```
ZM~~ #        PRinty#     C with ABC!   .......................................................................................................................+
```

Dr. Tom Murphy VII, Ph.D.
tom7@tom7.org

Hello, and welcome to my paper! I’m really happy to have you here! <3

In this paper, I describe a new compiler for the C89 programming language.

For good reasons that I will explain later, this paper must be 20 pages long. Due to unreasonable SIGBOVIK deadlines, I did not produce enough technical material to fill the minimum number of pages, so I will am going to take my time and I have inserted several unrelated ASCII-art drawings.

** 1. Typesetting note **

If you receive this paper in a raw text file, it may be difficult to read because of its two-column layout. It should be typeset in a monospace font on pages 160 characters wide and 128 characters tall (this is 4x the typical density of a line printer from the 1980s or 1990s). Many pages, including parts of this first one, have cropping marks outside the text body to make the correct alignment easier to verify. This file contains no carriage returns or newlines; each line just contains 160 characters and is padded with spaces. If you receive this paper in the SIGBOVIK proceedings, it may be hard to read because it is printed in a very small font to conserve paper. Squinting really hard to read tiny hard fonts is good exercise for your eyes.

Your antivirus software may detect this paper as a virus, for good reasons that I will describe later. It is not a virus. ;-)

** 2. Introduction **

On any normal computer, a program is just a data file. It usually contains some header information that tells the operating system about what it is (for example, to confirm that it is a program and not some other kind of file; to tell the operating system about how much memory it needs, or the libraries it depends on, etc.) and then contains commands for the processor to execute. I’m not talking about stuff like shell scripts and Python programs, which contain text-based commands (like 10 PRINT "HI") interpreted by some other program. I mean real executable files. These commands are low level instructions called opcodes, and are usually just a few bytes each. Maybe just one byte. For example, on the popular and elegant X86 architecture, the single byte 0xF4 is the "HLT" instruction, which halts the computer. (Could this be why ALT-F4 is the universal key code for quitting the current program? Intriguing!) (Of course, some instructions like HLT are strictly off limits for "user space" programs. When running a program, the operating system puts the processor into a mode where such rude instructions instead alert the operating system to the program’s misbehavior. We’ll talk more about rude instructions in Section 17.) The single byte 0x40 means "INC AX" -- add 1 to the "AX" register -- and a multibyte sequence like 0x6A 0x40 means "PUSH 0x40". All the time, the computer is just reading the next byte out of the program (or operating system, itself a program written using these same instructions), doing what it says to do, and then going on to the next one.

I wrote the opcodes above in hexadecimal notation, but they’re just stored in the files and memory as raw bytes (like all files). The byte 0xF4 is not considered "printable" because old-timey computer people couldn’t agree on how it should look. In DOS, it’s the top half of an integral sign, like this:

```
. . . . . . . . .
. . . . . . . . .
. . . . @@@@@@. .
. . . @@@@. @@@@.
. . . @@@@. @@@@.
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
. . . @@@@. . . .
```

The first half of all bytes (0x00 to 0x7F) are defined in ASCII, which is standard across almost all computers now. When you look at the @ symbols in the picture above, they are almost certainly represented as the byte 0x40, which means the character @ in ASCII. And so if you peered directly at the bytes in this file, you would see a lot of 0x40s in that region. Sometimes the @ sign can be the flower of a rose, like --,--’-<@. To the processor, it means INC AX, since 0x40 is that opcode.

Now, for good reasons that I will explain later, this paper must contain 8,224 repetitions of the string "~~Q(", another weird flower. Please proceed to Page 3 to continue reading this interesting paper.

```
                                                                ~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(~~Q(
```

Sorry about that!

Not all of the ASCII bytes are considered printable, either. For example, 0x14 means DEVICE CONTROL 4 in ASCII, forever enshrined as that useless idea. Even DOS didn't think it was useful, so DOS prints it as a "paragraph" symbol. The byte 0x07 just makes a beep sound if you try to display it.

The range of actually printable characters are:

```
   0x0A   NEWLINE
   0x0D.  CARRIAGE RETURN
   0x20.  SPACE
   ...    (all the keyboard characters are from 0x20-0x7e)
   0x7E.  ~
```

.. and no others. 0x0A and 0x0D are actually pretty questionable, because UNIX, MacOS and DOS/Windows could not agree on whether a line ends with newline, carriage return, or carriage return and then newline. This paper is concerned with reliably printable characters, so we say that's the 95 characters from 0x20 to 0x7E, inclusive. This is all of 'em, with the upper-left corner being 0x20 SPACE.

```
      ! " # $ % & ' ( ) * + , - . / 0 1 2
    3 4 5 6 7 8 9 : ; < = > ? @ A B C D E
    F G H I J K L M N O P Q R S T U V W X
    Y Z [ \ ] ^ _ ` a b c d e f g h i j k
    l m n o p q r s t u v w x y z { | } ~
```

By the way, I tried to be disciplined in this paper about writing hexadecimal numbers in C notation, like 0x42 to stand for 66. The x86 architecture is little-endian, so a 16-bit word 0x1234 is stored in memory as 0x34 0x12. Also, when I write x86, that is not a hexadecimal number, that's the name of the computer architecture.

** 3. Printable x86 **

Since only 37% of bytes are printable, if you inspect (i.e., "cat") an executable program, it will almost always contain unprintable characters, and may beep at you, etc. However, since the printable bytes do stand for some subset of X86 opcodes, it is technically possible to make X86 sequences that are printable. One famous example is the EICAR Test File:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

This string is used to test antivirus software, because you can hide this string away inside some file and then see if the antivirus software can successfully find it (?). What's cool about this string is that if you stick it in a file called, say, EICAR.COM, you can just run that file in DOS and it prints out

```
EICAR-STANDARD-ANTIVIRUS-TEST-FILE!
```

The EICAR Test File is clever, but there are a few problems with it:

- It was written by hand. Though it's easy to change the message it prints, everything else about it is extremely delicate.

- Because it's in a COM file, it only has access to a single 64k segment, which must hold the code, data, and stack.

- Most damningly, like many viruses it uses "self-modifying code" to first rewrite itself into different opcodes. This means that the processor ends up executing several non-printable opcodes. This is like telling the waiter that you don't eat poultry but eggs are okay, and then they bring you an egg, but that egg hatches into a chicken right after they bring it to you. Come on.

In this paper I present a compiler for the C89 programming language* called ABC. It produces completely printable executables from C code. While self-modifying code is a powerful technique, it makes this problem "too easy;" I want to explore what programs can be written natively in the printable subset of X86. Programs compiled with ABC do not modify themselves, or cause themselves to be modified; every instruction program executes (outside of the operating system) contains only the bytes 0x20-0x7E. Moreover, every byte in the file is printable, so programs can viewed as text.

Source code for this project is available at: http://tom7.org/abc

* Not every C feature is implemented. Some of these are just not feasible and some I just didn't get to before the deadline. The shortcomings are discussed in Section 26.

** 4. Difficulties **

This is a challenging programming problem!

- Well, you have to write a compiler;

- Due to some constraints, it has to produce reasonably good (small) code, or the compilation strategy will fail;

- You only get a handful of instructions;

- Some extremely important instructions are completely missing;

- Notably, superficially you can't load arbitrary numbers into registers, jump backwards, or interact with the operating system;

- Many remaining instructions can only be used in weird addressing modes;

- Several standard techniques for assembling programs don't work due to the subset targeted;

- The program's header must also be printable, which puts constraints on its size and layout;

- Unreasonable SIGBOVIK policies require that papers not be xxx-tra large-size.

```
+-------------------------------------------------------------------+
| By now you've probably guessed from the gibberish you've been      |\
| seeing that this paper is itself the output of ABC; that is, this  | |
| this paper is also an executable file. If so, you guessed correct!! | |
+-------------------------------------------------------------------+ |
 \___________________________________________________________________\|
```

** 5. The CISC Ridiculous **

Let's look at the printable opcodes available in X86. Don't actually read this table, but I will refer to it:

20-23	AND reg|mod/rm
24,25	AND AL/AX/EAX <- imm
26	ES segment override prefix
27	DAA Decimal Adjust AL after addition
28-2B	SUB reg|mod/rm
2C,2D	SUB AL/AX/EAX <- imm
2E	CS segment override prefix
2F	DAS Decimal Adjust AL after subtraction
30-33	XOR reg|mod/rm
34,35	XOR AL/AX/EAX <- imm
36	SS segment override prefix
37	AAA ASCII Adjust After Addition
38-3B	CMP reg|mod/rm
3C,3D	CMP AL/AX/EAX <- imm
3E	DS segment override prefix
3F	AAS ASCII Adjust After Subtraction
40-47	INC multibyte register
48-4F	DEC multibyte register
50-57	PUSH multibyte register
58-5F	POP multibyte register
60	PUSHA Push all registers
61	POPA Pop all register
62	BOUND Check array index against bounds
63	ARPL Adjust RPL field of segment selector
64	FS segment override prefix
65	GS segment override prefix
66	operand size override prefix
67	address size override prefix
68,6A	PUSH imm
69	IMUL
6C,6D	INS ES:DI <- DX
6E,6F	OUTS DX <- DS:SI
70-7E	Jcc+disp8 variants

Figure 1. Instructions in printable x86

That's all we get! Many of these opcodes take arguments, such as an immediate byte (or word, or double-word); for example the sequence 0x24 0x42 means AND AL <- 0x42. In these cases, the arguments must of course also be printable, which limits what we can do with them, sometimes severely.

It's not clear that it will even be possible to do basic things, and it was a pretty satisfying hacking challenge to work around its limitations. If you have some x86 assembly experience, you might want to give a little thought to the following puzzles:

- How can we load an arbitrary number (e.g. an address constant) into a register? Note that the immediate value in something like "PUSH imm" must be printable.

- Without the MOV instruction, how do we do loads and stores?

- Without the INT instruction, how can we even exit the program?

- How do we implement bitwise OR with the given instructions?

- The Jcc (e.g. JNZ, JAE) instructions take only an absolute displacement. How do we do function (pointer) calls and returns?

- The displacement must be printable, which means it is always a positive number. How do we even do loops?

I will explain those problems and my solutions in later sections; I think they are each interesting. (If you are not going to read the whole paper, which is likely, I think "18. Loops" and "17. Exiting and initializing the program" are the most interesting/funny hacks.) Various parts of the compiler's design are intertwined with the many constraints, so there is no easy path through the whole idea. For now, let's warm up with the file format.

** 6. Executable file formats **

In order for the compiler's output to be executable, it needs to be in a file that the operating system recognizes as program. This means that the header of the program needs to be printable too. We can rule out several formats that cannot possibly have printable headers:

On Linux, executables are ELF files. The first byte of these files is always 0x7F "DELETE", which is not printable. Several other bytes in the header have to be zero.

On MacOS, executables are Mach-O files. These files always start with 0xFEEDFACE, an amusing example of unprintable bytes whose hexadecimal representation nonetheless spells out words. It also requires a field called MH_EXECUTE to be 0x02, among other problems.

On Windows, most executables are EXE Files. The modern version of this format is called Portable Executable (PE) and is used for 32- and 64-bit programs. It contains a required COFF subheader which always starts with 0x50450000 (the zero bytes not printable). For backward "compatibility", PE EXE files actually start with old-style EXE headers, which are actually programs that print something like

```
This program cannot be run in DOS mode.
```

and then exit. Windows recognizes a secret code that tells it to ignore that part and look at the *real* program.

... this eliminates the main executable formats for the modern x86 platforms. :(We saw that the EICAR program is a COM file, so clearly that is a possibility?

A DOS .COM file has no header. The entire program is just inserted into memory at the address 0x0100 and starts running. This level of simplicity is a dream for a SIGBOVIK Compiler Author, but it has a fatal flaw. In order to understand, we need to take a break and talk about segmentation!

** 7. Segmentation break! **

DOS is a 16-bit operating system, and a 16-bit number can only denote 65,536 ("64k") different values. To allow programs to address more than

64k of memory, Intel introduced "segments" into the 8086. These are a nightmare for programmers, and when I was a teenager I thought I could perhaps live my whole life without really understanding them. We're back! Roughly speaking, the instruction set allows you to supply 16-bit addresses (offsets), but the processor internally combines these with 16-bit base addresses (segments). The "real address" is (segment * 16 + offset). Some annoying facts:

- The segment registers are changed through different instructions than the regular registers. None are available in printable X86.

- However, we can make some instructions use a different segment register with one of the prefix bytes (e.g. 0x36 makes the next instruction use the SS (stack) segment instead of the default, which might be DS (data)).

- However, some other instructions like PUSH or OUTS can only use a specific segment.

- There are multiple different SEG:OFF pairs that reference the same real address.

- The segment values are not predictable in DOS, because they depend on where DOS happens to place your program.

We'll have to deal with segments for sure, but one consolation (?) is that since we can't change the values, the program will only access the 64k of data within the segments it starts out with.

There are 6 segments, CS (code), DS (data), SS (stack), and three "other" segments ES, FS, and GS.

** 8. Executable file formats, continued... **

In a DOS .COM file, CS, DS, ES, and SS are all initialized to the same value. This is easy to think about, but it causes a super bad problem for us: The machine stack is inside the same segment as our code. The machine stack is a region of memory that the PUSH and POP instructions use (among others); it starts at the end of this single segment and grows downward (towards lower addresses, where the program's instructions are). If the stack collides with the program, then it will mess up the instructions (which might be an effective way to make self-modifying code, but we don't want to cheat). Most COM programs stay out of the way of the stack by being much smaller than 64k. For good reasons that I will explain later, in this project, execution will need to span the entire code segment. It might be possible to avoid using the stack in our programs, but DOS interrupts (Section 17) are constantly happening as our program runs. These interrupts use the stack, and although they put the stack pointer back where it was and don't modify anything currently on the stack, the values that they PUSH and then POP are still present in memory, overwriting whatever was there. We don't have any way to turn these off, because the CLI instruction ("clear interrupts") is 0xFA, which is not printable. It seems COM files will not work for this project.

This leaves old-style 16-bit DOS EXE files, which do just barely work, and this is what ABC produces. EXE files afford much more flexibility, such as the ability to access up to 640kb (barring tricks) of memory. They also have many features that we do not need or want. An EXE header looks like this:

offset	field	ABC's value (little-endian)	ASCII
00	magic number	0x5A 0x4D	ZM
02	extrabytes	0x7E 0x7E	~~
04	pages in file	0x20 0x23	#
06	relocation entries	0x20 0x20	
08	paragraphs in header	0x20 0x20	
10	minimum memory	0x20 0x20	
12	maximum memory	0x20 0x20	
14	initial stack segment	0x50 0x52	PR
16	initial stack pointer	0x69 0x6e	in
18	checksum	0x74 0x79	ty
20	initial ins pointer	(program dependent)	
22	code segment displacement	0x20 0x20	
24	relocation table start	0x20 0x20	
26	overlay number	0x43 0x20	C

Normally, the header is followed by the relocation table (if any; see below) and then the program image. The program image is some blob of data that gets placed contiguously in memory, with the data segment set to its beginning and the code and stack segments set to wherever the header asks. A typical layout would look like this, with the solid box being the contents of the EXE file:

```
                 DS,ES    CS       SS
                 |        |        |      (additional memory)
  (not in mem)  v        v        v
  +-----------+----------------------+ - - - - - - - - - - - - +
  |hdr| reloc | program image        |                         :
  |   |       |                      |                         :
  +-----------+----------------------+ - - - - - - - - - - - - +
   NOTVIRUS.EXE .......
```

All of the values in the header are printable, which causes some difficulty. The problem stems from the fact that we must use values that are much larger than is reasonable for several fields; the smallest 16-bit printable number is 0x2020, which is 8224. Several fields are measured in 16-byte "paragraphs" or 512-byte "pages" (anticipating their use in printable executables!), so these values can quickly get out of hand. Naive values cause the program's effective memory requirements to be too large, and DOS does not load our program. Nonetheless, it is possible. The gory details of the solution are documented in exe.sml, but the crux of the solution involves the following tricks:

- Overflow the "pages in file" (a page is 512 bytes, so 0x2320 is 4MB; way beyond the 1MB limit) field to provide a smaller effective value. The file still needs to be pretty big.

- Specify a much larger than usual "pages in header" (0x2020 * 16 = 131kb). Since the header isn't loaded into memory, it doesn't count against the program's memory needs. A really big header also gives us space to store the paper. You're looking at part of the "header" right now.

- Give technically invalid values for some fields (extrabytes, checksum, overlay number); DOS doesn't actually seem to care about these. This helps us get a paper title that's almost readable.

The layout of a compiled program is roughly like this:

```
                                 DS,ES    CS       SS
                                 |        |        |      (additional memory)
  (not in memory)               v        v        v
  +-------------------------+----------------------+ - - - - - -
  |hdr|paper| reloc | paper  |   program image      |            ...
  |   |intro|       |        |   paper   paper  paper|
  +-------------------------+----------------------+ - - - - - -
   PAPER.EXE .......
```

This results in an file size of 409,600 bytes, which I believe is the smallest possible. At 160x128 characters per page, this is exactly 20 pages. Since we can't change the segment registers, the active part of our program is only the 64kb data, code and stack segments, and since the stack segment is somewhat unreliable (as described above), we only put stuff in the data and code segments. As a result, we need to be thoughtful about code size; this will be a challenge.

It's not necessary to understand this diagram since you are looking at a 1:1 scale model right now, i.e., the program itself. I'll point these sections out as we encounter them.

** 9. The Program Segment Prefix **

The Program Segment Prefix, or PSP, is 256-bytes at the beginning of the data segment. Depending on how you look at it, DOS either overwrites the first 256 bytes of our program image, or the program image is loaded right after it, but starts at address DS:0x0100 rather than DS:0x0000. In any case, we get this for free whether we want it or not, for both COM and EXE files. Since this is just part of DS, programs will be able to read and write the data there. The most useful thing we get is the command line that the program is invoked with from the DOS prompt.

** 10. Relocations **

You already saw the header structure (it's the title of the paper) and the relocation table (the full page of "~~Q("). For normal programs, the purpose of the relocation table is for DOS to patch the program so that it can know where it's located in memory; each time a program is loaded it might be placed in a different spot. When the program is loaded, DOS goes through all of the entries in the relocation table, and modifies the given location in the program by adding the base segment to the word at that location. Usually this location is part of an instruction sequence like "PUSH imm; POP DS", where imm is some value that we want to be relative to the program's base segment. We can't change segment values, so the relocation table is useless to us. In fact it's harmful, because we have to have 8,224 (0x2020) relocation table entries in order to have a printable header, and whatever offsets are in there will get corrupted when the program is loaded. We repeat the same location over and over, and choose a location that's right after the code segment in memory, a part of the image we don't need. I'll point out the spot that gets overwritten when we get there. The locations are given as segment:offset pairs, which is nice because we have multiple ways to reference a given location. We simply solve for some seg:off such that (seg * 16 + off = addr) and both seg and off are printable.

** 11. Addressing modes, temporaries, calling convention **

In any compiler, one must decide on various conventions for how variables are laid out in memory, how registers and temporaries are used, how arguments are passed to functions, and so on. There are lots of such decision in ABC; some are basically normal and some are particular to the weird problems we have to solve. Let's talk about some of the limitations of the instruction set that we have access to, because those inform the low-level design.

In Figure 1, there are several instructions that look like this:

```
    AND reg|mod/rm
```

These are each a family of instructions like

```
    AND AX <- BX                      AND [12345] <- DI
    AND BX <- [BP+SI+4]               AND [EBP+12345] <- EBP
```

where the source (on the right) and destination are given by some bits in the instruction's encoding. The instruction always acts between a register and a "mod/rm", with two adjacent opcodes determining whether this is of the form "AND reg <- mod/rm" or "AND mod/rm <- reg". The mod/rm can be one of many possible values; here is a table which you need not absorb:

r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
		Reg:	000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	*20	*28	*30	*38
[ECX]		001	01	?09	11	19	*21	*29	*31	*39
[EDX]		010	02	?0A	12	1A	*22	*2A	*32	*3A
[EBX]		011	03	0B	13	1B	*23	*2B	*33	*3B
[sib]		100	04	0C	14	1C	*24	*2C	*34	*3C
disp32		101	05	?0D	15	1D	*25	*2D	*35	*3D
[ESI]		110	06	0E	16	1E	*26	*2E	*36	*3E
[EDI]		111	07	0F	17	1F	*27	*2F	*37	*3F
[EAX+disp8]	01	000	*40	*48	*50	*58	*60	*68	*70	*78
[ECX+disp8]		001	*41	*49	*51	*59	*61	*69	*71	*79
[EDX+disp8]		010	*42	*4A	*52	*5A	*62	*6A	*72	*7A
[EBX+disp8]		011	*43	*4B	*53	*5B	*63	*6B	*73	*7B
[sib+disp8]		100	*44	*4C	*54	*5C	*64	*6C	*74	*7C
[EBP+disp8]		101	*45	*4D	*55	*5D	*65	*6D	*75	*7D
[ESI+disp8]		110	*46	*4E	*56	*5E	*66	*6E	*76	*7E
[EDI+disp8]		111	*47	*4F	*57	*5F	*67	*6F	*77	7F
[EAX+disp32]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX+disp32]		001	81	89	91	99	A1	A9	B1	B9
[EDX+disp32]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX+disp32]		011	83	8B	93	9B	A3	AB	B3	BB
[sib+disp32]		100	84	8C	94	9C	A4	AC	B4	BC
[EBP+disp32]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI+disp32]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI+disp32]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI		111	C7	CF	D7	DF	E7	EF	F7	FF

Figure 2. Addressing modes

The "scaled index byte" (sib) has another table with 224 entries, which we won't get into. There is also a similar, but crazier, table for 16 bit addresses and 8 bit operands. Note that only part of this table is printable (marked with *), which means we can only use a subset of addressing modes. Notably:

- We can't do any register-to-register operations, like "AND AX <- BX". Most compilers use these instructions frequently!

- As a result, exactly one of the source or destination operand is some location in memory.

- The simple addressing modes can only be paired with some registers. For example, AND DI <- [EDX] is allowed, but AND AX <- [EDX] is not. [ESI] means the memory in the location pointed to by the value in the ESI register.

This is even more annoying than x86 usually is. That said, the fact that we don't have register-to-register operations means that register allocation is far less important than usual. Instead, we operate on a set of temporaries, accessed using the [EBP]+disp8 addressing mode. EBP's default segment is SS, so these temporaries are stored in the same segment as the stack. In fact, since we initialized the stack pointer towards the middle of SS (it has to be printable; the maximum value would be 0x7e7e, but we use 0x6e69 to make the title more readable), we have the entire region from that to 0xFFFF to use for temporaries. Each function frame (see below) has its own set of temporaries.

To perform a basic subtraction operation, whereas a traditional compiler is likely to emit an instruction like

```
    0x29 0xC2    SUB AX <- DX          ;; AX = AX - DX
```

ABC emits a sequence like

```
    ??                     MOV AX <- [EBP+0x22]    ;; AX = tmp2
    0x67 0x29 0x45 0x20    SUB [EBP+0x20] <- AX    ;; tmp0 = tmp0 - AX
```

which is not so bad. (Note that we do not have a MOV instruction; this puzzle is solved below). We often need to do much more work than this to perform a basic operation, and optimization is meaningful (especially things that reduce code size).

The [EBP+disp8] addressing mode denotes the location in memory at the address in EBP, plus the given 8-bit value (above, 0x22). Note that to encode this mod/rm, we need to write the displacement byte in the opcode, so it must be printable. The EBP register will therefore actually always point 32 bytes before the first temporary, so that temporary 0 is accessed as [EBP+0x20].

With this idea in mind, here is a summary of ABC's low-level design:

- A C pointer is represented as a 16-bit address into the data segment.

- Anything addressable therefore needs to be stored in DS. This includes global variables, local variables and function arguments.

- Global variables are just allocated at compile time to some locations near the beginning of DS.

- A traditional C compiler uses the machine stack to store local variables, but since these need to be in DS, not SS, we maintain a separate stack of arguments and locals in DS, which starts after the global variables and grows towards larger addresses. This is called the locals stack. The register EBX points 32 bytes before the locals stack, so that we can use [EBX+disp8] to efficiently access locals.

- EBP always points 32 bytes before the "temp stack".

- Both stacks (and the machine stack) advance when we make a function call, so that the values of locals and temporaries persist across the function call. ABC only stores the return address on the machine stack.

- Aside from EBX, EBP, and ESP (the machine stack pointer), all other registers can be used for any purpose.

Next, we need to implement a number of low-level primitives that let our program do computation. Let's warm up with something very basic.

** 12. Putting a value in a register **

When programming X86 like a normal person, a very common task is to put an arbitrary number (for example, the address of a global, or a value that appears in the user's program) into a register, like

```
     0xB8 0x34 0x12      MOV AX <- 0x1234
```

We don't have this instruction available, since its opcode 0xB8 is not printable. Moreover, we need to be able to load arbitrary values, not just printable ones (but the value is part of the instruction encoding).

We do have some ability to load values. For example, we can encode

```
                         AND AX <- 0x2020
```

since 0x2020 is printable. This clears most of the bits in AX, and then

```
                         AND AX <- 0x4040
```

will always clear the remainder, since (0x40 & 0x20 = 0x00). With AX containing 0x0000, we could then repeat "INC AX" 1,234 times to reach the desired value. This totally sucks, but it works.

There are often more direct routes. We can XOR and SUB and AND with printable 8- or 16-bit immediate values in addition to INC and DEC. There is probably no "closed form" solution for the quickest route to a given value (the presence of both XOR and SUB makes this rather like a cryptographic function), but we can use computers to help.

We build a routine that generates a series of x86 instructions that load a 16-bit value into AX. In the general case, we do this by loading two 8-bit values and jamming them into AX using a gross trick. To load an arbitrary value into AL (the low byte of AX), ABC uses a table that it creates upon startup. This table is of size 256x256, and gives us the shortest (known) sequence for putting some desired byte DST in AL when AL is known to already contain some byte SRC. This table is populated via something like Dijkstra's "shortest path" algorithm. For starters, the diagonal (SRC = DST) can be initialized to the empty instruction list. We can then use INC and DEC to fill the rest of the table with very inefficient but correct sequences (still, when SRC is 5 and DST is 6, INC AX will remain the best approach!). Next, we maintain a queue of cells that should be searched (everything goes on the queue except the diagonal, which is already optimal). We repeatedly remove items from the queue and then explore what cells we can reach from that source byte. For example, if we pull out the cell (SRC=0x80, DST=0x01), we try applying XOR, SUB, and AND (with printable immediate values), etc. to the source value 0x80 to see what we get. One such result is that we can get AL=0x00 by doing AND AL <- 0x40. Consulting the cell for (SRC=0x00, DST=0x01), we see that it contains a sequence of length 1 (INC AX), so this gives us a new best solution by concatenating these two paths (AND AL <- 0x40, INC AX), which is much better than (DEC AX, DEC AX, ... 79 times). We iterate this procedure until paths stop improving.

This works well, with only an average of 2.54 bytes of instructions needed to transform a source byte into a destination one (across all possible src/dst pairs). No sequence is longer than 4 bytes. Since this table is big and programmatically computed when the compiler starts, I took some trouble to optimize it (the naive implementation took 13 seconds, which is a bit of an annoying wait every time you run the compiler!). There were a few tricks, but the most fruitful one was to functorize the code that encodes x86 instructions. This code normally works with vectors, and then the test above for the shortest instruction sequence would use Word8Vector.size to compute the best one. In the functorized version, the type of vector is an abstract argument. We instantiate a size-only version of encoding where the "vector of bytes" is actually just the count of bytes, and concatenation is just +. The MLton compiler is then excellent at optimizing this code to throw away the computations of the byte values (they are dead), and this code becomes plenty fast (~800 ms).

The table of instructions contains interesting structure, or at least pretty structure. Since it is 256x256, it can't fit in this paper 1:1, but I cropped to the prettiest part, the leftmost 160 columns. It appears as two full pages in the data segment (Pages 8 and 9) as some cool ASCII triangles. In this graphic, a space character means 0 instructions (this is only the diagonal of course, mainly visible on the first page); '.' means one instruction byte (just INC and DEC, near the diagonal); '-' is two instruction bytes (like XOR AL <- 0x2A); '%' is three; and '#" is four. This fractal pattern (like the Sierpinski triangle?) shows up all over the place in mathematics and computer science and Hyrule. For example it is reminiscent of the matrix of game configurations in k/n Power Hours [KNPH'14].

Once we can load an arbitrary byte into AL, we can fill all of AX with this trick. Suppose that our goal is to load AH=0x12 and AL=0x34. If we don't know anything about AX, we can zero it with two AND instructions. Then we can emit the instructions to load 0x12 starting from the known value 0x00. Then this sequence:

```
   instruction    AH  AL        stack           (ww, xx, yy, zz stand for
                  ww  0x12      xx yy zz ...          some arbitrary junk)
   PUSH AX
                  ww  0x12      0x12 ww xx yy zz ...
   PUSH 0x3040
                  ww  0x12      0x40 0x30 0x12 ww xx yy zz ...
   INC SP
                  ww  0x12      0x30 0x12 ww xx yy zz ...
   POP AX
                0x12  0x30      ww xx yy zz ...
   INC SP
                0x12  0x30      xx yy zz ...
```

Remember that x86 is little endian, so the low byte goes on the top of the stack. This trick places two words adjacent on the stack, but then misaligns the stack by doing a manual INC SP (and again at the end to clean up). The result is that AL gets moved into AH, and a known printable value of our choice (0x30 above) into AL. We can then use our table to transform that known value to any desired value into AL, completing the 16-bit value. This is reasonably brief and only touches the AX register, and we use it all the time in the generated code.

** 13. Moving between registers and memory **

Another useful kind of instruction is MOV AX <- [EBP+0x20], which moves the 16-bit word at the address in EBP (offset by 0x20) into AX. This is how we read and write temporaries; the "AX <- [EBP+0x20]" part is printable, but we don't have the MOV opcode available (0x89). Fortunately, the XOR instruction is "information-preserving," so it can be used like a MOV. Specifically, if we already have zero in the destination, then XOR *is* a MOV. In order to load from memory we use an instruction sequence like:

```
    ... various ...      set ax <- 0x0000        ;; using tricks above
  0x67 0x33 0x45 0x20    XOR AX <- [EBP+0x20]
```

To write to memory, we do:

```
  0x50                   PUSH AX                 ;; save value to write
    ... various ...      set ax <- 0x0000        ;; using tricks above
  0x67 0x21 0x45 0x20    AND [EBP+0x20] <- AX    ;; clears to zero
  0x58                   POP AX                  ;; restore value
  0x67 0x31 0x45 0x20    XOR [EBP+0x20] <- AX    ;; write it
```

This is almost... nice! But don't worry, it gets grosser.

** 14. Bitwise OR **

We don't have the OR instruction, but it can be computed with this trick.

```
    1  1  0  0    A
    1  0  1  0    B

    1  0  0  0    A AND B
    0  1  1  0    A XOR B
    0  0  0  0    (A AND B) AND (A XOR B)
    1  1  1  0    (A AND B)  OR (A XOR B)
    1  1  1  0    (A AND B)  +  (A XOR B)

    1  1  1  0    A  OR B
```

This is the table of all possible bit combinations that A and B could have; the OR operation is of course only dependent on the pair of bits at each position. First, observe (in your mind; it's not in the table) that A OR B is the same as A + B unless both bits are 1; only in that case do we need to do a carry. So we compute A AND B, and A XOR B; the OR of these two is the same as A OR B (it separates A OR B into the cases where both bits in the input were 1, and the case where exactly one was 1). Since the two expressions never have a 1 bit in the same position, we can compute their OR with +, giving us the desired result. Implementing plus is also a multi-step process, described next:

** 15. Keeping track of what's up with the accumulator **

The ABC backend (tactics.sml) generates X86 for some low-level primitives that operate on temporaries, like "Add tmp1 <- tmp2". (This is described in Section 21 when discussing the phases of the compiler.) Because it's expensive to load constants into registers, we go through some trouble to keep track of the machine state as we generate code. This allows us to make some opportunistic improvements. For example, the actual SML code implementing Add on 16-bit numbers looks like this:

```
  fun add_tmp16 acc dst_tmp src_tmp : acc =
    let
      val acc = acc ++ AX
    in
      imm_ax16 acc (Word16.fromInt 0xFFFF) //
      XOR (S16, A <- EBP_TEMPORARY src_tmp) ??
      forget_reg16 M.EAX //
      INC AX ??
      forget_reg16 M.EAX //
      SUB (S16, EBP_TEMPORARY dst_tmp <~ A) -- AX
    end
```

The approach is to XOR the source value with 0xFFFF and then increment it by 1; this negates the value in two's complement. We can then use the SUB operator, whose opcode is printable, to subtract that negated value, which is the same as adding it. The "accumulator" (variable acc) lets us manage the steps. Without getting into tedious details, "acc ++ AX" claims the register AX so that tactics know not to clobber it; we later return it with "-- AX". The imm_ax16 function loads the value 0xFFFF into AX; this tactic gets to inspect what's known about the machine state. For example, if we happen to have just assembled something that left AX containing 0x0000 (very common) then we can simply DEC AX to get 0xFFFF in one byte. imm_ax16 updates the accumulator to record that AX now contains 0xFFFF, as well as emitting whatever instructions it needs. The // combinator emits a raw instruction, and the ?? combinator allows us to learn or forget a fact about a register. Because some tricks require knowledge of e.g. AL but not AH, the accumulator actually keeps track of each byte of each register independently. It also understands that if you claim ESI, then SI cannot be used (SI is part of ESI), and so on. This is nice, and the semi-monadic syntax allows what looks like assembly code in ML. (Also note the questionable <- and <~ (hyphen vs. tilde) datatype constructors that distinguish the two directions of instruction, "reg <- mod/rm" vs. "mod/rm <~ reg".) The biggest risk of this approach is if you don't accurately record the state of registers (e.g. you forget to "forget_reg16" after modifying it), because this can lead to tactics making wrong assumptions but only in certain unlucky situations. Some of my worst bugs were from this; it would be cleaner if the accumulator actually simulated the instructions to update its own internal facts, rather than have the programmer make assertions.

Since the accumulator is purely functional, another cool thing we can do is try out multiple different strategies for assembling some block, and pick the best one. For example, when we decrease EBP right before returning from a function (to restore the caller's temporaries), we can either subtract a constant (number of bytes depends on the machine state) or DEC BP over and over (frequently faster).

** 16. Pointer loads and stores **

Another primitive we must implement is "Load16 dst_tmp <- addr_tmp"; the temporary addr_tmp contains a 16-bit address, and we load the value contained at that address (in DS) and store it in dst_tmp. This is used for pointer dereferencing in the source C program, for example.

It's basically the same as loading from a temporary; we just need to do something like

```
    set DI <- 0            ;; macro
    XOR DI <- [EBP+0x20]   ;; appropriate addr temporary offset
    set SI <- 0            ;; macro
    XOR SI <- [DI]         ;; read from the address into SI
    set [EBP+0x24] <- 0    ;; appropriate dst temporary offset
    XOR [EBP+0x24] <- SI   ;; store it
```

(Again, the syntax [DI] means use the contents of the DI register as a memory address, and load from there. DI's default segment is DS, which is where C pointers always point.) The only complication is that the pure-indirect mod/rm bytes like [DI] can only be paired with certain registers or else they are not printable (Figure 2).

The reason to bring this primitive up is that there's a delightful hack that's possible if the destination temporary and address temporary are the same slot. This situation rarely occurs naturalistically, since it would correspond to unusual C code like (int*)x = (int*)*x. However, it is very commonly the output of temporary coalescing (Section 22), since it is typical for the final use of an address to be a load from it. So, this is actually useful (saves about 5% code size), but the main reason to do it is awesomeness! Let's say the single temporary is at EBP+0x20.

```
    set DI <- 0            ;; macro
    XOR DI <- [EBP+0x20]   ;; load the address into DI.
    XOR DI <- [DI]         ;; DI = DI ^ *DI        (!?)
    XOR [EBP+0x20] <- DI   ;; tmp = address ^ address ^ value
```

The first two steps are reasonable, and put the address into DI. We want to end up with the value (whatever address points to) in the single temporary. Next we execute a crazy instruction, which XORs the address stored in DI with the value it points to. After this, DI contains addr ^ value, sort of like an encrypted version of the value. However, the temporary still contains the address (the "decryption key"), so if we XOR DI into it, we get address ^ address ^ value, which is 0 ^ value, which is just value! It's really nice how short the instruction sequence is, and it only uses a single register. The instruction XOR DI <- [DI] is so weird--it probably occurs in almost no programs, because it is extremely rare for an absolute address to have any relationship with the value it points to. So we get extra style points for finding a legitimate use for it.

Stores are the same idea. The trick actually applies there too, but isn't useful because it doesn't save us instructions, and because it is uncommon for the address and value to be the same temporary in a store operation (store is not really the opposite of load in this sense; both temporaries are read and neither is modified in "Store16 addr_tmp <- src_tmp").

** 17. Exiting and initializing the program **

We also want to be able to exit the program when we're done. This is normally done by making a "system call" to an operating system routine to tell it that we're done and the program can be unloaded. In DOS, you make system calls by triggering a processor interrupt with the INT instruction, which is a way of telling the operating system, "Check this out!!" We don't have access to this instruction, whose opcode is 0xCD. Alas! The INT instruction is a gateway to all sorts of useful functionality, like printing strings and reading from the keyboard, reading and writing files, changing video modes, and so on, so it's very sad to go without it. (The EICAR test virus uses self-modifying code to create two INT instructions; one is to print the string and the second is to exit.) In DOS, INT 0x21 is the most useful one; you set registers to some values to access dozens of different functions.

INT 0x21 is so common that it appears in the Program Segment Prefix that's always loaded at the beginning of the data segment. It's just sitting there amidst some zeroes:

```
     ...
   DS:0x004A    0x00 0x00         ADD [BX+SI] <- AL
   DS:0x004C    0x00 0x00         ADD [BX+SI] <- AL
   DS:0x004E    0x00 0x00         ADD [BX+SI] <- AL
   DS:0x0050    0xCD 0x21         INT 0x21
   DS:0x0051    0xCB              RETF
   DS:0x004E    0x00 0x00         ADD [BX+SI] <- AL
   DS:0x004E    0x00 0x00         ADD [BX+SI] <- AL
     ...
```

It even tantalizingly has RETF (far return from function call) immediately after it, like it was planted there by some puzzlemaker of years past, exactly for this kind of situation. (I don't actually know why it's there!) RETF pops both a return address and return segment, so if we could manage to put a return address on the stack (not hard) and the code segment (we don't know it, but we could probably use the relocation table to write it somewhere) beneath it, and then somehow transfer control to DS:0x0050, we'd have a fully general INT 0x21 to use! It would even help with the loop problem (next section) since it lets us return to an arbitrary address, and could conceivably even let us escape the confines of always executing code within the initial code segment CS (because RETF modifies CS). But speaking of confines, none of this will work, because we have no way of modifying CS to start executing code out of DS. Too bad, so sad. (This idea might pan out for a COM file where CS=DS, but there we have no relocation table so figuring out what segment value to put in the stack would require some other hack. We also have the Loop problem, preventing us from reliably jumping to DS:0x0050. Might be worth further exploration.)

Jumping the program to a non-printable instruction is also a bit questionable, though it's not an instruction that we wrote there, so this does not violate our self-modifying code fatwa. Is it wrong for a waiter to serve the ovo lacto vegetarian with vegetarian food that causes him to eat non-vegetarian food that the customer himself brought with him? Who can say?

This is not hopeless. The way interrupts actually work is to stop the current execution (saving the state of the registers on the stack) and then consult a table of "interrupt vectors" (in my opinion the table itself should be called the "interrupt vector", containing addresses) at the address 0x0000:0x0000 (i.e., right at the beginning of memory). Each interrupt has a number, and each address is a 32-bit segment:offset pair. So the address at 4 * 0x21 = 0x0084 is the location of DOS's code for INT 0x21. In 16-bit real mode programs, there's nothing special about the operating system; you can just jump directly into it if you want, or overwrite it with your own stuff. In fact, this is how many viruses work; for example by replacing the address for INT 0x21 with their own code, and intercepting file operations to insert viruses before calling through to the original INT 0x21 handler so that everything still works.

Fetching the INT 0x21 address is not immediately useful, because we can't transfer control to it; we don't have the CALL instruction. In fact, the only JMP instructions we have must jump a small fixed distance forward (next section). But! The INT instruction is not the only way to trigger interrupts. The timer interrupt is firing continuously, messing with our stack, for example. We can modify the interrupt vector table to make the timer interrupt (INT 0x8) instead point to the INT 0x21 code, and then "wait" for a timer interrupt to happen, and maybe restore the old timer interrupt code when we're done. This might work, but it seems extremely brittle. (Also, the timer interrupt handler has to perform certain low-level duties or else the system will freeze.) Fortunately there's a better choice: The CPU will also trigger an interrupt when an illegal instruction is executed. Normally the illegal instruction handler would do something like crash the program gracelessly (in Unix, it sends the SIGILL signal. Sadly there is no SIGBOVIK.) Do we have an illegal instruction inside printable x86? In fact we do!

```
   0x63    Adjust RPL Field of Segment Selector
```

... it's just sitting in there, this totally weird instruction with no other possible uses amidst a bunch of sensible ones. This instruction is for some operating system privilege stuff, and is illegal in real mode.

So, when we first start up an ABC program, one of the first things we do is read the address of the INT 0x21 handler at 0x0000:0x0084, and write it over the INT 0x06 (illegal instruction) handler. Luckily the FS segment is set to 0x0000 when our program starts (we can't change it), so we can use the FS segment override instruction to access the beginning of RAM. Once we overwrite the address, then whenever we want we can set up argument registers for the system call "exit" (AH = 0x4c, AL = status code), and execute the illegal ARPL instruction. This will trigger interrupt 0x06, which is now actually the INT 0x21 code, and DOS will "cleanly" exit the program for us.

It is very tempting to use this trick to make other system calls through INT 0x21, or perhaps to jump to arbitrary addresses of our choosing! Sadly, there are two very serious issues:

- When the processor triggers the illegal instruction interrupt, the return address that it pushes on the stack is the address of the illegal instruction itself, not the one that follows it. So when the interrupt handler returns, it simply executes another illegal instruction.

- When the interrupt is triggered, it clears the interrupt flag (so that for example the timer interrupt doesn't fire while it's already running). Only a few instructions, which we don't have access to, can restore the interrupt flag. This means that we would only be able to do this once, and after we did, many things would stop working because interrupts would stop firing.

Neither of these issues are a problem for the exit system call, since we only exit once. YOEO!

The ARPL instruction takes two argument bytes which just have to be printable; the instruction we actually encode is

```
  0x63 0x79 0x61     ARPL [ECX+0x61] <- DI
```

The ASCII sequence is "cya", as in see ya, which we follow with an unexecuted exclamation mark for emphasis. You can find the string "cya!" in the code segment on page 16 if you’re good at Where’s Waldo stuff!

** 18. Loops **

The last major problem involves control flow. In printable x86 we have available a family of instructions Jcc+disp8. Jcc stands for "jump (on) condition code", and consists of 15 opcodes:

char	opcode			Also known as
p	0x70	JO	Jump if overflow	
q	0x71	JNO	Jump not overflow	
r	0x72	JB	Jump below	JNAE, JC
s	0x73	JNB	Jump not below	JNB, JAE, NKC
t	0x74	JZ	Jump zero	JE
u	0x75	JNZ	Jump not zero	JNE
v	0x76	JBE	Jump below or equal	JNA
w	0x77	JNBE	Jump not below or equal	JA
x	0x78	JS	Jump if sign	
y	0x79	JNS	Jump not sign	
z	0x7A	JP	Jump if parity even	JPE
{	0x7B	JNP	Jump if parity odd	JPO
|	0x7C	JL	Jump less	JNGE
}	0x7D	JNL	Jump not less	JNL
~	0x7E	JLE	Jump if less or equal	JNG

This is a fairly full set of conditions (although we are missing the last one, JNLE/JG, with opcode 0x7F). Each of these consults the processor’s FLAGS register and tests for a certain condition. FLAGS is updated on many operations; for example, the "zero flag" ZF is set to 1 if the result of certain operations is zero, such as if "SUB [EBP+0x24] <- AX" ends up writing 0x0000 into memory, and ZF is cleared to 0 if not. The JZ instruction jumps if ZF is set, and just continues on to the next instruction otherwise. JZ has an alias, JE (Jump equal); they are the same exact opcode because when you subtract two equal numbers, you get zero. Since it is common to want to set the appropriate FLAGS without actually subtracting, the CMP (compare) instruction is like SUB but it only updates flags. We have a version of the CMP instruction in printable x86, so all is well so far.

These particular instructions are Jcc+disp8, so we provide an 8-bit displacement. The address of the current instruction is stored in the EIP ("instruction pointer") register. When EIP points at Jcc+disp8 instruction, EIP is set to the instruction immediately after it (EIP+2) and then if we jump, incremented further by disp8. The disp8 byte is treated as signed, so jumps can go upward or downward. Unfortunately, all printable displacements are positive! This allows us to conditionally skip code, but only downward, and only between 32 and 127 bytes.

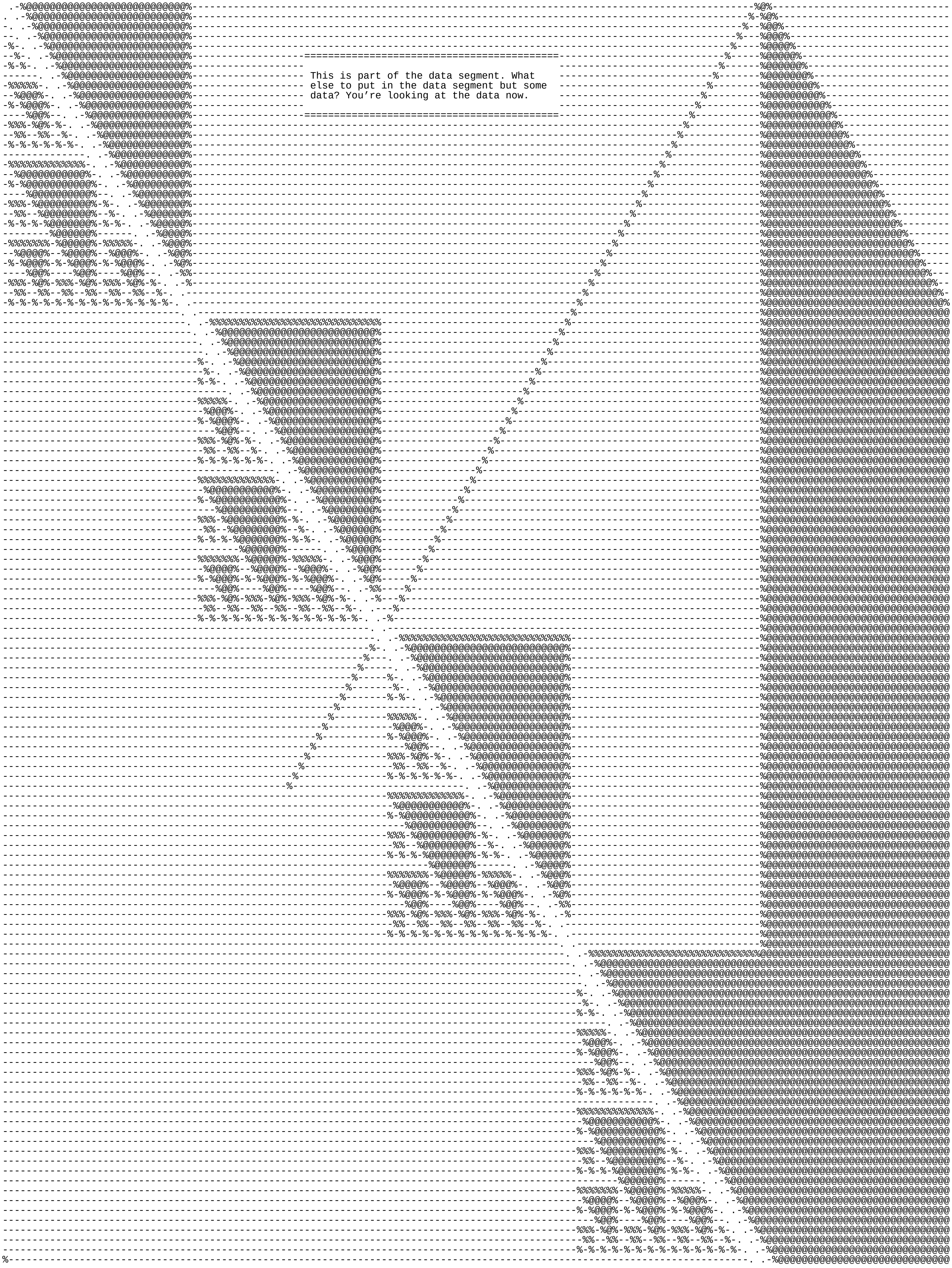
This subset won’t even be Turing-complete if we can’t jump backwards; all programs will terminate because the instruction pointer only increases. What actually happens when we reach the end of the code segment? If EIP is 0xFFFF and we execute a single-byte instruction like INC AX, EIP just continues on to 0x00010000; the EIP register is 32-bit despite us struggling with 16-bit segments and offsets. This instruction is right after the code segment, and indeed contains whatever followed our code segment in the program image. So we could conceivably break free of the 64k code segment. Unfortunately, performing a jump when in this weird state still just jumps downward, and the situation is very brittle (see Section 31 for some ideas and problems). However, there is a special case on the processor, probably for compatibility with an earlier processor; it’s right there in the pseudocode for this instruction in Intel’s manual [INTC’01]:

```
IF condition
   THEN
      EIP <- EIP + SignExtend(DEST)
      IF OperandSize = 16
         THEN
            EIP <- EIP AND 0000FFFFH;
      FI;                                          (sic -tom7)
         ELSE (* OperandSize = 32 *)
            IF EIP < CS.Base OR EIP > CS.Limit
                #GP
      FI;
FI;
```

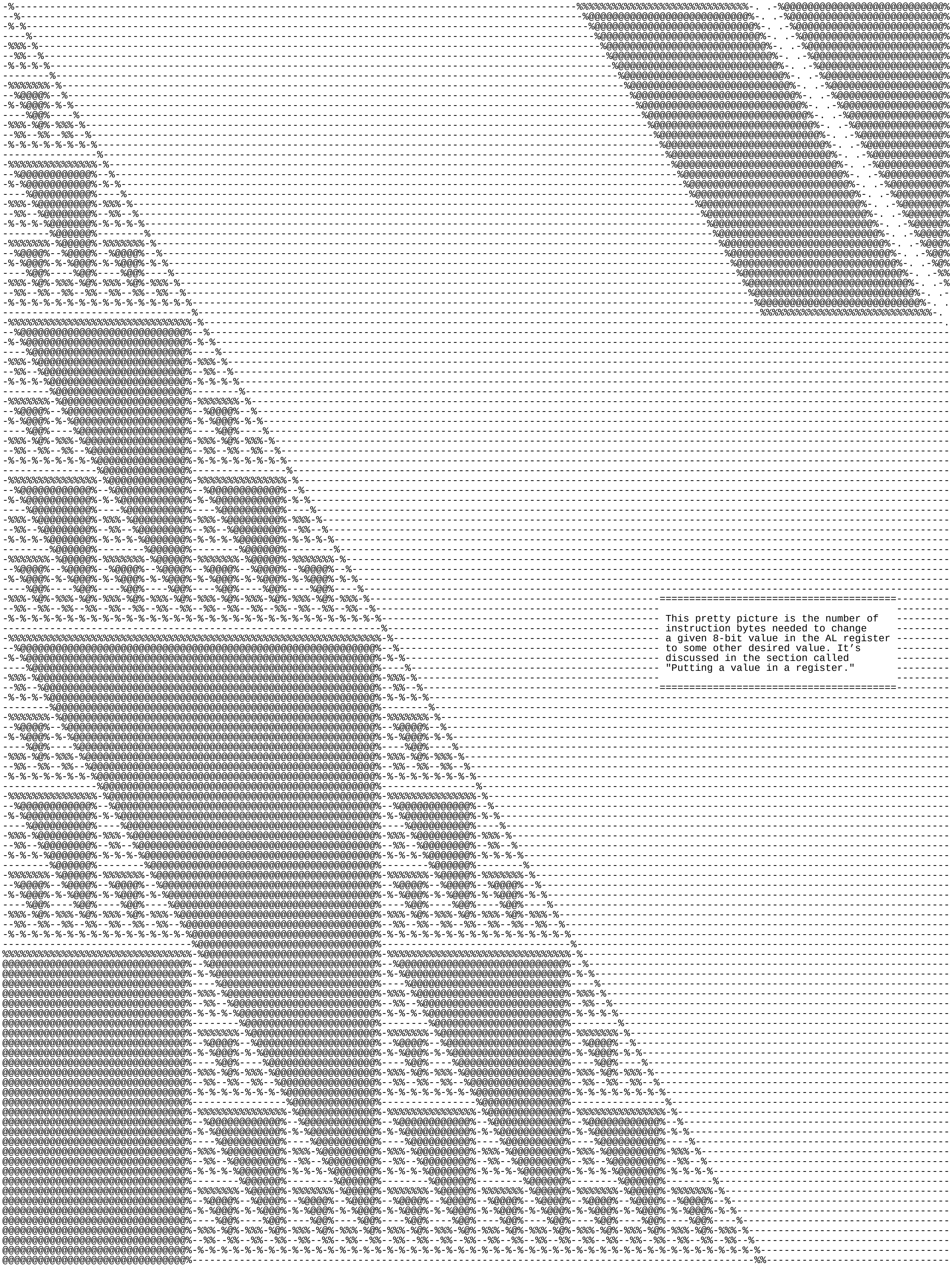
Specifically, if we are right at the end of the code segment, and our jump’s displacement takes us past the end, then we "wrap around" to the beginning, because EIP is bitwise-anded with 0xFFFF. This means that our program can do one backwards jump, from the end of the segment back to the beginning.

We’re approaching the data section now, so it’s time to take another break! Here it is:

```
                                                                                                                                         [Now you’re looking at the PSP.
The address of the opening square bracket is DS:0000, but this gets overwritten by DOS on load.  (Right here is where the command line is placed by DOS, up to
127 bytes. Before the open paren is its length in a byte.).....]____9;02457-^A8z2F4^G8F3c4^A4F4^A4^G8z8^A8z2c8z2^c8z2^d8z2f8z2F4F4F4F8-e’e’ze’zc’e’zg’z3g2z2c’z2
gz2ez2azbz^aazge’zg’a’2f’g’ze’zc’d’bz2c’z2gz2ez2azbz^aazge’zg’a’zf’g’ze’zc’d’bz4g’^f’f’^d’ze’z^gac’zac’d’z2g’^f’f’^d’ze’zc’’zc’’c’’z5g’^f’f’^d’ze’z^gac’zac’d’z2
^d’z2d’z2c’|DDzDzDDzgz3G2z2Gz2Ez2Cz2FzGz^FFzEczef2dezczABGz2Gz2Ez2Cz2FzGz^FFzEczefzdezczABGz2Cz2Gz2czFz2cczFzCz2Ez2Gczg’zg’g’zGzCz2Gz2czFz2cczFzCz^Gz2^Az2cz2GGz
C-C4C4G4G4A4A4G8F4F4E4E4D4D4C8G4G4F4F4E4E4D8G4G4F4F4E4E4D8C4C4G4G4A4A4G8F4F4E4E4D4D4C8-abd’b^f’3^f’3e’6ab^c’ae’3e’3d’^c’b4ab^c’ad’4e’2^c’2b2a2z2a2e’4d’4z4abd’b^
f’3^f’3e’6ab^c’aa’4^c’2d’2^c’b3ab^c’ad’4e’2^c’3ba2z2a2e’2d’2d’4|A4E4E4A4A4^F4^F4B4B4E4E4A4A4^F4B4A4A4E4E4A4A4^F4^F4B4B4E4E4A4A4^F4B4A2A2A2A2|A,2^F,2E,6A,4E2^F2E
2^F,6B,2A,2B,2A,2^F,2E,2^F,2G,2A,4E2^F2A2^F,6A,4E2^F2A2E,2E,2^F,2A,4E2^F2E2^F,6B,4^F2A2^F2E,2^F,2G,2A,4E2^F2E2^F,6A,4E2^F2A2------------?Qf{------8Y}---#R----8Y
}---#R----8Y}---#R----8Y}---#R----8Y}---#R----8Y}---#R----8Y}---#R----8Y}---#R----------------        !!!!!!!!!!!!"""""""#####&&&&&&&’’’’’*******+++++......///
//22222223333366666677777::::::;;;;;>>>>>>>????????????????--Now this is the part of the data segment that stores global variables. This is actually a string
constant in the program itself, so you’ll see it again when I show you the source code later. We have almost 64kb of space to store stuff, although this segment
 is also used for the stack of local variables and arguments, and would be used for malloc as well, if it were implemented. Storing a string like this is basica
lly free, because everything in it is printable, aside from the terminating \0 character. At program startup, non-printable characters are overwritten by instru
ctions in the code segment. Like, here’s one: --> - <-- It’s stored in the data segment as a printable placeholder.--bluehair--plumber--alphabet-???????????????
?_______________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
________________________________________________________________________________________________________________________________________________________________
```



This is part of the data segment. What else to put in the data segment but some data? You’re looking at the data now.



This pretty picture is the number of instruction bytes needed to change a given 8-bit value in the AL register to some other desired value. It’s discussed in the section called "Putting a value in a register."

Anyway, one backwards jump is enough! We can set things up so that whenever we need to jump backwards, we instead jump forward until we're at the end of the segment, then jump across that boundary (overflowing back to the beginning) and then keep jumping forward until we get where we need to be. This is delicate, but it works.

One other issue with jumps is that we can only jump a fixed distance; there is no equivalent to "MOV EIP <- AX" to jump to a computed location. We need this functionality to implement two C features: Function pointers (the destination of a function call is not known at compile time) and returning from functions (the function can be called from multiple sites, so we need to know which site to return to).

** 19. The ladder **

To solve the various problems with jumps, we build the program around what's called a "ladder" in the code. The whole program is broken up into small blocks of code. Each one is given a sequential "number" (this has nothing to do with the memory location, just its sequence in the list of blocks). Each block starts with a "rung," which is the following code

```
    DEC SI
    JNZ +disp8
```

where disp8 is a printable displacement that brings us downward to the next block. We decrement the SI register to count down to the block we want, and if it is Not Zero yet, then we jump to the next one. If zero, we execute the block. Inside a block, if we ever want to perform a jump to some arbitrary block dest_block, then we can compute:

```
    offset = (dest_block - current_block) mod num_blocks
    si = (if offset = 0 then num_blocks else offset)
    jmp to next rung
```

Every block knows its current number, so the offset is just a constant. Note that the destination block's number may be before the current block, which is why we need to mod by the total number of blocks (yielding a non-negative result). SI cannot be zero, because the first thing we do is DEC it, so a self-loop requires setting to num_blocks, a full cycle.

To perform a jump to a code location not known at compile time (e.g. from a return address (block number) on the stack, we can just perform the same computation as above. We do not have an efficient mod operation (implementing it seems to need loops, in fact, a circularity!), so instead we actually compute (dest_block - current_block) + num_blocks. This is always positive as needed, but requires forward jumps to make an entire cycle around the entire ladder ("Turn the dial to the left, passing zero and the first number...").

The blocks are laid out sequentially in the program until we get too close to the end of the segment; when we do, we make sure to perform an unconditional jump across the segment boundary, wrapping around. This jump need not DEC SI. In fact, most programs do not fill the entire code segment, so we end up padding the end and beginning of the segment with jumps to span the unused space. For these padding jumps, we definitely don't want to DEC SI, both because that's more instructions to execute, and because we don't know the amount of padding ahead of time (see the section on Assembling below).

There are many annoyances! A jump cannot be too short (less than 32 bytes) or too long (127 bytes). The viable range is large enough to build nontrivial programs, but is a significant constraint for us.

We don't have access to a non-conditional JMP instruction. There are a few tricks for simulating it. When computing a jump to a known label, we can just know the state of flags because we've just performed some computation. Even when doing a jump to a computed block number, we know that the result of subtraction is not zero, so we can always use the JNZ instruction. Occasionally we need to do a jump without knowing our state at all. XOR always clears the Overflow flag, so something like

```
    XOR AX <- [DI]
    XOR AX <- [DI]
    JNO disp
```

keeps AX unperturbed and always performs the jump. A little shorter is

```
    JNO disp
    JO (disp - 2)
```

which jumps to the same target whether the Overflow flag is set or not, but is more annoying because we need to keep track of two displacements.

** 20. Assembling **

Assembling the program is the process of generating actual instruction bytes (here, printable x86) from some semi-abstract representation of instructions (in ABC, this is the LLVMNOP language discussed in the next section). Assembling has a self-dependency: In order to generate instructions like jumps and loads of addresses, the assembler needs to know where code is located. But in order to know where code is located, the assembler needs to generate it. In most assembler tasks, this is reasonably straightforward: When we need to generate an instruction like "MOV AX <- offset data", we just emit "MOV AX <- 0x0000" and save for later an obligation to overwrite the zeroes with the address of "data", once we know where we placed it. This works because the encoding of the MOV instruction is the same length no matter what 16-bit value we load. The same holds for JMP instructions (with the caveat that smart assemblers can JMP+disp8 for nearby labels and JMP+disp16 for further ones; these instructions have different lengths) and others.

For the ABC compiler this step is quite bad:

- Loading any immediate value has a length ranging from 0 bytes (it's already in the register) to like 16. It's dependent on both the value being loaded and the context (contents of registers).

- The rungs that start each code block must be able to Jcc+disp8 all the way to the next block. This jump distance can't be too big, or else it can't be encoded (or is not printable).

- Jumps within a block always target the next block, but the jump distance can't be too short (or the displacement byte is not printable).

- Since blocks are numbered sequentially and relative addresses are computed modulo the total number of blocks, logical code addresses depend on the number of blocks and their order.

As a result, assembling is an iterative process. We take the program's blocks and translate them into position-independent machine code. One positive thing about the printable, non-self-modifying subset of x86 is that none of the instructions actually depend on what address they're placed at (except perhaps a Jcc instruction used to overflow the instruction pointer). Still, we don't know even the relative location of the next block yet, so we also record the offset of the displacement byte for any Jcc instruction we emit.

Next, we take these blocks and attempt to allocate them into the code segment. This can fail for the reasons above, usually after we've placed a block far enough from the preceding one that all jumps in the first are printable (at least 0x20 bytes), the rung at the beginning of that block can't target the second (because it is more than 0x7e+0x03 bytes away). We gather all such problem blocks and bisect the LLVMNOP code into two smaller blocks. Then we try again. When we succeed, we can fill in the displacement bytes for the Jcc instructions to create valid printable code. There are various opportunities to be smarter about this (for example, bisecting the LLVMNOP assumes that all such instructions assemble to the same length, which is not remotely true); tox86.sml contains several ideas.

Since the initial instruction pointer must be printable, we start laying out blocks towards the middle of the code segment. If a block would run off the end of CS, then we need to pad that region with jumps that get up close to the end of the segment and then do an overflowing jump past CS:0xFFFF before continuing layout. Once we run out of blocks, we also need to pad any remaining code space with jumps in order to bring control back to the first rung, since the ladder needs to be a complete cycle in order to work. It's easy to pick out the texture of this padding in the code segment (e.g. pages 14, 16).

** 21. LLVMNOP **

Knowing our low-level endpoint, I can now work backwards through the compiler. The compiler generally proceeds by a series of intermediate languages, the last of which is called LLVMNOP.

This language is an assembly-like language that has explicit *data* layout, but not not explicit *code* layout. By that, I mean that every function knows the size and offset of its locals and arguments in the current local frame, and the size and address of each global variable is known, as well as the global's initial values (if printable). It is akin to LLVM [LLVM'04], but doesn't really have anything to do with it. LLVM is an excellent tool for writing compilers (superficially, it looks like a good way to write a new C compiler targeting an architecture like printable x86!) but isn't really suitable for this project because it assumes that the output architecture has certain standard operations efficiently available, which is frequently not the case for printable x86.

A sample of LLVMNOP constructs are:

```
 cmd  ::= Add tmp <- tmp
        | Xor tmp <- tmp
        | Push tmp
        | Pop tmp
        | Mov tmp <- tmp
        | Immediate16 tmp <- word16
        | Load16 tmp <- tmp
        | Store16 tmp <- tmp
        | Load8 tmp <- tmp
        | Store8 tmp <- tmp
        | ExpandFrame i
        | PopJumpInd
        | JumpCond cond, label
        | ...
        | Out8
        | Init
        | Exit

 cond ::=  Below tmp, tmp
        |  BelowEq tmp, tmp
        |  ...
        |  EqZero tmp
        |  True
```

LLVMNOP exists in both a "named" and "explicit" version. In the named version, temporaries (tmp) are strings paired with a size (16 or 32 bits). In the explicit version, temporaries are given as a size and offset from the current temporary frame (EBP). The named version is transformed to the explicit version by the process called Allocation (below).

Commands are basically assembly instructions that we might have in a more expressive architecture; note for example that we have Add, which is not native in printable x86 (we implement it by computing the two's complement negation, and then subtracting). Even commands that have a corresponding printable x86 instruction like XOR are still compiled into multiple opcodes, since they read and write arguments to temporaries, not registers. We discussed the implementation of operations like Load16, Immediate16, and Mov in a previous section.

A program consists of a series of labeled blocks. JumpCond pairs a condition (signed and unsigned comparisons, etc.) with a jump to a label. The possible conditions map to the Jcc instructions that we have available. Since opcode 0x7F (Jump Greater) is not actually printable, all of the conditions "face less;" the condition Greater(A, B) is equivalent to Less(B, A). An earlier phase does this rewrite. Also note that in C, a < b is an expression that can be used in any context, not just for control flow; here the comparison is inextricably linked to a jump, since CMP only sets FLAGS, and FLAGS can only be used for jumping. An earlier phase removes the expression forms as well, without being too wasteful when the programmer writes "if (x < 1)" to begin with.

The only way to jump to a non-constant destination is with PopJumpInd, which is basically the RET assembly instruction. It pops an address (block number) from the top of the machine stack, and unconditionally transfers control to that label (by computing the number of blocks to traverse, then jumping to the ladder). This is indeed used to return from a function call, as well as to call a function through a function pointer. It takes its argument on the stack (as opposed to using the existing "Pop tmp" and then "JumpInd tmp") because while we're setting up a function call, we need to move the temporary frame pointer, after which point it is unsafe to access temporaries. The stack, however, is a stable place to stash data.

Since we have some higher-level operations like Mov available, we can implement some delicate maneuvers like function calls as sequences of multiple commands. On the other hand, for some primitives like Init and Exit, there's no real value in breaking them into smaller pieces. Some other complex primitives like Out8 have no analogous feature in C; these are provided as sort of "intrinsics" that can be used to do low-level programming in C. We'll discuss Out8 in Section 27 when we talk about IO. Other primitives, such as one called "Argv" that is used to initialize the argv parameter to main during initialization, is compiled away when we convert to LLVMNOP. In this case, the Argv primitive just creates a global array containing two elements: The second is zero ("null") as required by the standard, and the first is the constant address 0x0081, which is a pointer into the Program Segment Prefix where DOS stores the command line (untokenized; the programmer must do any processing she desires).

** 22. Temporary allocation **

Temporary allocation is fairly standard. We use a dataflow-based liveness calculation to determine which temporaries interfere with one another; if two temporaries of the same size don't interfere, then they can use the same slot, so they are coalesced into one. We prioritize coalescing temporaries in a "Mov tmp1 <- tmp2" so that we get the no-op instruction "Mov tmp1 <- tmp1"; this is possible for a great many Movs, and allows us to be much more regular in the phase that generates LLVMNOP without compromising code size. We then prioritize temporaries that appear in a "Load16 tmp1 <- tmp2" instruction since we have a nice trick for that one when both are the same. After that, we just greedily coalesce temporaries until it is no longer possible. Fancier register allocation techniques like graph coloring would work here (this part of the compiler is very traditional), but there's not much need: We have over 40 16-bit temporaries, all of which are just as efficient to access, so we mainly just want to keep the total number used small so that EBP offsets are printable. Having a smaller temporary frame size allows deeper recursion, as well.

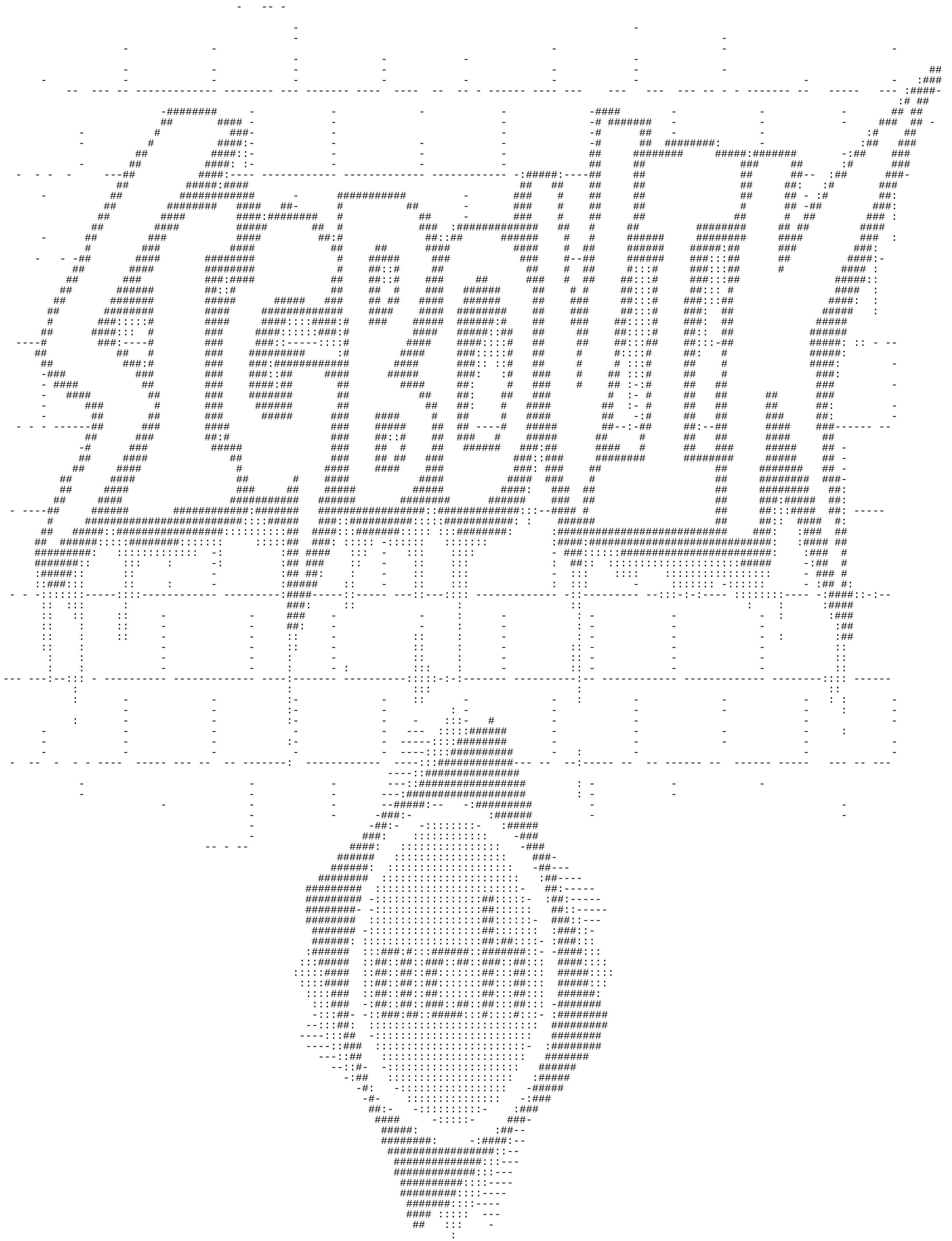
The compilation strategy ends up storing almost all immediate results in temporaries, which is not that suboptimal since all operations need to be between a register and memory anyway. However, many pairs of instructions could keep a just-computed value in a register rather than bothering to write it. This is not yet implemented, but the idea is that we could introduce a small number of registers (probably just one?) in addition to the numbered temporaries, and use those in the output of Allocation. This could produce significantly closer to hand-written code, without the need to change much in the backend.

** 23. CIL **

The intermediate language that precedes the named LLVMNOP code is called CIL, for C Intermediate Language. It's intended to be a desugared and more explicit version of C. Some examples of the of CIL grammar:

```
  signedness ::= Signed | Unsigned

  type ::= Pointer type
         | Code type, type list
         | Word32
         | Word16
         | Word8
         | ...

  builtin ::= B_EXIT | B_ARGC | B_ARGV | B_PUTC | B_OUT8

  value ::= Var v
          | AddressLiteral loc, type
          | FunctionLiteral name, type, type list
          | Word8Literal w8
          | Word16Literal w16
          | Word32Literal w32

  exp ::= Value value
        | Plus width, value, value
        | LessEq width, value, value
        | Load width, value
        | Promote width, width, signedness, value
        | Call value, value list
        | Builtin builtin, value list
        | ...

  stmt ::= Bind v : type = exp in stmt
         | Store width value = value in stmt
         | GotoIf cond, string, stmt
         | Return value
         | ...
```

And lots more stuff. A program is a collection of functions, each of which is a collection of named statements (the stmt type is recursive, with a single statement representing a series of C statements until we reach a Return or unconditional Goto). Programs also have a set of globals with initialization code for them. Note that CIL has ML-style lexically scoped variables which are only in scope for the given block. Since C's semantics for variables allow them to be addressed and modified, we convert all C variables into explicit loads from and stores to memory.



The CIL language is typed, with one important use of this being that we determine the calling convention for a function pointer from its type. (This includes the size of the return address slot, which is on the locals stack and shared between the caller and callee, as well as the number and sizes of the arguments, also on the locals stack.) We make the representation (Word8, Word16, Word32) of integral types explicit, but signed and unsigned ints are represented the same way (just as on the processor itself). Instead, expressions like Promote (which converts e.g. an 8-bit word to a 16-bit word) are explicit about whether they perform sign extension. We are careful to distinguish between 8-, 16- and 32-bit quantities throughout the compiler, because printable x86 has the ability to work with all three widths, and we can produce significantly better code if we can use the correct width. (As a simple example, loading a 16-bit word is much cheaper than the zero-extended 32-bit version.)

Some low-level ideas are threaded throughout the compiler. In the case of "out8" and "exit", for example, these are available to the programmer if she simply declares them:

```
     int _out8(int, int);
     int _exit(int);
```

They can be called like _exit(1), but are translated to the Builtin expression rather than a function call. It is not permitted to take their addresses.

Unlike LLVMNOP, we have both expression forms of operators and "cond" forms. The expression forms evaluate to 1 or 0, whereas the cond forms are only used as a combined test-and-branch in the GotoIf construct. Optimizations try to put these in the most useful form for later work.

** 24. Optimization **

CIL code is optimized via a series of conservative transformations until no more simplifications are possible. Among important optimizations are dead variable removal and constant folding, which clean up the code generated by the translation from C to CIL. Lots more is possible here, but since these problems are not specific to printable x86, I did not spend that much time on optimization. The main thing is to keep the code size for the programs we want to write under the 64k limit. There is a natural tension between implementing optimizations for the "high-level" CIL language (which is easier to analyze) and the low-level LLVMNOP language (more flexibility, access to incidental tricks that don't make sense at the high level, and opportunity to clean up after more of the compiler's work).

Optimizations are implemented using the "Pass" functor idea presented in my Ph.D. dissertation [MTMC'08].

The optimization phase is also responsible for eliminating some features from the language so that we don't need to think about them when converting to LLVMNOP:

- Multiplication. In printable x86, we have access to the IMUL instruction, but only versions that multiply by a constant immediate value (opcodes 0x6B, 0x69). Since that immediate needs to be printable, this instruction is not very useful -- we can't even use it to implement multiplication by arbitrary constants. Instead, the "Optimization" phase for CIL replaces the Times expression with a function call to a built-in hand-written routine that implements multiplication by repeated addition.

- Comparison ops. Expressions like LessEq are transformed into GotoIf(cond, ...), since we don't have any way of comparing values without also branching.

- String literals. These are replaced with references to globally-allocated arrays.

- Global initialization. All initialization code for globals (e.g. int global = 15;) is moved into a wrapper around the main function.

These tasks aren't really optimizations, but we want to perform optimization both before and after doing them. So optimization code needs to at least be aware of their existence so that it doesn't e.g. reintroduce string literals after they have been eliminated!

** 25. Converting to CIL **

The frontend of the compiler uses the ckit library [CKIT'00] to parse the input C code into an ML datatype called "AST." The details of this language are mostly uninteresting, but it is mostly in direct correspondence to C89 itself. When we convert to CIL, we remove "syntactic sugar" constructs that can be built from more fundamental things. "For" example, a for loop is broken apart into a few gotos. The && and || operators make their short-circuiting behavior explicit by sequencing the tests. Implicit widening and narrowing between types is made explicit. Compound assignment ops like ^= and ++ are sequenced into the primitives that make them up. Array subscripts and structure references are converted into pointer arithmetic. Although there's a lot of code involved to implement C, it is mostly standard.

** 26. Limitations **

ABC has some limitations, some of which are fundamental and some of which are simply due to the unconscionably strict SIGBOVIK deadlines:

- Floating point is not available. We have access to none of the floating point instructions, so native support is not really possible. It would be possible to provide software implementations of the floating-point operations; prior to the Intel 80486, support for floating point was usually provided in software anyway, so this helps us avoid anachronism.

- Standard libraries are not available. Since we can only call the DOS INT 0x21 handler one time, and we use that to exit, there is no way to access the filesystem or write to the console. One could conceivably write their own device drivers using I/O ports (see the next section), but this usually also involves using or implementing hardware interrupts, so probably wouldn't pan out.

- malloc/free. This can be supported in software, with no significant limitations other than the amount of memory available.

- Operand widths. Though ABC architecturally supports most operations at 8, 16, and 32 bit widths, most operations are only implemented for 16 bit operands. This is easily fixed, but should be done with some care to correctness and performance.

- Performance. Multiplication is linear time, since we use a software routine. This can be done (somewhat) better, but will always involve loops in the general case. Other constructs like "if" and "while" can have unexpectedly bad performance due to the "ladder" technique for control flow; these issues can make algorithms perform asymptotically worse than they should.

- Division and modulus. These need to be done in software like multiplication, which is trickier than usual due to the lack of efficient bit shifts. Note that many computer processors don't even have an integer division instruction (e.g. Alpha, 6502), so this is not even that weird.

- struct copying. Not a huge deal, but it means emitting code that copies struct field-by-field because we don't have anything like memcpy, and around the time of a function call or return, the state of the machine is pretty delicate.

- sizeof. Actually sizeof is so easy I just went and implemented it just now, instead of writing this sentence. I saved further time by not deleting the previous sentence.

- Bit fields. These are garbage so nobody implements them unless they have to. No fundamental limitation here, although the compiler does assume that lvalues have an address.

I am shamed that ABC does not compile the complete feasible subset. Perhaps check http://tom7.org/abc/ for an updated version, published postpartum.

** 27. Programming **

Since we're working in reverse order, we've reached the very front of the ABC compiler, and now can talk about the program we feed to it.

Obviously the program that is this paper should do something, but so far we've only talked about how to do loops and exit. We do have access to the command line via the PSP (properly piped through to argv), and we do have the possibility of looping forever, or exiting with some status. These would at least demonstrate computation, but are pretty lame, let's be honest.

A natural thing to do when thinking about "printable x86" would be to have the paper print itself out, i.e., a quine. This would be quite challenging given the ratio of accessible data (64kb data segment + data embedded in the 64k of code) to the size of the paper itself (409k), but it might be possible. Sadly, the major obstacle is that we cannot repeatedly invoke INT 21, so we cannot print anything out.

Like some kind of miracle, though, two of the opcodes available to us in printable x86 are practically made for I/O. In fact they are literally made for I/O, and in fact their names are INS and OUTS. These are part of a family of CPU instructions that interact with peripherals on the motherboard. DOS uses these to implement some of its INT 21 system calls (e.g., to talk to the disk controller to implement the file system), but I/O ports are sometimes also used by application programmers.

In this case, there is one nice piece of hardware that is standard on DOS-era computers, and that grabbed a standard set of port numbers before the concept of configuring I/O was a thing: The Adlib FM synthesis card. By writing bytes to various ports, we can make this thing make stupid sounds.

The out8 primitive I've mentioned a few times provides a way for the C programmer to access the OUTS instruction. OUTS is actually a routine intended for writing a whole string to an I/O port, but we can set things up so that it just writes one byte. We temporarily locate the string at offset DS:0000, i.e., what the "null pointer" points to, for efficiency and to avoid interfering with any program data. Incidentally, this also gives us style points for using the rare instruction

```
     AND [SI] <- SI
```

which bitwise-ands an address into the thing the address points to (!), because we know SI is 0.

... Oh wait, here comes the code segment!

```
                                                                                                                               NuTj X4 g!E g1] ,~4}Ph  DXD,jg)E
 j X4 P_g3} 3=g1} P^Fu                                 NuWj X4*P^$ g3E tHj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P^Fu                                NuTj X4 P_g3} 3
=g1} g!E"g1]",~4}Ph  DXD,jg)E"j X4 P^Fu                                 Nu[j X4 P_g3}"3=g1}",~4}Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4 P^Fu
      NuXj X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^P_g3u$!<g3} 1<P^Fu                              NuXj X4 g!E$g1]$,~4}Ph  DXD,Dg)E$j X4 P^P_g3u$!<g3}"1<P^Fu
                            NuYj X4"PLXD$@$ 44P%@ g!E Xg1E j X4 g3E PEEEEEEEEj X,aP^4Uq                             NuPMMMMMMMMj4XP44g!E(Xg1E(g+](j X4"PLXD$
@$ 4.P^4Uq                                Nuyj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} ,~,|P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
       Nuej X4 g!E"g1]",~4}Ph  DXD,pg)E"j X4 P_g3}"3=g1}"@PHg!E$Xg1E$j X4 P^Fu                              NuTj X4 g3E$g)E"j X4 P^P_g3u !<g3}"1<@@PLXD$@$ 4*
P^4Uq                                 Nuij X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} ,~HHP$ g!E$Xg1E$j X4 P^Fu                                Nuyj X,!Ph  DX
D,TPj X4 g!E(Xg1E(g+](j X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^P_g3u"!<g3} 1<P^Fu                              Nuvj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^P_g3u"!<g3
}$1<@@PLXD$@$ 4<P%@ g!E Xg1E j X4 P^Fu                                Nu=j X4 g3E PEEEEEEEEj X,iP^4Uq                               NuPMMMMMMMMj4XP44g!E(Xg1E
(g+](j X4"PLXD$@$ 4&P^4Uq                                 Nuzj X,!Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4"PLXD$@$ 4>P%@ g!E Xg1E j X4 g3E PEEEEEEEEj X,XP^4Uq
                       NuIMMMMMMMMj4XP44g!E(Xg1E(g+](j X4 g!E P^Fu                                           Nupj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^HHHHHH
HPj X4 g!E(Xg1E(g+u(u                                 Nuij X4!PHg!E$Xg1E$j X4!Ph  DXD4"P%  g!E Xg1E j X4 g!E"P^P_g3u !<g3}"1<P^Fu
  Nujj X4!PLXD$@$ P%  g!E&Xg1E&j~X@@@P$ g!E"Xg1E"j X4 P^P_g3u&!<g3}"1<CCCCP^Fu                               NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g
3}$1<P^Fu                                NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}&1<P^Fu                              Nuaj X4"Ph  DXD4eP%  g!E Xg
1E j X4 g3E PEEEEEEEEj X4!Ph  DXD,1P^4Uq                                Nu MMMMMMMMKKKKh LX4 cya!         4G4Gq~
                                                                                4G4Gq~
                                                    4G4Gq~
                        4G4Gq~                                                                                                                                      4G4G
q~                                                                                                                             4G4Gq~
```

```
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~            4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~            4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~            4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~            4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~            4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~            4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~
4G4Gq~                                                                                 @@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@NujDf% @ @f%@ @ fPf[fPf^fPf_,|P[df37,lP[df!?df17fW$(Ph  DXD@Pf[5AvPf]-‘~P^Fu          Nuaj X4(PLXD$@$ 4?P
%@ g!E Xg1E j X4!Ph  DXD4$P%  g!E"Xg1E"j X4 P^Fu          NuZj X4 P^P_g3u !<g3}"1<,~,zPLXD$@$ 4=P%@ g!E Xg1E j X4 P^Fu
           NuUj X4!Ph  DXD4,P%  g!E"Xg1E"j X4 P^P_g3u !<g3}"1<P^Fu          Nuqj X4(PLXD$@$ 4;P%@ g!E Xg1E j X4!Ph  DXD4gP%  g!E"Xg1E"
j X4 P^P_g3u !<g3}"1<P^Fu          Nuaj X4(PLXD$@$ 49P%@ g!E Xg1E j X4"Ph  DXD,~P%@ g!E"Xg1E"j X4 P^Fu          Nu
Zj X4 P^P_g3u !<g3}"1<,~,zPLXD$@$ 47P%@ g!E Xg1E j X4 P^Fu          NuWj X4"Ph  DXD,)Pj X4 g!E"Xg1E"j X4 P^P_g3u !<g3}"1<P^Fu
           Nuqj X4(PLXD$@$ 45P%@ g!E Xg1E j X4$Ph  DXD4}P%  g!E"Xg1E"j X4 P^P_g3u !<g3}"1<P^Fu          Nuaj X4(PLXD$@$ 43P%@ g
!E"Xg1E"j X4$Ph  DXD,BP%  g!E Xg1E j X4 P^Fu          NuZj X4 P^P_g3u"!<g3} 1<,~,zPLXD$@$ 41P%@ g!E Xg1E j X4 P^Fu
         NuVj X4%PLXD$@$ 4_P%  g!E"Xg1E"j X4 P^P_g3u !<g3}"1<P^Fu          NuWj X4(PLXD$@$ 4’P%@ g!E Xg1E j X4)P$ g!E"Xg1E"j X4 P^Fu
           Nu~%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05,~,zPh  DXD4>P%  g!E Xg1E j X4(P$ g!E"Xg1E"j X4 P^Fu          Nu
m%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05,~,zPh  DXD44P%  g!E Xg1E j X4 P^Fu          NuOj X4)P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 g!E"P^Fu
           NuZj X4 P_PZg3}  5g2u"05,~,}PLXD$@$ 4_P%  g!E Xg1E j X4 P^Fu          NuWj X4"Ph  DXD4SPj X4 g!E"Xg1E"%@@
%  Hg3E"@g)E j X4 P^Fu          NuMj X,!P$~4~g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nuaj X4%PLXD$@$ 4_P%  g!E
 Xg1E j X4"Ph  DXD,lP%@ g!E"Xg1E"j X4 P^Fu          Nu~%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05,~,~Ph  DXD,BP%  g!E Xg1E j~X@P4~Hg!E"Xg1
E"j X4 P^Fu          NuN%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05P^Fu          Nujj X4$Ph  DXD,BP%  g!E Xg1E j~X@@
P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 g!E"P^Fu          NuYj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu
      Nujj X,wP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nukj X4!PHg!E"Xg1E"%@@%  Hg3E"@g)E j X,mP$@g!
E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nufj X4$Ph  DXD4}P%  g!E Xg1E j X4"P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
         Nu{j X,cP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4#P$ g!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E j X,W
P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nulj X4$P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,KP$ g!E"Xg1E"j X4 P
_PZg3}  5g2u"05P^Fu          Nufj X4$Ph  DXD4}P%  g!E Xg1E j X4%P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu}
j X,?P,~4cg!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4&P$ g!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,1P,~4qg!E"Xg
1E"j X4 P^Fu          Nujj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4’P$ g!E"Xg1E"j X4 P^Fu          Nu|%@@
%  Hg3E"@g)E j X,#P$}4}g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nu\j X4(P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X
4,P$ g!E"Xg1E"j X4 P^Fu          Nuzj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4)P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
           Nu{j X4<P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4/P$ g!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E
"@g)E j X,~43P$ g!E"Xg1E"j X4 P^Fu          Nujj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X40P$ g!E"Xg1E"j X4 P^Fu
         Nu[%@@%  Hg3E"@g)E j X,wP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nuwj X4$Ph  DXD4}P%  g!E Xg1E j X41P$ g!E"Xg1E"%@@
%  Hg3E"@g)E j X,^P$ g!E"Xg1E"j X4 P^Fu          Nuzj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X42P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4
P^Fu          Nu{j X,CP$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X43P$ g!E"Xg1E"j X4 P^Fu
   Nu|%@@%  Hg3E"@g)E j X,&P,~4|g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nulj X44P$ g!E"Xg1E"%@@%  Hg3E
"@g)E j X48P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nufj X4$Ph  DXD4}P%  g!E Xg1E j X48P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
           Nu{j X,}P$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X49P$ g!E"Xg1E"j X4 P^Fu          Nuz%@@%
 Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nunj X4:P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,*P,
~4xg!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nuyj X4$Ph  DXD4}P%  g!E Xg1E j X4=P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P^F
u          Nuzj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4>P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          N
uzj X,fP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j XHP$ g!E"Xg1E"j X4 P^Fu          Nu|%@@%  Hg3E"@g)E j X,-P,~4ug!E"Xg1
E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nujj XP4 g!E"Xg1E"%@@%  Hg3E"@g)E j X48P$ g!E"Xg1E"j X4 P_PZg3}  5g2u
"05P^Fu          Nuuj X4$Ph  DXD4}P%  g!E Xg1E j$XP4$g!E"Xg1E"%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"j X4 P^Fu
Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j%XP4%g!E"Xg1E"j X4 P^Fu          NuK%@@%  Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P^Fu
           Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j&XP4&g!E"Xg1E"j X4 P^Fu          Nu|%@@%  Hg3E"@g)E j X,*P,~4
xg!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nulj)XP4)g!E"Xg1E"%@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P_P
Zg3}  5g2u"05P^Fu          Nudj X4$Ph  DXD4}P%  g!E Xg1E j*XP4*g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nuyj X,
fP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j+XP4+g!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,-P,~4ug!E"Xg1E"j X4
P^Fu          Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j,XP4,g!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g
)E j X48P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          NuZj0XP40g!E"Xg1E"%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"
j X4 P^Fu          Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j1XP41g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
     Nuyj X,UP$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j2XP42g!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,*P,~4xg!
E"Xg1E"j X4 P^Fu          Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j5XP45g!E"Xg1E"j X4 P^Fu          Nu]%
@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nuuj X4$Ph  DXD4}P%  g!E Xg1E j6XP46g!E"Xg1E"%@@%  Hg3E"@g)E j X,fP$
@g!E"Xg1E"j X4 P^Fu          Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j7XP47g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
         Nu{j X,-P,~4ug!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j8XP48g!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E
j X48P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nujj<XP4<g!E"Xg1E"%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"j X
4 P_PZg3}  5g2u"05P^Fu          Nuuj X4$Ph  DXD4}P%  g!E Xg1E j=XP4=g!E"Xg1E"%@@%  Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P^Fu
           Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j>XP4>g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,*P,~4xg!E"Xg1E"
j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jAXP$ g!E"Xg1E"j X4 P^Fu          Nu|%@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P_PZg3}  5g2u
"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          NuZjBXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,fP$@g!E"Xg1E"j X4 P^Fu
    Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jCXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,-P,~4ug!E"Xg1E"j X4 P_PZg3
}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jDXP$ g!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E j X48P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DX
D4}P%  g!E Xg1E j X4 P^Fu          NujjHXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu
      Nuuj X4$Ph  DXD4}P%  g!E Xg1E jIXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P^Fu          Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  D
XD4}P%  g!E Xg1E jJXP$ g!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,*P,~4xg!E"Xg1E"j X4 P^Fu          Nuhj X
4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jMXP$ g!E"Xg1E"j X4 P^Fu          Nu|%@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05
,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          NujjNXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,fP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu
             Nudj X4$Ph  DXD4}P%  g!E Xg1E jOXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,-P,~4ug!E"Xg1E"j X4 P_PZg3}  5g2u
"05,~,~Ph  DXD4}P%  g!E Xg1E jPXP$ g!E"Xg1E"j X4 P^Fu          NuK%@@%  Hg3E"@g)E j X48P$ g!E"Xg1E"j X4 P^Fu
    Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jTXP$ g!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"j X4 P_PZg3}
 5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          NuZjUXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P^Fu
        Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jVXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,*P,~4xg!E"Xg1E"j X4 P
_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jYXP$ g!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P^Fu
             Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jZXP$ g!E"Xg1E"j X4 P^Fu          Nu[%@@%  Hg3E"@g)E j X,fP$@g!E"Xg1E
"j X4 P_PZg3}  5g2u"05P^Fu          Nuwj X4$Ph  DXD4}P%  g!E Xg1E j[XP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X,-P,~4ug!E"Xg1E"j X4 P^Fu
             Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j\XP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nuyj X48P$ g!E"X
g1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j‘XP4‘g!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"j X4 P_PZg3}  5g
2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          NujjaXP4ag!E"Xg1E"%@@%  Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu
             Nuwj X4$Ph  DXD4}P%  g!E Xg1E jbXP4bg!E"Xg1E"%@@%  Hg3E"@g)E j X,*P,~4xg!E"Xg1E"j X4 P^Fu          Nuxj X4 P_PZg3}
 5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jeXP4eg!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,~4’P$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DX
D4}P%  g!E Xg1E jfXP4fg!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E j X,fP$@g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4
 P^Fu          NuljgXP4gg!E"Xg1E"%@@%  Hg3E"@g)E j X,-P,~4ug!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nuuj X4$Ph
 DXD4}P%  g!E Xg1E jhXP4hg!E"Xg1E"%@@%  Hg3E"@g)E j X48P$ g!E"Xg1E"j X4 P^Fu          Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j
lXP4lg!E"Xg1E"j X4 P^Fu          NuK%@@%  Hg3E"@g)E j X,}P$@g!E"Xg1E"j X4 P^Fu          Nuhj X4 P_PZg3}  5g2u"05,~
,~Ph  DXD4}P%  g!E Xg1E jmXP4mg!E"Xg1E"j X4 P^Fu          Nu[%@@%  Hg3E"@g)E j X,UP$ g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu
         Nuwj X4$Ph  DXD4}P%  g!E Xg1E jnXP4ng!E"Xg1E"%@@%  Hg3E"@g)E j X,*P,~4xg!E"Xg1E"j X4 P^Fu          Nuhj X4 P_PZg3}  5g2u"0
5,~,~Ph  DXD4}P%  g!E Xg1E jqXP4qg!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,~4’P$ g!E"Xg1E"j X4 P^Fu
    Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jrXP4rg!E"Xg1E"j X4 P^Fu          Nuz%@@%  Hg3E"@g)E j X,fP$@g!E"Xg1E"j X4 P_PZg3}
 5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          NuljsXP4sg!E"Xg1E"%@@%  Hg3E"@g)E j X,-P,~4ug!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu
                 Nudj X4$Ph  DXD4}P%  g!E Xg1E jtXP4tg!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,!P$~4~g!E"Xg1E"j X4 P_P
Zg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E juXP4ug!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,!P$~4~g!E"Xg1E"j X4 P^Fu
           Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jvXP4vg!E"Xg1E"j X4 P^Fu          Nu|%@@%  Hg3E"@g)E j X,!P$~4~g!E"Xg1E
"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nu\jwXP4wg!E"Xg1E"%@@%  Hg3E"@g)E j X,!P$~4~g!E"Xg1E"j X4 P^Fu
               Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jxXP4xg!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu          Nu{j X,!P$~4
~g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jyXP4yg!E"Xg1E"j X4 P^Fu          NuM%@@%  Hg3E"@g)E j X,!P$~4~g!E"Xg1E"j X4 P^F
u          Nuhj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E jzXP4zg!E"Xg1E"j X4 P^Fu          Nu]%@@%  Hg3E"@g)E
j X,!P$~4~g!E"Xg1E"j X4 P_PZg3}  5g2u"05P^Fu          Nuwj X4$Ph  DXD4}P%  g!E Xg1E j{XP4{g!E"Xg1E"%@@%  Hg3E"@g)E j X,!P$~4~g!E"Xg1E"j X
4 P^Fu          Nuxj X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j|XP4|g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu
  Nu{j X,!P$~4~g!E"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j}XP4}g!E"Xg1E"j X4 P^Fu          Nu|%@@%  Hg3E"@g)E j X,!P$~4~g!E
"Xg1E"j X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j X4 P^Fu          Nulj~XP4~g!E"Xg1E"%@@%  Hg3E"@g)E j X,!P$~4~g!E"Xg1E"j X4 P_PZg3}
```

```
  5g2u"05P^Fu                                  Nujj X4$Ph  DXD4}P%  g!E Xg1E j~X@P4~Hg!E"Xg1E"%@@%  Hg3E"@g)E j X4 g!E"P^Fu                              Nuzj
X4 P_PZg3}  5g2u"05,~,~Ph  DXD4}P%  g!E Xg1E j~X@@P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu                             Nuzj X4 g!E"P_PZg3}  5g2u"05@@Ph  DXD,)P
j X4 g!E Xg1E j X4!Ph  DXD4EPj X4 g!E"Xg1E"j X4 P^Fu                                 Nuk%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05@@Ph  DXD,~P%@ g!E Xg1E j X4 P^
Fu                                Nu]jTXP$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05P^Fu                                Nutj X4!Ph  DXD4gP%  g!E Xg1E
 j X4!Ph  DXD4zP%  g!E"Xg1E"%@@%  Hg3E"@g)E j X4 g!E"P^Fu                           Nuej X4 P_PZg3}  5g2u"05@Ph  DXD4,P%  g!E Xg1E j:XP4:g!E"Xg1E"j X4 P^F
u                                Nu{%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05@Ph  DXD4$P%  g!E Xg1E j X4'P$ g!E"Xg1E"j X4 P^Fu
NuR%@@%  Hg3E"@g)E j X4 g!E"P_PZg3}  5g2u"05CCCCP^Fu                              NuUj X,~44P$ g!E Xg1E j X4 g3E PEEEEj X4!Ph  DXD,uP^4Uq
            NuXMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"P^Fu                               NuzKKKKj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P^P_g3u !<g3}"
1<^,~4|Ph  DXD4qPj X4 g!E$Xg1E$g+u$u                          Nu\j X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu
    Nuwj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} ,~,~P^$ g3E t:j X4!PHg!E"Xg1E"j X4 P^Fu                                Nuej X4 g3E"g)E j X4 g!E"g1]",~4}P
h  DXD,Dg)E"j X4 P^P_g3u"!<g3} 1<P^Fu                     Nu~j X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,Fg)E"j X4 P_g3}"3=g1
}"P^Fu                         Nut%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Fg)E"j X4 P^P_g3u"!<g3} 1<@@Ph  DXD4cP^4Uq                                       Nu
Tj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} P^Fu                            Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4|Ph  DXD4xPj X4 g
!E$Xg1E$g+u$u                         Nusj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} @@@PLXD$@$ ,xP%  g!E"Xg1E"j X4 P^Fu
    Nuoj X4 PPP^_Z!4g3} 1<g3U"ng!E g1] ,~4}Ph  DXD,Fg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                          Nuij X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3
} 3=g1} ,~,vP$ g!E"Xg1E"j X4 P^Fu                                  Nudj X4#P^$ PYg3M"g;M ~Og!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} P^Fu
       NuJj X4!PHg!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu                        Nudj X4 g!E"g1]",~4}Ph  DXD,Fg)E"j X4 P^P_g3u"!<g3} 1<@@Ph  DXD4cP^4Uq
                               Nusj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} @@@PLXD$@$ ,wP%  g!E"Xg1E"j X4 P^Fu                              Nuoj X4 PPP^_
Z!4g3} 1<g3U"ng!E g1] ,~4}Ph  DXD,Fg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                            Nugj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} 4TP$ g!E"
Xg1E"j X4 P^Fu                           Nudj X4#P^$ PYg3M"g;M ~Og!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} P^Fu                                    NuJj X4!PH
g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu                         Nudj X4 g!E"g1]",~4}Ph  DXD,Fg)E"j X4 P^P_g3u"!<g3} 1<@@Ph  DXD4cP^4Uq
         Nuzj X4 g!E g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4|Ph  DXD4EPj X4 g!E$Xg1E$g+u$u                              Nugj X4 g!E g1] ,~4}Ph  D
XD,Bg)E j X4 P_g3} 3=g1} ,PP$@g!E"Xg1E"j X4 P^Fu                            Nud%@@%  Hg3E @g)E"j X4 g!E$CCCCCCCCg!E g1] ,~4}Ph  DXD,Bg)E j X4 P^Fu
                       Nu{j X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}$1<,PP$@g!E Xg1E j X4 P^Fu                              NuEj X4 g3
E PEEEEEEj X4"PLXD$@$ 46P^4Uq                          NuuMMMMMMMKKKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} ,'P$@g!E$Xg1E$j X4 P^Fu
                   Nub%@@%  Hg3E @g)E$j X4(PLXD$@$ 45P%@ g!E Xg1E j X4 P_g3} 3=g1} P^Fu                           NuTj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4
 P_g3}"3=g1}"P^Fu                         NuL%@@%  Hg3E"@g)E j X4 P_PZg3} 25g!E"g0u"P^Fu                                 Nu'CCCCCCCCj X4 g!E g1] ,~4}Ph
 DXD,Bg)E j X4 P^P_g3u !<g3}$1<P^Fu                                Nukj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}"1<,IP$@g!E Xg1E j X4 P^Fu
               NuEj X4 g3E PEEEEEEj X4"PLXD$@$ 4/P^4Uq                           NuuMMMMMMMKKKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} ,PP$@g
!E$Xg1E$j X4 P^Fu                        Nub%@@%  Hg3E @g)E$j X4(PLXD$@$ 43P%@ g!E"Xg1E"j X4 P_g3}"3=g1}"P^Fu                                  NuTj X4 g
!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P^Fu                         NuL%@@%  Hg3E @g)E"j X4 P_PZg3}"25g!E"g0u"P^Fu                                  Nu'
CCCCCCCCj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}$1<P^Fu                            Nukj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}"1<,BP$@g!E X
g1E j X4 P^Fu                      NuEj X4 g3E PEEEEEEj X4"PLXD$@$ 4(P^4Uq                              NuRMMMMMMMKKKKKKKK^j X,"Ph  DXD4XPj X4 g!E&
Xg1E&g+u&u                      NuEj X,pP$@g!E"Xg1E"j X4 g!E$CCCCCCCP^Fu                              Nuvj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u
 !<g3}"1<g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^Fu                         Nuhj X4 P^P_g3u !<g3}$1<,>P$ g!E Xg1E j X4 g3E PEEEEEEj X4"PLXD$@$ 4$P^4Uq
                   NuGMMMMMMMKKKKKKj X,oP$@g!E"Xg1E"j X4 P^Fu                          Nubj X4 g!E$CCCCCCg!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1
<P^Fu                          Nukj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}$1<,:P$ g!E Xg1E j X4 P^Fu                                   NuBj X4 g3E
PEEEEEEj X4"Ph  DXDP^4Uq                              NuGMMMMMMMKKKKKKj X,nP$@g!E"Xg1E"j X4 P^Fu                            Nubj X4 g!E$CCCCCCg!E g1] ,~4
}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                        Nukj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}$1<,6P$ g!E Xg1E j X4 P^Fu
                   NuDj X4 g3E PEEEEEEj X4"Ph  DXD4<P^4Uq                          NutMMMMMMMKKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4!PHg!E"Xg1E"j X4 P
^P_g3u !<g3}"1<P^Fu                     Nu{j X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} ,~4wP,~4wg!E"Xg1E"j X4'P^$ PYg3M"g;M |%P^Fu
               Nu^j X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} g!E$CCCCCCCP^Fu                             NuXj X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^P_g3u
"!<g3} 1<P^Fu                       NuXj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^P_g3u"!<g3}$1<P^Fu                                NuUj X,PP$ g!E Xg1E j X
4 g3E PEEEEEEj X4"Ph  DXD46P^4Uq                      NuqMMMMMMMKKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu
                  Nut%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^P_g3u"!<g3} 1<@@Ph  DXD4'P^4Uq                           NuE^j X,"PLXD$@$ ,tP
j X4 g!E&Xg1E&g+u&u                 Nu\j X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                                 Nuyj X4 g!E g
1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P_PZg3} 25g!E g0u ,~,}P^$ g3E t*j X4 P^Fu                    Nuej X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1}
 @PHg!E"Xg1E"j X4 P^Fu                   NuX%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^Fu                                  Nuuj X4 P^P_g3
u"!<g3} 1<g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu                      Nut%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4
P^P_g3u"!<g3} 1<@@Ph  DXD4bP^4Uq                   NuTj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P^Fu                                  Nuwj X4
g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4|PLXD$@$ ,lPj X4 g!E$Xg1E$g+u$u                  Nu\j X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 g!E"P^P_
g3u !<g3}"1<P^Fu                          Nu~j X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,Jg)E"j X4 P_g3}"3=g1}"P^Fu
                Nuj%@@%  Hg3E"@g)E j X4 P_PZg3} 25g!E"g0u"g!E g1] ,~4}Ph  DXD,Fg)E j X4 P^Fu                     NuXj X4 P^P_g3u !<g3}"1<g!E g1] ,
~4}Ph  DXD,Dg)E j X4 P^Fu                        NuTj X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,Jg)E"j X4 P^Fu                                NuWj X4 P_g3}"3
=g1}"Hg3E"@g)E j X4 P_PZg3} 25g!E"g0u"P^Fu                     NuXj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P^P_g3u !<g3}"1<P^Fu
      NuTj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} P^Fu                        NuTj X4 g!E"g1]",~4}Ph  DXD,Hg)E"j X4 P_g3}"3=g1}"P^Fu
                  Nudj X4#P^$ PYg3M g;M"uOg!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} P^Fu                        Nutj X4#P^$ g3E tej X4 g!E g1] ,~4}Ph
  DXD,Jg)E j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu                      Nuu%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Jg)E"j X4 P^P_g3u"!<g3} 1<@@PLXD$@
$ 4<P^4Uq                           Nu{j X4 g!E g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4|PLXD$@$ ,_Pj X4 g!E$Xg1E$g+u$u
        NuXj X4!PHg!E Xg1E j X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^Fu                        NuXj X4 P^P_g3u"!<g3} 1<^,~4|Ph  DXD,}Pj X4 g!E$Xg1E$g+u$u
                      NuTj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                          Nudj X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,Dg)
E"j X4 P^P_g3u"!<g3} 1<P^Fu                     Nufj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P_PZg3} 25g!E"g0u"P^Fu
     Nusj XP4 g!E Xg1E j X4'P^$ PYg3M"g;M uOg!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P^Fu                     NuJj X4!PHg!E"Xg1E"%@@%  Hg3E"@g)E j
X4 P^Fu                         Nudj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^P_g3u"!<g3} 1<@@Ph  DXD4cP^4Uq                           Nufj X4 g!E g1] ,
~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P_PZg3} 25g!E"g0u"P^Fu                   Nuuj X4-P$ g!E Xg1E j X4%P^$ PYg3M"g;M tOg!E g1] ,~4}Ph  DXD,Dg)E j X4 P
_g3} 3=g1} P^Fu                       NuJj X4!PHg!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu                            Nudj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j
X4 P^P_g3u"!<g3} 1<@@Ph  DXD4cP^4Uq                   Nurj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^Fu
                     NuRj X4 P_g3}"3=g1}"P^P_g3u !<g3}"1<@@Ph  DXD4aP^4Uq                       Nuhj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1}
g!E"P_PZg3}  5g2u"05P^Fu                          Nuzj X4 g!E g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4|Ph  DXD,oPj X4 g!E$Xg1E$g+u$u
                   Nugj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} 4AP$ g!E"Xg1E"j X4 P^Fu                       NuUj X4 g3E"g)E j X4 g!E"g1]",~4
}Ph  DXD,Lg)E"j X4 P^Fu                    NuZj X4 P^P_g3u"!<g3} 1<,~,zPLXD$@$ 4?P%@ g!E Xg1E j X4 P^Fu                              Nu'j X4 P_g
3} 3=g1} g!E"g1]",~4}Ph  DXD,Lg)E"j X4 P_g3}"3=g1}"P^Fu                              Nuz%@@%  Hg3E"@g)E j X4 P_PZg3} 25g!E"g0u"g!E g1] ,~4}Ph  DXD,Hg)E j X4
P^P_g3u !<g3}"1<P^Fu                   NuTj X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P_g3}"3=g1}"P^Fu                            Nu'j X4 g!E g1] ,~4}
Ph  DXD,Fg)E j X4 P_g3} 3=g1} P_g3} 3=g1} P^Fu                        Nuz%@@%  Hg3E @g)E"j X4 P_PZg3}"25g!E"g0u"g!E g1] ,~4}Ph  DXD,Jg)E j X4 P^P_g3u !
<g3}"1<P^Fu                      NuTj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} P^Fu                             Nu\j'XP4'g!E"Xg1E"j X4#P^$ PY
g3M g;M"t84,P4,g!E"Xg1E"j X4 P^Fu                          Nudj X4'P^$ PYg3M g;M"tOg!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} P^Fu
      Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4|Ph  DXD,cPj X4 g!E&Xg1E&g+u&u                          Nuij X4 g!E g1] ,~4}Ph  DXD,Hg
)E j X4 P_g3} 3=g1} ,~,vP$ g!E"Xg1E"j X4 P^Fu                    Nuh%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Hg)E"j X4 P^P_g3u"!<g3} 1<P^Fu
                     NuTj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} P^Fu                       Nu'j X4 g!E"g1]",~4}Ph  DXD,Fg)E"j X4 P_g3}"3=g1
}"P_g3}"3=g1}"P^Fu                     Nugj X4!PHg!E$Xg1E$%@@%  Hg3E$@g)E"j X4 P^P_g3u !<g3}"1<@@PLXD$@$ 4;P^4Uq                                Nui
j X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} ,~,vP$ g!E"Xg1E"j X4 P^Fu                        Nuqj X4 g3E"g)E j X4 g!E"g1]",~4}Ph  DXD,Hg)E"j X4 P^P_
g3u"!<g3} 1<@@Ph  DXD4bP^4Uq                   Nu~j X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P_g3}"3=g1}"g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P^Fu
                      NuWj X4 P_g3} 3=g1} Hg3E @g)E"j X4 P_PZg3}"25g!E"g0u"P^Fu                      NuXj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P^
P_g3u !<g3}"1<P^Fu                     NuTj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} P^Fu                          Nu{j2XP42g!E"Xg1E"j X4
$P^$ PYg3M"g;M ~Wg!E g1] ,~4}Ph  DXD,Fg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                    Nugj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} ,~,yP^
$ g3E u*j X4 P^Fu                     NuYj X,XP$ g!E Xg1E j X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^Fu                            NuXj X4 P^P_g3u"!<
g3} 1<^,~4|Ph  DXD,TPj X4 g!E&Xg1E&g+u&u                      Nugj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} 48P48g!E"Xg1E"j X4 P^Fu
             Nulj X41P^$ PYg3M g;M"~Wg!E g1] ,~4}Ph  DXD,Jg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                      Nugj X4 g!E g1] ,~4}Ph  DXD,Jg
)E j X4 P_g3} 3=g1} ,~,qP^$ g3E u*j X4 P^Fu                   Nu\j X4 g!E g1] ,~4}Ph  DXD,Ng)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu
          NuTj X4 g!E"g1]",~4}Ph  DXD,Ng)E"j X4 P_g3}"3=g1}"P^Fu                    Nuej X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P^P_g3u !<g3}"1<@@PLXD$@
$ 4>P^4Uq                          Nugj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} 40P40g!E"Xg1E"j X4 P^Fu                           Nuej X4 g3E"
g)E j X4 g!E"g1]",~4}Ph  DXD,Lg)E"j X4 P^P_g3u"!<g3} 1<P^Fu                   Nurj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DX
D,Dg)E"j X4 P^Fu                      NuSj X4 P_g3}"3=g1}"P_g3}"3=g1}"@PHg!E$Xg1E$j X4 P^Fu                         Nuh%@@%  Hg3E$@g)E"j X4 P
^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Lg)E j X4 P^Fu                  Nuyj X4 P_g3} 3=g1} ,8P$ g!E$Xg1E$CCCCCCCCCCCCCCCCCCCCj X4 g!E"g1]",~4}Ph  DXD,B
g)E"j X4 P^Fu                   NuXj X4 P^P_g3u"!<g3}$1<g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^Fu                            NuVj X4 P^P_g3u"!<g3} 1
<@Ph  DXD@@P%@ g!E Xg1E j X4 P^Fu                      NuDj X4 g3E PEEEEEEj X4!Ph  DXD,cP^4Uq                          NuZMMMMMMj X4 g!E"g1]",
~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"P^Fu                    NuZKKKKKKKKKKKKKKKKKKKKj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P^Fu
   NuXj X4 P^P_g3u !<g3}"1<^,~4|Ph  DXD,BPj X4 g!E&Xg1E&g+u&u                     Nuuj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4!PHg!E"Xg1E"j X4 P^P_g3u !<g3
}"1<@@PLXD$@$ 46P^4Uq                    Nuuj X4 g!E g1] ,~4}Ph  DXD,Ng)E j X4!PHg!E"Xg1E"j X4 P^P_g3u !<g3}"1<@@PLXD$@$ 47P^4Uq
            Nu\j X4 g!E g1] ,~4}Ph  DXD,Tg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                     NuTj X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P_g3}"3=g1}"P
^Fu                         Nu'j X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P_g3} 3=g1} P^Fu                        Nuj%@@%  Hg3E @g)E"j X4 P
_PZg3}"25g!E"g0u"g!E g1] ,~4}Ph  DXD,Lg)E j X4 P^Fu                    NuXj X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Lg)E j X4 P^Fu
            Nuej X4 P_g3} 3=g1} 4&P^4&g3E tHj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^Fu                        Nu}j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,D
g)E"j X4 P_g3}"3=g1}"P_g3}"3=g1}"@PHg!E*Xg1E*j X4 P^Fu                  Nut%@@%  Hg3E*@g)E"j X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g
3} 3=g1} P^Fu                      Nu|j^XP$ g!E"Xg1E"j XHP^$ PYg3M g;M"tY4_P$ g!E"Xg1E"j!XP^4!PYg3M g;M"t84=P4=g!E"Xg1E"j X4 P^Fu
           NuMj X4=P^$ PYg3M g;M"t84zP4zg!E"Xg1E"j X4 P^Fu                        Nudj X43P^$ PYg3M g;M"tOg!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g3} 3=g1} P
^Fu                         Nu{jAXP$ g!E"Xg1E"j X4&P^$ PYg3M"g;M ~Wg!E g1] ,~4}Ph  DXD,Hg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu
  NuTj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} P^Fu                    Nubj XHP^$ g3E uTj X4 g!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g3} 3=g1} P^Fu
                         Nu{jaXP4ag!E"Xg1E"j X4)P^$ PYg3M"g;M ~Wg!E g1] ,~4}Ph  DXD,Jg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                          Nuej
 X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} 40P^40g3E u*j X4 P^Fu                        Nu5j X4"PLXD$@$ 47P^4Uq                                Nugj
 X4 g!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g3} 3=g1} 4GP$ g!E"Xg1E"j X4 P^Fu                    NujjBXP^$ PYg3M g;M"~Wg!E g1] ,~4}Ph  DXD,Ng)E j X4 g!E"P^
P_g3u !<g3}"1<P^Fu                     Nuej X4 g!E g1] ,~4}Ph  DXD,Ng)E j X4 P_g3} 3=g1} 4CP^$ g3E u*j X4 P^Fu                               Nu\j
X4 g!E g1] ,~4}Ph  DXD,Vg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                      NuTj X4 g!E"g1]",~4}Ph  DXD,Vg)E"j X4 P_g3}"3=g1}"P^Fu
              Nuej X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P^P_g3u !<g3}"1<@@PLXD$@$ 4<P^4Uq                         Nugj X4 g!E g1] ,~4}Ph  DXD,Lg)E j X4 P
_g3} 3=g1} 4gP4gg!E"Xg1E"j X4 P^Fu                    Nujj=XP^4=PYg3M g;M"~Wg!E g1] ,~4}Ph  DXD,Pg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu
               Nuej X4 g!E g1] ,~4}Ph  DXD,Pg)E j X4 P_g3} 3=g1} 4>P^4>g3E u*j X4 P^Fu                     Nu\j X4 g!E g1] ,~4}Ph  DXD,Xg)E j X4 g!
E"P^P_g3u !<g3}"1<P^Fu                  NuTj X4 g!E"g1]",~4}Ph  DXD,Xg)E"j X4 P_g3}"3=g1}"P^Fu                         Nuej X4 g!E g1] ,~
4}Ph  DXD,Jg)E j X4 P^P_g3u !<g3}"1<@@PLXD$@$ 49P^4Uq                    Nu~j X4 g!E$g1]$,~4}Ph  DXD,Fg)E$j X4 P_g3}$3=g1}$g!E"g1]",~4}Ph  DXD,Bg)E
"j X4 P_g3}"3=g1}"P^Fu                  NuTj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P^Fu                       NudCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCj X4 g!E*g1]*,~4}Ph  DXD,Bg)E*j X4 P^Fu                   NuXj X4 P^P_g3u*!<g3}*1<g!E*g1]*,~4}Ph  DXD,Dg)E*j X4 P^Fu
               NuVj X4 P^P_g3u*!<g3} 1<@Ph  DXD4jP%  g!E Xg1E j X4 P^Fu                          NuLj X4 g3E PEEEEEEEEEEEEEEj X4"Ph  DXD4(P^4Uq
                   Nu{MMMMMMMMMMMMMMMj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P_g3} 3=g1} ,~,fP$ g!E.Xg1E.g+].j X4 P^Fu                              NuOj X4 P^
P_g3u$!<g3} 1<,~HHP$ g!E Xg1E j X4 P^Fu                   Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXD4&Pj X4 g!E.Xg1E.g+u.u
                         Nu5j X4"PLXD$@$ 40P^4Uq                                Nuej X4 g!E g1] ,~4}Ph  DXD,Tg)E j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu
                         Nuu%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Tg)E"j X4 P^P_g3u"!<g3} 1<@@PLXD$@$ 4.P^4Uq                                 Nuej X4
 g!E g1] ,~4}Ph  DXD,Tg)E j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu                        Nurj X4 g3E"g)E j X4 g!E"g1]",~4}Ph  DXD,Tg)E"j X4 P^P_g3u"!<g3
} 1<@@PLXD$@$ 4,P^4Uq                    Nuzj X4 g!E g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXD4,Pj X4 g!E.Xg1E.g+u.u
               Nu~j X4 g!E*g1]*,~4}Ph  DXD,Bg)E*j X4 P_g3}*3=g1}*g!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g3} 3=g1} P^Fu                           NuTj X4 g!E"
```

```
g1]",~4}Ph  DXD,Dg)E"j X4 P_g3}"3=g1}"P^Fu                              NutCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCj X4 g!E,g1],,~4}Ph  DXD,Bg)E,j X4 P^P_g3u,!<g3}*1<P^F
u                              NuXj X4 g!E,g1],,~4}Ph  DXD,Dg)E,j X4 P^P_g3u,!<g3} 1<P^Fu                                        NuXj X4 g!E,g1],,~4}Ph  DXD,Fg)E,j
X4 P^P_g3u,!<g3}"1<P^Fu                              Nugj X4!Ph  DXD4yP%  g!E Xg1E j X4 g3E PEEEEEEEEEEEEEEj X4!Ph  DXD,:P^4Uq
  NubMMMMMMMMMMMMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"P^Fu                              Nuij X4<P$ g!E.Xg1E.g+].j X4 g!E g1] ,~4}Ph  DXD,Tg)E j X
 4 P_g3} 3=g1} P^Fu                       NuX%@@%  Hg3E @g)E"j X4 g!E g1] ,~4}Ph  DXD,Rg)E j X4 P^Fu                                      NuXj X4 P^P_g3u !<
g3}"1<g!E&g1]&,~4}Ph  DXD,Fg)E&j X4 P^Fu                              NuTj X4 P_g3}&3=g1}&g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^Fu
Nu'j X4 P_g3}"3=g1}"g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P^Fu                              NutCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCj X4 g!E*g1]*,~4}Ph  DXD,Bg)E*
j X4 P^P_g3u*!<g3}"1<P^Fu                              Nuvj X4 g!E*g1]*,~4}Ph  DXD,Dg)E*j X4 P^P_g3u*!<g3} 1<@Ph  DXD4BPj X4 g!E Xg1E j X4 P^Fu
                    NuLj X4 g3E PEEEEEEEEEEEEEEj X4!Ph  DXD,0P^4Uq                              NuVMMMMMMMMMMMMMMMj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P^Fu
                         Nu_j X4 P_g3} 3=g1} ,~,fP$ g!E.Xg1E.g+].j X4 P^P_g3u&!<g3} 1<P^Fu                              NuTj X4 g!E g1] ,~4}Ph  DXD,Rg)E j
X4 P_g3} 3=g1} P^Fu                       Nuuj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXDHPj X4 g!E.Xg1E.g+u.u
             NuTj X4 g!E*g1]*,~4}Ph  DXD,Bg)E*j X4 P_g3}*3=g1}*P^Fu                              Nugj X4 g!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g3} 3=g1} 4 P4 g!E"Xg
1E"j X4 P^Fu                              Nu}j X4 g3E"g)E j X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P_g3}"3=g1}"CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCP^Fu
               NuXj X4 g!E,g1],,~4}Ph  DXD,Bg)E,j X4 P^P_g3u,!<g3}*1<P^Fu                              Nuvj X4 g!E,g1],,~4}Ph  DXD,Dg)E,j X4 P^P_g3u,!<g3} 1<g!
E,g1],,~4}Ph  DXD,Fg)E,j X4 P^Fu                              Nuzj X4 P^P_g3u,!<g3}"1<@Ph  DXD4LPj X4 g!E Xg1E j X4 g3E PEEEEEEEEEEEEEEj X4!Ph  DXD,MP^4Uq
                              NubMMMMMMMMMMMMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"P^Fu                              Nu'j X4<P$ g!E.Xg1E.g+].j X4,
P$ g!E Xg1E %@@%  Hg3E @g)E"j X4 P^Fu                              NuTj X4 g!E g1] ,~4}Ph  DXD,Tg)E j X4 P_g3} 3=g1} P^Fu                              NuX
%@@%  Hg3E @g)E"j X4 g!E g1] ,~4}Ph  DXD,Rg)E j X4 P^Fu                              NuXj X4 P^P_g3u !<g3}"1<g!E(g1](,~4}Ph  DXD,Fg)E(j X4 P^Fu
                    Nu'j X4 P_g3}(3=g1}(g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P_g3}"3=g1}"P^Fu                              NuTj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P_g3
} 3=g1} P^Fu                              NudCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCj X4 g!E*g1]*,~4}Ph  DXD,Bg)E*j X4 P^Fu                              NuXj X4 P^P_
g3u*!<g3}"1<g!E*g1]*,~4}Ph  DXD,Dg)E*j X4 P^Fu                              NuXj X4 P^P_g3u*!<g3} 1<@Ph  DXD4WPj X4 g!E Xg1E j X4 P^Fu
           NuLj X4 g3E PEEEEEEEEEEEEEEj X4!Ph  DXD,EP^4Uq                              NuVMMMMMMMMMMMMMMMj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P^Fu
                    Nu_j X4 P_g3} 3=g1} ,~,fP$ g!E.Xg1E.g+].j X4 P^P_g3u(!<g3} 1<P^Fu                              NuTj X4 g!E g1] ,~4}Ph  DXD,Rg)E j X4 P_g3}
3=g1} P^Fu                              Nuwj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}PLXD$@$ 44Pj X4 g!E.Xg1E.g+u.u
     Nutj X4 g!E g1] ,~4}Ph  DXD,Ng)E j X4!PHg!E"Xg1E"j X4 P^P_g3u !<g3}"1<@@Ph  DXD4%P^4Uq                              Nutj X4 g!E g1] ,~4}Ph  DXD,Pg)E j X
4!PHg!E"Xg1E"j X4 P^P_g3u !<g3}"1<@@Ph  DXD4*P^4Uq                              Nutj X4 g!E g1] ,~4}Ph  DXD,Vg)E j X4!PHg!E"Xg1E"j X4 P^P_g3u !<g3}"1<@@Ph  D
XD4%P^4Uq                              Nutj X4 g!E g1] ,~4}Ph  DXD,Xg)E j X4!PHg!E"Xg1E"j X4 P^P_g3u !<g3}"1<@@Ph  DXD4*P^4Uq
 Nugj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} 4 P4 g!E"Xg1E"j X4 P^Fu                              Nun%@@%  Hg3E @g)E"j X4!PHg!E$Xg1E$CCCCCCj X4 g!E g1]
 ,~4}Ph  DXD,Bg)E j X4 P^Fu                              NuXj X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^Fu                              NuWj X4 P
^P_g3u !<g3}$1<@PLXD$@$ ,|P%  g!E Xg1E j X4 P^Fu                              NuDj X4 g3E PEEEEEEj X4!Ph  DXD4BP^4Uq                              Nu'MMMMM
MKKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              Nu'j@XP$ g!E"Xg1E"%@@%  Hg3E @g)E"j X40P$ g!E$Xg1E$CCCCCCj X4 P^Fu
                              NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P
^P_g3u !<g3}$1<P^Fu                              Nu'j X4!PLXD$@$ ,wP%  g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4}P^4Uq                              Nu'MMMMMM
KKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              Nubj'XP4'g!E"Xg1E"%@@%  Hg3E @g)E"j X,0P,~4rg!E$Xg1E$CCCCCCj X4 P^Fu
                              NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4
P^P_g3u !<g3}$1<P^Fu                              Nu'j X4!PLXD$@$ ,rP%  g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4xP^4Uq                              Nu'MMMMM
MKKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              Nu'j~X@@P$ g!E"Xg1E"%@@%  Hg3E @g)E"jwXP4wg!E$Xg1E$CCCCCCj X4 P^Fu
                              NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P
^P_g3u !<g3}$1<P^Fu                              Nu'j X4!PLXD$@$ ,mP%  g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4sP^4Uq                              Nu'MMMMMM
KKKKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              Nu_j#XP4#g!E"Xg1E"%@@%  Hg3E @g)E"j X4!PHg!E$Xg1E$CCCCCCj X4 P^Fu
                              NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P
_g3u !<g3}$1<P^Fu                              Nu'j X4!PLXD$@$ ,hP%  g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4nP^4Uq                              Nu'MMMMMMMKK
KKKKj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              NuSjCXP$ g!E"Xg1E"%@@%  Hg3E @g)E"j X4 g!E$CCCCCCP^Fu
               NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}$1<P
^Fu                              Nu'j X4!PLXD$@$ ,cP%  g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4iP^4Uq                              Nu'MMMMMMKKKKKKj X4 g!E g
1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              NubjcXP4cg!E"Xg1E"%@@%  Hg3E @g)E"j X,0P,~4rg!E$Xg1E$CCCCCCj X4 P^Fu
               NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}$1<
P^Fu                              Nu_j X4!Ph  DXD,~P%@ g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4dP^4Uq                              Nu'MMMMMMKKKKKKj X4 g!E g
1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              Nubj X,~4!P$ g!E"Xg1E"%@@%  Hg3E @g)E"jwXP4wg!E$Xg1E$CCCCCCj X4 P^Fu
               NuXj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^P_g3u !<g3}"1<P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Dg)E j X4 P^P_g3u !<g3}$1<
P^Fu                              Nu'j X4!Ph  DXD,yP%@ g!E Xg1E j X4 g3E PEEEEEEj X4!PLXD$@$ 4?P^4Uq                              NuPMMMMMMKKKKKK^j X,!Ph
 DXD4APj X4 g!E&Xg1E&g+u&u                              Nutj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} ,~,zPLXD$@$ 4'P%@ g!E"Xg1E"j X4 P^Fu
                    Nu^CCCCCCj X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^P_g3u$!<g3} 1<P^Fu                              Nutj X4 g!E$g1]$,~4}Ph  DXD,Dg)E$j X4 P^P_g3u
$!<g3}"1<@Ph  DXD,tP%@ g!E Xg1E j X4 P^Fu                              NuDj X4 g3E PEEEEEEj X4!Ph  DXD4VP^4Uq                              Nu'MMMMMMj X4 g
!E g1] ,~4}Ph  DXD,@g)E j X4 P_g3} 3=g1} KKKKKKP^Fu                              NuWj X42P^$ g3E uHj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^Fu
          NuUj X4 P_g3} 3=g1} ,~,zPh  DXD4>P%  g!E"Xg1E"j X4 P^Fu                              NuNCCCCCCj X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^Fu
                         Nuhj X4 P^P_g3u$!<g3} 1<g!E$g1]$,~4}Ph  DXD,Dg)E$j X4 P^P_g3u$!<g3}"1<P^Fu                              Nu_j X4!Ph  DXD,nP%@ g!E Xg1E
j X4 g3E PEEEEEEj X4!Ph  DXD4PP^4Uq                              Nu'MMMMMMj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P_g3} 3=g1} KKKKKKP^Fu
      NuWj X4.P^$ g3E uHj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P^Fu                              NuUj X4 P_g3} 3=g1} ,~,zPh  DXD44P%  g!E"Xg1E"j X4 P^Fu
                         NuNCCCCCCj X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^Fu                              Nuhj X4 P^P_g3u$!<g3} 1<g!E$g1]$,~4}Ph  DXD,Dg)E$j X4
P^P_g3u$!<g3}"1<P^Fu                              Nu_j X4!Ph  DXD,hP%@ g!E Xg1E j X4 g3E PEEEEEEj X4!Ph  DXD4JP^4Uq                              NuZMMMMMM
j X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P_g3} 3=g1} P^Fu                              NuiKKKKKKj X4*P^$ g3E uTj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu
                              NuzCCCCCCj X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^P_g3u"!<g3} 1<@Ph  DXD,dP%@ g!E Xg1E j X4 P^Fu                              NuD
j X4 g3E PEEEEEEj X4!Ph  DXD4~P^4Uq                              NudMMMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"KKKKKKg!E P^Fu
          Nufj X4(P^$ PYg3M g;M"|Q,~,zPLXD$@$ 47P%@ g!E Xg1E j X4 P_g3} 3=g1} P^Fu                              Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3
u"!<g3} 1<^,~4}Ph  DXD4XPj X4 g!E&Xg1E&g+u&u                              NuRj X4(PLXD$@$ 49P%@ g!E Xg1E j X4 P_g3} 3=g1} P^Fu
  Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXD4ZPj X4 g!E&Xg1E&g+u&u                              NuRj X4(PLXD$@$ 4;P%@ g!E Xg1E j X4 P
_g3} 3=g1} P^Fu                              Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXD4\Pj X4 g!E&Xg1E&g+u&u
          NuRj X4(PLXD$@$ 4=P%@ g!E Xg1E j X4 P_g3} 3=g1} P^Fu                              Nuvj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXD
4^Pj X4 g!E&Xg1E&g+u&u                              NuTj X4 g!E g1] ,~4}Ph  DXD,Bg)E j X4 P_g3} 3=g1} P^Fu                              Nuwj X4 g!E"g1]",~
4}Ph  DXD,@g)E"j X4 P^P_g3u"!<g3} 1<^,~4}Ph  DXD4_@Pj X4 g!E&Xg1E&g+u&u                              NuLj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 g!E"P^Fu
                         Nudj X4 P^P_g3u !<g3}"1<g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P_g3}"3=g1}"P^Fu                              NuXj X4 g!E g1] ,~4}Ph  DXD,Ng)E j
 X4 P^P_g3u !<g3}"1<P^Fu                              Nu\j X4 g!E g1] ,~4}Ph  DXD,Lg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                              Nurj X4
g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P_g3}"3=g1}"g!E g1] ,~4}Ph  DXD,Lg)E j X4 P^Fu                              NuWj X4 P_g3} 3=g1} Hg3E @g)E"j X4 P_PZg3}"25g!E"g0
u"P^Fu                              Nuvj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Fg)E j X4 P^Fu
Nukj X4 P_g3} 3=g1} 4|P4|g!E"Xg1E"j X4<P^$ PYg3M g;M"t7@@@P^$ g3E t*j X4 P^Fu                              Nuej X4 g!E g1] ,~4}Ph  DXD,Lg)E j X4 P_g3} 3=g1}
@PHg!E"Xg1E"j X4 P^Fu                              Nut%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Lg)E"j X4 P^P_g3u"!<g3} 1<@@Ph  DXD4aP^4Uq
          Nurj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P_g3}"3=g1}"g!E g1] ,~4}Ph  DXD,Jg)E j X4 P^Fu                              Nu{j X4 P_g3} 3=g1} ,~,zP$ g!E
&Xg1E&CCCCCCCCCCCCCCCCCCCj X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^Fu                              NuXj X4 P^P_g3u$!<g3}&1<g!E$g1]$,~4}Ph  DXD,Dg)E$j X4 P^Fu
                              NuVj X4 P^P_g3u$!<g3} 1<@Ph  DXD,JP%  g!E Xg1E j X4 P^Fu                              NuFj X4 g3E PEEEEEEEEj X4!Ph  DXD4)P^4Uq
                              NunMMMMMMMMMj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P_g3} 3=g1} KKKKKKKKKKKKKKKKKKKP^Fu                              NuKj X4 g!E$g3E"g1
E$%@@%  Hg3E @g)E$j X4 P^Fu                              Nuvj X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 P^P_g3u !<g3}$1<g!E g1] ,~4}Ph  DXD,Hg)E j X4 P^Fu
                    NuTj X4 P_g3} 3=g1} g!E$g1]$,~4}Ph  DXD,Ng)E$j X4 P^Fu                              Nupj X4 P_g3}$3=g1}$P^P_g3u !<g3}$1<g!E g1] ,~4}Ph
DXD,Hg)E j X4 P_g3} 3=g1} P^Fu                              NuKj X4$P$ g!E$Xg1E$%@@%  Hg3E$@g)E j X4 P^Fu                              Nuuj~X@@P$ g!E$Xg1E
$j X4 P^P_g3u !<g3}$1<g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} P^Fu                              NuKj X4&P$ g!E$Xg1E$%@@%  Hg3E$@g)E j X4 P^Fu
                    Nuhj X4 g!E$P^P_g3u !<g3}$1<g!E g1] ,~4}Ph  DXD,Hg)E j X4 P_g3} 3=g1} P^Fu                              NuKj X4"P$ g!E$Xg1E$%@@%  Hg3E$@g)
E j X4 P^Fu                              Nu>j X4 g!E$P^P_g3u !<g3}$1<P^Fu                              Nurj X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P_g3}$3=g1}$g
!E g1] ,~4}Ph  DXD,Lg)E j X4 P^Fu                              NuIj X4 P_g3} 3=g1} Hg3E @g)E$j X4 g!E P^Fu                              Nuuj X4 P_PZg3}$ 5
g2u 05g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} @PHg!E$Xg1E$j X4 P^Fu                              NuX%@@%  Hg3E$@g)E j X4 g!E$g1]$,~4}Ph  DXD,Jg)E$j X4 P^Fu
                    NuXj X4 P^P_g3u$!<g3} 1<g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^Fu                              Nuqj X4 P_g3}$3=g1}$g!E g1] ,~4}Ph
DXD,Lg)E j X4 P_g3} 3=g1} @PHg!E&Xg1E&j X4 P^Fu                              Nuh%@@%  Hg3E&@g)E %@@%  Hg3E @g)E$j X4 g!E g1] ,~4}Ph  DXD,Ng)E j X4 P^Fu
                              NuXj X4 P^P_g3u !<g3}$1<g!E g1] ,~4}Ph  DXD,Fg)E j X4 P^Fu                              NuRj X4 P_g3} 3=g1} @@P^$ g3E t5j X4"PLXD$
@$ 4,P^4Uq                              Nu5j X4"PLXD$@$ 4-P^4Uq                              NuTj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} P^Fu
                              Nuvj X4 g!E$g1]$,~4}Ph  DXD,@g)E$j X4 P^P_g3u$!<g3} 1<^,~4}Ph  DXD,zPj X4 g!E(Xg1E(g+u(u                              Nu'j X4 g!E g
1] ,~4}Ph  DXD,Dg)E j X4 P_g3} 3=g1} P_g3} 3=g1} P^Fu                              NuXj X4 g!E"g1]",~4}Ph  DXD,dg)E"j X4 P^P_g3u"!<g3} 1<P^Fu
                    Nu]j X4 g!E g1] ,~4}Ph  DXD,dg)E 4pPj X4 g!E(Xg1E(g+](j X4 P^Fu                              Nuvj X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^P_g3u"!
<g3} 1<@Ph  DXD,.Pj X4 g!E Xg1E j X4 P^Fu                              NuGj X4 g3E PEEEEEEEEj X4!PLXD$@$ 4?P^4Uq                              NucMMMMMMMMMj
4XP44g!E(Xg1E(g+](j X4 g!E g1] ,~4}Ph  DXD,dg)E j X4 P^Fu                              Nuyj X4 P_g3} 3=g1} ,~4}Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4 g!E"g1]",~4}P
h  DXD,Bg)E"j X4 P^Fu                              Nutj X4 P^P_g3u"!<g3} 1<@Ph  DXD,+Pj X4 g!E Xg1E j X4 g3E PEEEEEEEEj X4!Ph  DXD,&P^4Uq
          NuPMMMMMMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P^Fu                              Nukj X4 P_g3}"3=g1}"44P44g!E(Xg1E(g+](j X4 g!E g1] ,~4}Ph  DXD,n
g)E j X4 P^Fu                              NuXj X4 P^P_g3u !<g3}"1<g!E"g1]",~4}Ph  DXD,ng)E"j X4 P^Fu                              NuTj X4 P_g3}"3=g1}"g!E
 g1] ,~4}Ph  DXD,Lg)E j X4 P^Fu                              Nuij X,!Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4 g!E$g1]$,~4}Ph  DXD,Bg)E$j X4 P^Fu
             Nuhj X4 P^P_g3u$!<g3}"1<g!E$g1]$,~4}Ph  DXD,Dg)E$j X4 P^P_g3u$!<g3} 1<P^Fu                              Nucj X4!Ph  DXD,$Pj X4 g!E Xg1E j X4 g3E
PEEEEEEEEEj X4"Ph  DXD42P^4Uq                              Nu\MMMMMMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"P^Fu                              Nukj
4XP44g!E(Xg1E(g+](j X4 g!E g1] ,~4}Ph  DXD,lg)E j X4 P^P_g3u !<g3}"1<P^Fu                              Nuoj X,!Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4"PLXD$@$ P%  g
!E Xg1E j X4 g3E Pj X4 P^Fu                              Nu<EEEEEEEEj X4!Ph  DXD4&P^4Uq                              NuwMMMMMMMMMj4XP44g!E(Xg1E(g+](j X4 g!
E g1] ,~4}Ph  DXD,fg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                              Nurj X4 g!E g1] ,~4}Ph  DXD,fg)E j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,lg)E"
j X4 P^Fu                              Nutj X4 P_g3}"3=g1}",~,{P^$ PYg3M"g;M ~Og!E g1] ,~4}Ph  DXD,fg)E j X4 P_g3} 3=g1} P^Fu
 Nuij X,!Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4 g!E"g1]",~4}Ph  DXD,Bg)E"j X4 P^Fu                              NuWj X4 P^P_g3u"!<g3} 1<@@Ph  DXD4&P%  g!E Xg1E j X
4 P^Fu                              NuFj X4 g3E PEEEEEEEEj X4!Ph  DXD,'P^4Uq                              NuEMMMMMMMMMj4XP44g!E(Xg1E(g+](j X4 P^Fu
                         Nuej X4 g!E g1] ,~4}Ph  DXD,fg)E j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu                              Nuu%@@%  Hg3E"@g)E j X4 g!E"g1]"
,~4}Ph  DXD,fg)E"j X4 P^P_g3u"!<g3} 1<@@PLXD$@$ 4?P^4Uq                              Nuhj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4!PHg!E"Xg1E"j X4 P^P_g3u !<g3}"1<P^
Fu                              Nu\j X4 g!E g1] ,~4}Ph  DXD,Hg)E j X4 g!E"P^P_g3u !<g3}"1<P^Fu                              Nu~j X4 g!E g1] ,~4}Ph  DXD,Hg
)E j X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,lg)E"j X4 P_g3}"3=g1}"P^Fu                              NuXj X4.P^$ PYg3M"g;M ~Cg!E g1] ,~4}Ph  DXD,Hg)E j X4 P^Fu
                              Nulj X4 P_g3} 3=g1} ,~,zP$ g!E$Xg1E$j X,!Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4 P^Fu                              NuXj X4 g!E"g1]",~4
}Ph  DXD,Bg)E"j X4 P^P_g3u"!<g3}$1<P^Fu                              NuXj X4 g!E"g1]",~4}Ph  DXD,Dg)E"j X4 P^P_g3u"!<g3} 1<P^Fu
   NuXj X4"Ph  DXD41P%  g!E Xg1E j X4 g3E PEEEEEEEEj X,RP^4Uq                              NuPMMMMMMMMMj X4 g!E g1] ,~4}Ph  DXD,@g)E j X4 P^Fu
                    Nukj X4 P_g3} 3=g1} 44P44g!E(Xg1E(g+](j X4 g!E"g1]",~4}Ph  DXD,Lg)E"j X4 P^Fu                              NuX%@@%  Hg3E @g)E"j X4 g!E g1] ,
~4}Ph  DXD,Jg)E j X4 P^Fu                              Nudj X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} P^Fu
 NuKj X4$P$ g!E"Xg1E"%@@%  Hg3E"@g)E j X4 P^Fu                              Nudj X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,hg)E"j X4 P^P_g3u"!<g3} 1<P^Fu
                         Nugj X4 g!E g1] ,~4}Ph  DXD,hg)E j X4 P_g3} 3=g1} ,~,|P^$ g3E u*j X4 P^Fu                              Nuej X4 g!E g1] ,~4}Ph  DXD,Hg)E
 j X4 P_g3} 3=g1} @PHg!E"Xg1E"j X4 P^Fu                              Nuu%@@%  Hg3E"@g)E j X4 g!E"g1]",~4}Ph  DXD,Hg)E"j X4 P^P_g3u"!<g3} 1<@@PLXD$@$ 48P^4Uq
                              Nupj X4 g!E g1] ,~4}Ph  DXD,Fg)E j X4 P_g3} 3=g1} 4#P^4#g3E u5j X4"PLXD$@$ 45P^4Uq                              Nu4j X4"Ph  D
XD4cP^4Uq                              Nu4j X4"Ph  DXD4eP^4Uq                              Nuyj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} ,~,|P$ g!E"Xg
1E"%@@%  Hg3E"@g)E j X4 P^Fu                              NuTj X4 P_g3} 3=g1} g!E"g1]",~4}Ph  DXD,pg)E"j X4 P^Fu                              Nu\j X4 P^P_
g3u"!<g3} 1<g!E g1] ,~4}Ph  DXD,Fg)E j X4 g!E"P^Fu                              Nudj X4 P^P_g3u !<g3}"1<g!E g1] ,~4}Ph  DXD,pg)E j X4 P_g3} 3=g1} P^Fu
                              Nu'j X4 g!E",~,nP^$ PYg3M"g;M |Cg!E g1] ,~4}Ph  DXD,Jg)E j X4 P^Fu                              Nulj X4 P_g3} 3=g1} P_g3} 3=g1} g!E
"g1]",~4}Ph  DXD,Jg)E"j X4 P_g3}"3=g1}"P^Fu                              Nuij X4"P$ g!E$Xg1E$%@@%  Hg3E$@g)E"j X4 g!E$g1]$,~4}Ph  DXD,Jg)E$j X4 P^Fu
```

```
                         Nu[j X4 P_g3}$3=g1}$,~,|P$ g!E&Xg1E&%@@%  Hg3E&@g)E$j X4 P^Fu                                 Nuyj X,!Ph  DXD,TPj X4 g!E(Xg1E(g+](j X4 g
!E&g1]&,~4}Ph  DXD,Bg)E&j X4 P^P_g3u&!<g3} 1<P^Fu                              Nuvj X4 g!E&g1]&,~4}Ph  DXD,Dg)E&j X4 P^P_g3u&!<g3}"1<g!E&g1]&,~4}Ph  DXD,Fg)E
&j X4 P^Fu                                     Nutj X4 P^P_g3u&!<g3}$1<@@PLXD$@$ 4(P%@ g!E Xg1E j X4 g3E PEEEEEEEEj X4!Ph  DXD4fP^4Uq
  NusMMMMMMMMj X4 g!E"g1]",~4}Ph  DXD,@g)E"j X4 P_g3}"3=g1}"44P44g!E(Xg1E(g+](j X4 P^Fu                                 NuXj X4 g!E g1] ,~4}Ph  DXD,jg)E j X4 P^
P_g3u !<g3}"1<P^Fu                                     NuTj X4 g!E g1] ,~4}Ph  DXD,Jg)E j X4 P_g3} 3=g1} P^Fu                                  NuKj X4$P$ g!E"Xg1E"%@
@%  Hg3E"@g)E j X4 P^Fu                                     Nudj X4 g!E"g1]",~4}Ph  DXD,jg)E"j X4 P_g3}"3=g1}"P^P_g3u !<g3}"1<P^Fu                                  4
w4wq;                                                           <-- That was the end of the code segment, where we overflow the instruction pointer past 0xFFFF.
```

Also, remember when we talked about the relocation table, and how it has to corrupt some pair of bytes in our file? That's right here: --> XX <--.
If you load this program in a debugger and look at memory approximately starting at CS:FFFF, you'll see the XX changed to something else (unpredictable).

~@~

** 28. PAPER.EXE **

Executing this paper in DOS, with an AdLib-compatible sound card (such as the Sound Blaster) configured at 0x388, will play some music. The music to play is specified on the command line, using a subset of a standard text-based music format called ABC [ABC'05]. For example, invoking

```
PAPER.EXE C4C4G4G4A4A4G8G8F4F4E4E4D4D4C8
```

will play a segment of the "Now I know my ABC's" song and then exit. The language supported is as follows:

```
  A-G   Basic notes
  a-g   Same, up one octave
    z   Rest
    ^   (Prefix) Sharp
    _   (Prefix) Flat
    =   (Prefix) Natural - does nothing since key of C is assumed
    '   (Suffix) Up one octave
    ,   (Suffix) Down one octave
  2-8   (Suffix) Set duration of note to this many eighth notes
```

Up to three simultaneous tracks can play, all using the same dumb-sounding organ-like instrument, by separating tracks with |. DOS treats | specially on the command line, so quote the argument, like PAPER.EXE "AA|BB|CC".

Running PAPER.EXE with arguments like "-song" will play a built-in song. Available songs include: "-alphabet", "-plumber", and "-bluehair". There's plenty of space in the data segment for more!

Running PAPER.EXE without any arguments will play a default song.

** 28. Running, debugging **

Speaking of running the program, old-style EXE files no longer run on 64-bit versions of Windows. So if you do not have an old DOS computer around with a sound card, you can run ABC-compiled programs inside an emulator. DOSBox is an excellent choice. It runs on pretty much all platforms (well, it doesn't run on DOS, but on DOS you can just use DOS) and tends to just work. You have to do something like

```
MOUNT C C:\DOWNLOADS\ABC\
```

in order to mount one of your real directories as a "hard drive". To verify that PAPER.EXE is printable with no beeping or funny characters, you could do

```
COPY PAPER.EXE CON
```

to copy it to your console, or COPY PAPER.EXE LPT1 to copy it to your simulated computer's printer (spoiler: It doesn't have one). But why bother? You're reading PAPER.EXE right now!

I used DOSBox frequently during development, and modified its debugger, especially for understanding the header values are actually used. The ABC compiler outputs each of the intermediate languages for a program as it compiles, as well as lightly-commented X86 assembly with address maps back into the code segment, which makes it possible to easily set breakpoints on particular pieces of code. Since compiling other people's software on Windows is a special nightmare, I frequently worked inside a Linux virtual machine (VirtualBox) containing a DOS virtual machine (DOSBox), a surreal scenario that I was tickled to find a practical use for. Let us one day simulate Windows 7 on our iPhones 21 so that we may render this development environment one level deeper.

My modifications to DOSBox are included in the ABC source repository, although they are not necessary to run ABC-compiled programs. When running these programs under DOSBox with the debugger enabled, it will complain about a "weird header" when loading the program (you're tellin' me!) and the debugger will output the error

```
Illegal/Unhandled opcode 63
```

upon exiting (because we do execute an illegal opcode). For cosmetic style points, the local version of DOSBox has been modified to instead output

```
Thank you for playing Wing Commander!
```

** 29. PAPER.C **

This section contains the C source code that was compiled into this paper. It may be interesting to see how the code (e.g. string literals) make their way into the data for the paper. You may also laugh at my many troubles:

- I'm playing music, which has some dependency on timing, but there is no way to get access to the system clock. Instead, I use for loops with built-in constants determined empirically. At least this technique of relying on the CPU's cycle timing for delays was common in the DOS era, so this is, like, a period piece.

- However, since the routine that calculates lengths performs a multiplication, and multiplication of m * n is O(n), the delays are not actually linear.

- You can see the many places where I'm applying explicit casts, either because an implicit coercion is not yet implemented for ABC (I want to do it right, and the rules are a little subtle), because some operation is not yet available at char or long type (I implemented 16-bit first), or for efficiency.

- You can see the reliance on string literals for efficient lookup tables, in keeping with the "printable" theme.

I also still need to fill up 20 pages in this ridiculously small font!

```
/**********************************************************
 *  paper.c, Copyright (c) 2017 Tom Murphy VII Ph.D.
 *  This copyright notice must appear in the compiled
 *  version of this program. Otherwise, please distribute
 *  freely.
 *
 *  Plays music in a simplified ABC notation, given on the
 *  command line, or one of several built-in songs.
 **********************************************************/

int _out8(int, int);

unsigned char *meta_note = "Now this is the part of the data segment "
  "that stores global variables. This is actually a string constant in "
  "the program itself, so you'll see it again when I show you the source "
  "code later. We have almost 64kb of space to store stuff, although "
  "this segment is also used for the stack of local variables and "
  "arguments, and would be used for malloc as well, if it were "
  "implemented. Storing a string like this is basically free, because "
  "everything in it is printable, aside from the terminating \\0 "
  "character. At program startup, non-printable characters are "
  "overwritten by instructions in the code segment. Like, here's one: "
  "--> \xFF <-- It's stored in the data segment as a printable "
  "placeholder.";

// Adlib uses two bytes to do a "note-on", and the notes are specified
// in a somewhat complex way (octave multiplier plus frequency.) These
// tables give the upper and lower byte for each MIDI note. Computed
// by makefreq.sml.
unsigned char *upper = "\x20\x20\x20\x20\x20\x20\x20\x20!!!!!!!!!!!!"
  "\x22\x22\x22\x22\x22\x22\x22######&&&&&&''''''*******+++++"
  ".......//////2222222333336666666677777:::::::;;;;;>>>>>>>"
  "?????????????????\0";
unsigned char *lower = "\xA9\xB3\xBD\xC9\xD5\xE1\xEF\xFD\x0C\x1C-?Qf{"
  "\x91\xA9\xC2\xDD\xFA\x18" "8Y}\xA3\xCB\xF6#R\x85\xBA\xF3\x18"
  "8Y}\xA3\xCB\xF6#R\x85\xBA\xF3\x18" "8Y}\xA3\xCB\xF6#R\x85\xBA"
  "\xF3\x18" "8Y}\xA3\xCB\xF6#R\x85\xBA\xF3\x18" "8Y}\xA3\xCB\xF6#R"
  "\x85\xBA\xF3\x18" "8Y}\xA3\xCB\xF6#R\x85\xBA\xF3\x18" "8Y}\xA3\xCB"
  "\xF6#R\x85\xBA\xF3\x18" "8Y}\xA3\xCB\xF6#R\x85\xBA\xF3\xFF\xFF"
  "\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\0";

unsigned char *default_song =
  "abd'b^f'3^f'3e'6ab^c'ae'3e'3d'^c'b4ab^c'ad'4e'2^c'2b2a2z2a2e'4d'4z"
  "4abd'b^f'3^f'3e'6ab^c'aa'4^c'2d'2^c'b3ab^c'ad'4e'2^c'3ba2z2a2e'2d'2d'4|"
  "A4E4E4A4A4^F4^F4B4B4E4E4A4A4^F4B4A4A4E4E4A4A4^F4^F4B4B4E4E4A4A4^F4B4A2"
  "A2A2A2|A,2^F,2E,6A,4E2^F2E2^F,6B,2A,2B,2A,2^F,2E,2^F,2G,2A,4E2^F2A2^F,6"
  "A,4E2^F2A2E,2E,2^F,2A,4E2^F2E2^F,6B,4^F2A2^F2E,2^F,2G,2A,4E2^F2E2^F,6A,"
  "4E2^F2A2";

unsigned char *alphabet =
  "C4C4G4G4A4A4G8" "F4F4E4E4D4D4C8"
  "G4G4F4F4E4E4D8" "G4G4F4F4E4E4D8"
  "C4C4G4G4A4A4G8" "F4F4E4E4D4D4C8";

unsigned char *plumber =
  "e'e'ze'zc'e'zg'z3g2z2c'z2gz2ez2azbz^aazge'zg'a'2f'g'ze'zc'd'bz2c'z2gz"
  "2ez2azbz^aazge'zg'a'zf'g'ze'zc'd'bz4g'^f'f'^d'ze'z^gac'zac'd'z2g'^f'f'"
  "^d'ze'zc''zc''c''z5g'^f'f'^d'ze'z^gac'zac'd'z2^d'z2d'z2c'|"
  "DDzDzDDzgz3G2z2Gz2Ez2Cz2FzGz^FFzEczef2dezczABGz2Gz2Ez2Cz2FzGz^FFzEczef"
  "zdezczABGz2Cz2Gz2czFz2cczFzCz2Ez2Gczg'zg'g'zGzCz2Gz2czFz2cczFzCz^Gz2^A"
  "z2cz2GGzC";

unsigned char *bluehair =
  "^A8z2F4^G8F3c4^A4F4^A4^G8z8"
  "^A8z2c8z2^c8z2^d8z2f8z2F4F4F4F8";

typedef struct {
  unsigned char *song;
  int idx;
  int midi_note;
  unsigned int ticksleft;
} Channel;

int Adlib(int reg, int value) {
  int i;
  _out8((int)0x0388, (int)reg);
  // We have to wait "12 cycles" after writing the port.
  for (i = 0; i < (int)12; i++) {}
  _out8((int)0x0389, (int)value);
  // And 84 cycles after writing the value. These numbers are
  // probably far too high; recall that a for loop like this
  // has to jump through every rung in the program! (i.e.,
  // A single iteration is linear in the program size.)
  for (i = 0; i < (int)84; i++) {}
  return 0;
}

int PlayNote(int ch, int midi_note) {
  // First turn note off; silence is better than weird "accidentals."
  Adlib((int)0xB0 + ch, 0x00);
  // midi_note = 128 actually accesses the terminating \0 in the
  // above strings, which is what we want to turn off the channel.
  Adlib((int)0xA0 + ch, (int)(lower[midi_note]));
  Adlib((int)0xB0 + ch, (int)(upper[midi_note]));
}

// Zero all the adlib ports, which both silences it and
// initializes it.
int Quiet() {
  int port;

  // Clear the main tones first, so that we don't hear artifacts during
  // the clearing process if a note is playing.
  Adlib((int)0xB0, (int)0x00);
  Adlib((int)0xB1, (int)0x00);
  Adlib((int)0xB2, (int)0x00);
  for (port = (int)0x01; port <= (int)0xF5; port++) {
    Adlib((int)port, (int)0x00);
  }
}

// ABC provides no standard library, so you gotta roll
// your own.
int strlen(unsigned char *s) {
  int len = 0;
  while ((int)*s != (int)0) {
    len++;
    s = (unsigned char *)((int)s + (int)1);
  }
  return len;
}

int streq(unsigned char *a, unsigned char *b) {
  int i;
  for (i = 0; /* in loop */; i++) {
    int ca = a[i], cb = b[i];
    if (ca != cb) return (int)0;
    if (ca == (int)0) return (int)1;
  }
}

// DOS command lines always start with a space, which is annoying.
// Strip that. DOS also terminates the command line with 0x0D, not
// 0x00. This function updates it in place so that we can use normal
// string routines on it.
int MakeArgString(unsigned char **argstring) {
  unsigned char *s = *argstring;
  while (*s == (int)' ') {
    s = (unsigned char *)((int)s + (int)1);
  }
  *argstring = s;

  while ((int)*s != (int)0x0D) {
    s = (unsigned char *)((int)s + (int)1);
  }
  *s = (unsigned char)0;
  return 0;
}

// We pick octave 4 as the base one; this is fairly canonical and
// benefits us since this array is all printable. Note that A4 is
// higher than C4, since octave 4 begins at the note C4. This
// array maps A...G to the corresponding MIDI note.
unsigned char *octave4 =
  "9"  // A = 57
  ";"  // B = 59
  "0"  // C = 48
  "2"  // D = 50
  "4"  // E = 52
  "5"  // F = 53
  "7"; // G = 55
// Parse a character c (must be capital A,B,C,D,E,F,G)
// and interpret any suffixes as well.
int ParseNote(unsigned char *ptr, int c, int *idx) {
  int midi;
  int offset = c - (int)'A';
  int nextc;
  midi = octave4[offset];
  for (;;) {
    nextc = (int)ptr[*idx];
    switch (nextc) {
    case '\'':
      // Up octave.
      midi += (int)12;
      break;
    case ',':
      // Down octave.
      midi -= (int)12;
      break;
    default:
      // Not suffix, so we're done (and don't consume
      // the character.)
      return midi;
    }
    *idx = *idx + (int)1;
  }
}

unsigned int ParseLength(unsigned char *ptr, int *idx) {
  int c = (int)ptr[*idx];
  if (c >= (int)'2' && c <= (int)'8') {
    int m = c - (int)'0';
    *idx = *idx + (int)1;
    return (unsigned int)200 * m;
  }
  return (unsigned int)200;
}

// Parse the song description (ptr) starting at *idx. Updates *idx to
// point after the parsed note. Updates *len to be the length in some
// unspecified for-loop unit. Returns the MIDI note to play next, or 0
// when the song is done.
int GetMidi(unsigned char *ptr, int *idx, unsigned int *len) {
  int c, midi_note;
  int sharpflat = 0;
  for (;;) {
    c = (int)(ptr[*idx]);

    // End of string literal.
    if (c == (int)0) return 0;

    // Advance to next character.
    *idx = *idx + (int)1;

    switch (c) {
    case '^':
      sharpflat++;
      break;
    case '_':
      sharpflat--;
      break;
    case '=':
      // Nothing. We assume key of C, so there are no naturals.
      break;
    case 'z':
      *len = ParseLength(ptr, idx);
      // No sound.
      return 128;

    default:
      if (c >= (int)'A' && c <= (int)'G') {
        midi_note = ParseNote(ptr, c, idx) + sharpflat;
        *len = ParseLength(ptr, idx);
        return midi_note;
      } else if (c >= (int)'a' && c <= (int)'g') {
        midi_note = ParseNote(ptr, c - (int)32, idx) + (int)12 + sharpflat;
        *len = ParseLength(ptr, idx);
        return midi_note;
      }
    }
  }
}

// Adlib has 9 channels, but they are packed in groups of
// three (c1o1, c2o1, c3o1, c1o2, c2o2, c3o2, c4o1, ...).
// So this only works for the first three channels. Not
// too hard to generalize, especially with a table.
int InitInstrument(int ch) {
  // Initialize the Adlib instrument.
  Adlib((int)0x20 + ch, 0x01); // Modulator multiple 1.
  Adlib((int)0x40 + ch, 0x10); // Modulator gain ~ 40db.
  Adlib((int)0x60 + ch, 0xF0); // Modulator attack: quick. Decay: long.
  Adlib((int)0x80 + ch, 0x77); // Modulator sustain: med. Release: med.
  Adlib((int)0x23 + ch, 0x01); // Carrier multiple to 1.
  Adlib((int)0x43 + ch, 0x00); // Carrier at max volume.
  Adlib((int)0x63 + ch, 0xF0); // Carrier attack: quick. Decay: long.
  Adlib((int)0x83 + ch, 0x77); // Carrier sustain: med. release: med.
}

// First test for known songs. After that, if we have a command line,
// use it. Otherwise, use the default song.
unsigned char *GetSong(unsigned char *cmdline) {
  if (streq(cmdline, (unsigned char *)"-alphabet")) {
    return alphabet;
  } else if (streq(cmdline, (unsigned char *)"-plumber")) {
    return plumber;
  } else if (streq(cmdline, (unsigned char *)"-bluehair")) {
    return bluehair;
  } else if (strlen(cmdline) > (int)0) {
    return cmdline;
  } else {
    return default_song;
  }
}

// Note: Doesn't check that the input is within the maximum number of
// channels!
int SplitChannels(unsigned char *song, Channel *channels) {
  int i, current_channel = 0;
  unsigned char *prevsong = song;
  for (i = (int)0; /* in loop */; i++) {
    int c = song[i];
    switch (c) {
    case '|':
    case '\0': {
      Channel *channel = &channels[current_channel];
      channel->song = prevsong;
      // Silence; ready for next note.
      channel->midi_note = (int)128;
      channel->ticksleft = (int)0;
      channel->idx = (int)0;

      song[i] = (unsigned char)'\0';
      current_channel++;
      // Start after the nul-terminator byte.
      prevsong = &song[i + (int)1];
      if (c == (int)0) return current_channel;
    }
    }
  }
}

int main(int argc, unsigned char **argv) {
  Channel channels[3];
  unsigned char *song, *cmdline = *argv;
  int i, num_channels;

  MakeArgString(&cmdline);
  song = GetSong(cmdline);

  // Initialize channels. Note that this will just blow the
  // stack-allocated channels array if there are more than
  // two in the input string!
  num_channels = SplitChannels(song, (Channel *)&channels);

  Quiet();

  for (i = 0; i < num_channels; i++)
    InitInstrument(i);

  for (;;) {
    int ch, all_done = 1;
    // At each tick (whose rate is governed just by the time
    // it takes to do this loop), reduce each channel's ticksleft;
    // if it was (already) zero, load a new note.
    for (ch = 0; ch < num_channels; ch++) {
      Channel *channel = &channels[ch];
      int midi_note = channel->midi_note;
      if (midi_note != (int)0) {
        int ticksleft = channel->ticksleft;
        all_done = 0;
        if (ticksleft > (int)0) {
          channel->ticksleft = ticksleft - (int)1;
        } else {
          int new_note = GetMidi(channel->song, &channel->idx,
                                 &channel->ticksleft);
          channel->midi_note = new_note;
          if (new_note == (int)0) {
            // Quiet the channel -- forever!
            PlayNote(ch, (int)128);
          } else {
            PlayNote(ch, new_note);
          }
        }
      }
    }
    if (all_done) break;
  }

  Quiet();
  return 0;
}
```

** 30. Is this useful for anything? **

No. This is a SIGBOVIK paper. <3

** 31. Future work **

There are many code size optimizations possible, and while nontrivial programs can fit in 64k (such as the one in this paper), larger ones will run up against that boundary quickly. Probably a factor of about 4 can be gained through a few hard but straightforward optimizations. Can we break free of the 64k boundary? Earlier we noted that when execution exceeds CS:0xFFFF, it simply continues to CS:0x00010000 unless a jump is executed across that boundary; this address is pointing to bytes that are part of our program image (this text is there, in fact), so conceivably we could write code here. One significant issue is that interrupts, which are constantly firing, push 16-bit versions of CS and IP onto the stack, and then RETF (return far) to that address. This means that if an interrupt happens while we are executing in this extended address space, we will return to CS:(EIP & 0xFFFF). If we had control over interrupts, this might be a good way to return to the normal 16-bit code segment (i.e., to perform a backwards jump), but as discussed, we do not. We may be able to globally suppress interrupts, like by using our single illegal instruction interrupt during initialization, with the interrupt handler pointing just to code that we control (and never returning from it). This leaves the interrupt flag cleared, as discussed. The computer will be non-functional in many ways, because the operating system will no longer run, but we might still be able to do rudimentary port-based I/O, or build our own non-interrupt-based OS. With interrupts suppressed, we can't use the interrupt trick to return to CS:0000. However, my reading of the Intel manual [INTC] seems to imply that a jump performed from this region can be forced into 16-bit mode (thus being subject to the & 0xFFFF overflow) with an address size prefix; however, this does not seem to be the case in DOSBox. Given how unusual this situation is, it may even be a bug in DOSBox's CPU emulator. Having access to a full megabyte of code (it still needs to fit in the EXE container) would be exciting, since it would allow us to build much more significant systems (e.g. standard malloc and a floating point emulator); more investigation is warranted here.

I initially designed CIL with the thought that it could be used for multiple such "compile C to X" projects. These are primarily jokes, but can occasionally be of legitimate use for low-level domain-specific tasks where the existence of a reasonable and familiar high-level syntax pays for the effort of writing a simple backend. (When making such a decision I like to also weight the effort by the enjoyment of each task: i.e., the cost is like

```
   (1 - fun of writing backend) * time writing backend   vs
   (pain of writing low-level code by hand) *
       time writing low-level code by hand
```

... but I have been informed that not all computer work is done purely for fun.) This "portable assembler" application of C remains relevant today, and CIL or LLVMNOP is a much simpler than GCC or LLVM.

Anyway, I discovered that the design of such a thing is not so easy. While it is possible to "compile away" certain features by turning them into something "simpler," it's not straightforward what feature set to target. For example, for ABC, we compile away the | operator into &, ^, -, and +1. In another setting, | may very well be present instead of &. We normally think of the >> and << shift operators as being fundamental, but in ABC they are inaccessible. I find the expression forms like "a < b" much easier to think about then the combined test-and-branch version, but the latter is much better when targeting x86, and important for producing reasonable code in ABC. I do think it would be possible to develop a simple and general language for this niche where certain constructs could be compiled away in favor of others, at the direction of the compiler author, but such a thing is firmly future work.

On the topic of taking away, one might ask: What is the minimal subset of bytes we could imagine using?

There are some trivial subtractions: We never emit the BOUND instruction (0x62, lowercase b) and it does not seem useful; a few of the segment prefix instructions are also unused. The instructions like "ASCII Adjust After Addition" are currently unused, but since they act on AX in a predictable way, they could provide ways to improve the routines to load immediate values. But we're talking about reducing the surface, not increasing it. And speaking of loading immediate values, we do certainly make use of the entire set of printable bytes in these routines (as arguments to XOR, SUB, PUSH, etc.), but on the other hand, we can also reach any value from a known starting point by INC and DEC, taking at most 0x7FFF instructions (half the size of the code segment, unfortunately). More essential is our ability to set a register to a known value, which today requires two or more printable values whose bitwise AND is 0. Sadly, though we could go through some pains to remove bytes from the gamut here and there, no natural subset like "lowercase letters" or "alphanumeric" jumps out; we rely on the control flow in the late lowercase letters (Jcc) and the basic ops in the early punctuation (AND/XOR), not to mention that the EXE header barely works within the existing constraints with access to both "small" (0x2020) and "large" (0x7e7e) constants.

Others have produced compilers for high-level languages with very reduced instruction sets. In an extreme case, Dolan shows [MOV'13] that the mov instruction on its own is Turing-complete (note however that this requires a "single absolute jump" to the top of the program, an issue similar to what we encounter in printable x86, only we do not cheat by inserting any out-of-gamut instructions). Another enterprising programmer, Domas, implemented a C compiler that produces only MOV instructions [MVF'16]. I didn't look at it while writing ABC (spoilers!) but he avoids using any JMP instruction the same way that I exit the program (generating illegal instructions but rewriting the interrupt handler). While awesome, the problem is somewhat different from what ABC solves; here we are fundamentally concerned with what bytes appear in the executable, which influences what opcodes are accessible (and their arguments and addressing modes), but is not the only constraint created. For example, in MOV-only compilation, the program's header does not need to consist only of MOV instructions, and so the compiler's output does not suffer the same severe code and data limitations that DOS EXEs do. (The executables it produces are extremely large and slow; they also seem to have non-MOV initialization code.) The MOV instruction is also very rich, and no versions of it are printable!

Of course, everyone knows that even unary numbers (just like one symbol repeated a given number of times) is Turing complete, via Godel encoding. So what's the big deal?

Figure 7. Printable X86

** 32. Acknowledgements **

The author would like to thank the fastidious SIBOVIK "Program" Committee for "Evaluating" my paper.

** 33. Bibliography **

[KNPH'14] Tom Murphy VII. "New results in k/n Power-Hours." SIGBOVIK, April 2014.

[MTMC'08] Tom Murphy VII. "Modal Types for Mobile Code." Ph.D. thesis, Carnegie Mellon University, January 2008. Technical report CMU-CS-08-126.

[LLVM'04] Chris Lattner and Vikram Avde. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." CGO, March 2004.

[CKIT'00] David Ladd, Satish Chandra, Michael Siff, Nevin Heintze, Dino Oliva, and Dave MacQueen. "Ckit: A front end for C in SML." March 2000. http://smlnj.org/doc/ckit/

[INTC'01] Intel Corporation. "IA-32 Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference." 2001.

[ABC'05] Steve Mansfield. "How to interpret abc music notation." 2005.

[MOV'13] Stephen Dolan. "mov is Turing-complete". 2013.

[MVF'16] Chris Domas. "M/o/Vfuscator2". August 2015. https://github.com/xoreaxeaxeax/movfuscator

Please see http://tom7.org/abc for supplemental material.

** Appendix **

Here is a histogram of every character that appears in this file. There are no non-printable bytes.

char	byte	number of occurrences
	0x20	179076
!	0x21	1242
"	0x22	1771
#	0x23	3216
$	0x24	557
%	0x25	3922
&	0x26	113
'	0x27	467
(	0x28	8770
)	0x29	955
*	0x2A	528
+	0x2B	233
,	0x2C	2088
-	0x2D	27690
.	0x2E	7339
/	0x2F	252
0	0x30	1045
1	0x31	1618
2	0x32	697
3	0x33	1346
4	0x34	2594
5	0x35	373
6	0x36	321
7	0x37	185
8	0x38	296
9	0x39	102
:	0x3A	1423
;	0x3B	306
<	0x3C	482
=	0x3D	1048
>	0x3E	49
?	0x3F	96
@	0x40	11867
A	0x41	559
B	0x42	366
C	0x43	664
D	0x44	1516
E	0x45	2762
F	0x46	860
G	0x47	200
H	0x48	291
I	0x49	422
J	0x4A	118
K	0x4B	192
L	0x4C	261
M	0x4D	490
N	0x4E	773
O	0x4F	248
P	0x50	2784
Q	0x51	8236
R	0x52	147
S	0x53	375
T	0x54	372
U	0x55	154
V	0x56	81
W	0x57	127
X	0x58	2821
Y	0x59	62
Z	0x5A	143
[	0x5B	105
\	0x5C	132
]	0x5D	459
^	0x5E	946
_	0x5F	23111
`	0x60	38
a	0x61	5342
b	0x62	1268
c	0x63	2527
d	0x64	2370
e	0x65	8806
f	0x66	1381
g	0x67	4943
h	0x68	3456
i	0x69	5406
j	0x6A	1638
k	0x6B	513
l	0x6C	3072
m	0x6D	1988
n	0x6E	5008
o	0x6F	5159
p	0x70	1763
q	0x71	209
r	0x72	4376
s	0x73	4917
t	0x74	7297
u	0x75	3625
v	0x76	677
w	0x77	1219
x	0x78	837
y	0x79	1035
z	0x7A	361
{	0x7B	73
|	0x7C	138
}	0x7D	1188
~	0x7E	17126
total		409600

This

column

is

unintentionally

left

blank.

.

.

.

.

OR IS IT ?!?!

The following characters were inserted to make the above converge: 000004669