Mathematics 308—January 10, 1996

Remarks about running GhostScript

GhostScript, the PostScript interpreter we shall use in this course, is started by typing

```
gs
```

and when you do that you will get a sequence that looks like this:

```
Initializing...  done.
Ghostscript 2.6.1 (5/28/93)
Copyright (C) 1990-1993 Aladdin Enterprises, Menlo Park, CA.
  All rights reserved.
Ghostscript comes with NO WARRANTY: see the file COPYING for details.
GS>
```

and a window will pop up on the screen. The program is now waiting for you to type in PostScript commands. You can type them in one by one, and they will be interpreted one by one. Or you can read in a file with a command like

```
(test.ps) run
```

which causes `gs` to load your file and run it.

By the way, to avoid confusion later on, it is a very good idea to name all your PostScript files with `.ps` extensions.

If things go wrong in `gs` you will get a lot of garbage on the screen. **It is not easy to understand this garbage!** Usually you will have made a simple error in typing, in which case near the beginning of the garbage you will probably get a message about something being `undefined`. One thing to be careful about is distinguishing capital from small letters. Another common error is `stackunderflow` which means that you have forgotton to put something on the stack before a command. For example if you have the sequence

```
4 add
```

you will get stack underflow because `gs` is expecting two numbers to add, not one. Here is a sample:

```
Error:  /stackunderflow in exch
Operand stack:
    /s
Execution stack:
     %interp_exit  --nostringval--  --nostringval--  --nostringval--  %loop_continue  --
nostringval--  --nostringval--  false  --nostringval--  --nostringval--  --nostringval--
false  --nostringval--  --nostringval--  --nostringval--
Dictionary stack:
    511/547  0/20  15/200
Current file position is 138
GS<1>
```

Here `exch` was expecting two arguments on the stack but only got one. You will usually want to start over after error garbage on file input by typing `clear`, which empties the stack. You will probably want to run `gs` with both an `xedit` window and the `gs` window visible at the same time.

The last prompt illustrates something useful about `gs`—the prompt `GS<1>` tells you the depth of the stack. Sometimes this is very useful. For example we have this sequence:

```
GS>2
GS<1>3
GS<2>add
GS<1>=
5
GS>5
GS<1>dup =
5
GS<1>
```

which illustrates this and also shows you one of the simplest debugging tools you have: typing `=` will show you what is on top of the stack—but it will remove it as well. If you type `dup =` you will make a copy on top which you can throw away.

You can look at the whole stack without changing it by `stack`.

```
GS>1
GS<1>2
GS<2>3
GS<3>4
GS<4>stack
4
3
2
1
GS<4>
```

There is a variation on `=` which is `==`. It behaves a bit differently on complicated objects.

```
GS<4>{add sub mul}
GS<5>dup =
--nostringval--
GS<5>dup ==
{add sub mul}
GS<5>
```

In effect, running `gs` interactively you have a way to try out drawing on the screen interactively. You can save stuff you type in such a fashion, say to to a file, by selecting from the `xterm` window and then pressing the middle mouse in an `xedit` window. Of course this will give you a little extra junk to get rid of in the `xedit` window.

You exit from gs by typing `quit` or the keys `control-C`.