$).

B. We can write a formula, $\mathsf{proof}(c, p)$ that is true if the integer p is the encoding of a valid proof for the formula encoded by c.

C. PA includes quantifiers. For any claim, c, we can write the formula $\exists p.\ \mathsf{proof}(c, p)$ to say that there is a proof for c. This says that the claim encoded by c has a proof; in other words, c encodes a theorem of PA.

D. Just as we can write a program that prints itself, we can write a formula that evaluates to its own encoding.

1. Even better, recall that the method `myString` produced the string for the entire program, not just for itself.

2. In the same way, we can write a formula f that includes a sub-formula g, such that g evaluates to the encoding of f. Rather than writing g, I'll write $myCode$ for this formula.

E. We use this to write the formula γ with

$$\gamma \quad = \quad \neg\exists p.\ \mathsf{proof}(myCode, p)$$

1. This formula says "There is no proof for this formula".

2. The ability to write programs that print themselves and equivalently formulas that evaluate to their own encoding gives us a way to say "this" in a purely mathematical way.

3. Is γ a theorem?

a. If it is, then γ is false. If fact, we could easily prove $\neg\gamma$. Thus, γ and $\neg\gamma$ would both be theorems; in which case, PA would be unsound.

b. PA has been closely scrutinized by many people and appears to be sound.

c. Thus, it seems reasonable to conclude that γ is not a theorem.

d. You might argue that this is a weak argument, as it relies on our belief that PA is sound. However, **everything** that we supposedly know about mathematics relies on the soundness of PA or similar.

4. Is γ true?

a. As argued above, if PA is sound, then there can be no proof for γ.

b. This means that γ is true.

5. We conclude that γ is a true statement in PA for which there is no proof within PA.

F. While we assumed PA as the formal system for our proof, the same argument applies to **any** formal system that is powerful enough to be able to encode itself.

1. It takes very little for a formal system to be able to encode itself.

2. The minimalism of PA and TMs shows this.

3. Other "tiny" formal systems that have the same properties include the lambda calculus (invented by A. Church, the basis for programming languages such as lisp), Post correspondence systems (invented by E. Post), and others.

4. Any widely used programming language is Turing complete, and can express these kinds of problems. Any language that cannot is so limited that you'd complain loudly if you had to program in it.

III. Malware

A. Viruses and Worms

1. Viruses "infect" other programs.

a. A virus requires a mechanism for inserting a copy of its code into the host program.

b. The self-replication idea that we've looked at today is essential to this.

c. A virus gets access to the resources it needs to spread by the user running the program. Back in the MS-DOS days (including Windows 3.x, 95, and 98), this was easy because the application and the OS ran in the same address space. This meant that a virus in one application could copy itself into the "OS" part of the address space and linger there to infect other applications.
More accurately, DOS, Windows 3.x, 95, and 98 were not operating systems. They were application loaders, and rather clumsy ones at that. Badly written software is its own punishment.

2. A worm can propagate on its own.

a. Like a virus, a worm needs a way to create copies of itself, and the self-replication ideas we've looked at today are the enabling mechanism for worms.

b. Worms exploit security weaknesses in other software to propagate themselves. Thus, the other half of writing a worm is finding a security flaw in XP, Internet Explorer, Outlook, etc. Fortunately, these are so full of security holes that there has been no shortage of worm infections. Thus, a lecture like this won't lose its relevance any time in the near future.

B. Trojan Horses

1. A Trojan horse is a program that appears to do one thing (something good), but actually does something else instead of or in addition to, the advertised function.

2. A Trojan horse does not necessarily involve self-replication.

3. The paper today explored the possibility of putting a Trojan horse into a compiler.

a. A compiler is an example of where self-replication is a very desirable capability – you shouldn't take the description of malware above and decide that all software capable of self-replication should be banned.

b. As Thompson argued, the self-replication ability makes it possible to create malicious software that would be very difficult to detect.

i. Thompson described an application of compromising the security of the login program.

ii. How about making a version of Word that under special conditions recognizes the wording of a legal contract and makes subtle changes to the final version?

iii. How about making a version of a civil engineering CAD program that calculates slightly inadequate load capacities for some critical supporting structures in sky scrapers?

c. Now you know.

C. Rice's Theorem and the Java Sandbox

1. Can you write a perfect anti-virus program?

2. Rice's theorems say any non-trivial property of the language accepted by a TM is undecidable. Because useful programming languages are Turing complete, we can extend this to say that any non-trivial property of a program is undecidable.

3. The Java sandbox restricts the permissions of an applet so that it can't possibly do anything malicious (assuming that the JVM is correct).

a. Thus, all applets are safe.

b. The safety of a Java applet is a trivial property.

c. Rice's theorem allows us to decide it.

4. Once you let a program execute outside of the sandbox, there is no possibility of detecting *all* viruses, worms, and other malware.