Project phase 1 — Scanner front-end — assigned Tuesday 2 September, due Tuesday 16 September

1.1 Task

Write Java classes and interfaces to implement the scanner component of a PostScript interpreter.

1.2 Task in detail

Study the syntax of PostScript as defined in the PostScript Language Reference Manual, Second Edition (PLRM2) in Section 3.2.

The scanner (Section 3.2.1) conceptually has two components: a front-end, which consumes a list of characters and produces a stream of tokens, and a back-end, which consumes a list of tokens and assembles the tokens into PostScript objects. In this assignment, you will only write the front-end.[1]

Additionally, you must write a wrapper that consumes a string, opens a file the name of which is given by the string, reads in the characters of the file into a list and then invokes the scanner, returning a list of tokens as the final result. If there is any problem opening the file, a suitable exception should be raised.

Additionally, you must write a pretty-printer that accepts a list of tokens and prints the tokens one to a line. You should write the pretty-printer before you write the scanner itself, so that you can use the pretty-printer for debugging the scanner.

Features to pay attention to:

- correct handling of comments
- correct handling of white space
- correct recognition of numbers, including integers, reals, and radix numbers
- correct handling of integer numbers that exceed the implementation's limit on integer size
- correct handling of strings, including strings in parentheses and hexadecimally encoded strings
- correct handling of *all* string escape sequences (PLRM2, p. 29)

Deviations from PLRM2:

- You do not need to implement the standard PostScript behavior when errors in the input are discovered (such as PostScript limitcheck and syntaxerror errors. Instead, you should declare and raise corresponding Java exceptions. No specific recovery mechanism needs to be implemented. When these errors are raised, a suitable error message must be printed, and then the interpreter may quit.[2]

Features in PLRM2 that need not be implemented:

- binary token and binary object sequence encoding (see Section 3.2 and Section 3.12): only the ASCII encoding (Section 3.2.2) needs to be implemented
- ASCII base-85 strings

[1] For the back-end, we will have to implement a substantial part of the PostScript interpreter first; specifically, we need PostScript virtual memory (VM) in order to store components of composite objects in it.

[2] This simplified treatment of errors will apply in all future phases of the interpreter as well.

1.3 Hints on the Java implementation

You need to choose a Java representation for PostScript integers and reals. I recommend the types `int` and `double`.

You need to choose a Java representation for PostScript names. I recommend the type `String`.

You need to choose some way of representing lists of characters and lists of tokens.

You need to choose a Java representation for tokens. One possibility, which I recommend, is an abstract class of tokens with several concrete variants for integers, reals, names, delimiters, etc., as follows (here equipped with a visitor class interface):

```
abstract class TokenD
{
    abstract Object accept (TokenVisitorI ask);
}

interface TokenVisitorI
{
    Object forTokenInteger (Integer i);
    Object forTokenReal (Double d);
    Object forTokenString (String s);
    Object forTokenExecutableName (String n);
    Object forTokenLiteralName (String n);
    Object forTokenImmediatelyEvaluatedName (String n);
    Object forTokenLeftBracket ();
    Object forTokenRightBracket ();
    Object forTokenLeftBrace ();
    Object forTokenRightBrace ();
    Object forTokenLeftDoubleAngle ();
    Object forTokenRightDoubleAngle ();
}

class TokenExecutableName extends TokenD
{
    private final String n;
    TokenExecutableName (String _n)
    {
        n = _n;
    }
    //---------------------------
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenExecutableName (n);
    }
}

class TokenImmediatelyEvaluatedName extends TokenD
{
    private final String n;
    TokenImmediatelyEvaluatedName (String _n)
    {
        n = _n;
```

```
    }
    //---------------------------
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenImmediatelyEvaluatedName (n);
    }
}

class TokenInteger extends TokenD
{
    private final Integer i;
    TokenInteger (Integer _i)
    {
        i = _i;
    }
    //---------------------------
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenInteger (i);
    }
}

class TokenLeftBrace extends TokenD
{
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenLeftBrace ();
    }
}

class TokenLeftBracket extends TokenD
{
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenLeftBracket ();
    }
}

class TokenLeftDoubleAngle extends TokenD
{
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenLeftDoubleAngle ();
    }
}

class TokenLiteralName extends TokenD
{
    private final String n;
    TokenLiteralName (String _n)
    {
```

```
        n = _n;
    }
    //---------------------------
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenLiteralName (n);
    }
}

class TokenReal extends TokenD
{
    private final Double d;
    TokenReal (Double _d)
    {
        d = _d;
    }
    //---------------------------
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenReal (d);
    }
}

class TokenRightBrace extends TokenD
{
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenRightBrace ();
    }
}

class TokenRightBracket extends TokenD
{
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenRightBracket ();
    }
}

class TokenRightDoubleAngle extends TokenD
{
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenRightDoubleAngle ();
    }
}

class TokenString extends TokenD
{
    private final String s;
    TokenString (String _s)
```

```
    {
        s = _s;
    }
    //---------------------------
    Object accept (TokenVisitorI ask)
    {
        return ask.forTokenString (s);
    }
}

class TokenToStringV implements TokenVisitorI
{
    public Object forTokenInteger (Integer i)
    {
        return "Integer " + i;
    }
    public Object forTokenReal (Double d)
    {
        return "Real " + d;
    }
    public Object forTokenString (String s)
    {
        return "String " + s;
    }
    public Object forTokenExecutableName (String n)
    {
        return "ExecutableName " + n;
    }
    public Object forTokenLiteralName (String n)
    {
        return "LiteralName " + n;
    }
    public Object forTokenImmediatelyEvaluatedName (String n)
    {
        return "ImmediatelyEvaluatedName " + n;
    }
    public Object forTokenLeftBracket ()
    {
        return "[";
    }
    public Object forTokenRightBracket ()
    {
        return "]";
    }
    public Object forTokenLeftBrace ()
    {
        return "{";
    }
    public Object forTokenRightBrace ()
    {
        return "}";
```

```
    }
    public Object forTokenLeftDoubleAngle ()
    {
        return "<<";
    }
    public Object forTokenRightDoubleAngle ()
    {
        return ">>";
    }
}
```

How to turn in

Turn in your code by running

~barrick/handin your-file

on a regular UNM CS machine.

You should use whatever filename is appropriate in place of your-file. You can put multiple files on the command line, or even directories. Directories will have their entire contents handed in, so please be sure to clean out any cruft.