

# The Trivial Program “yes”

*Nick Montfort*

January 2012

TROPE-12-01

## **Abstract**

A trivial program, one that simply prints “y” or a string that is given as an argument repeatedly, is explicated and examined at the levels of function and code. Although the program by itself is neither interesting or instructive, the argument is presented that by looking at “yes” it is possible to better understand how programs exist not only on platforms but also in an ecology of systems, scripts, and utilities.

(Prepared for the Critical Code Studies Working Group 2012.)

A technical report from

The Trope Tank  
Massachusetts Institute of Technology  
77 Massachusetts Ave, 14N-233  
Cambridge, MA 02139 USA  
<http://trope-tank.mit.edu>

© 2012 Nick Montfort  
This work is licensed under the Creative Commons  
Attribution-ShareAlike 3.0 Unported License.  
To view a copy of this license, visit:  
<http://creativecommons.org/licenses/by-sa/3.0/>  
or send a letter to Creative Commons, 444 Castro Street,  
Suite 900, Mountain View, California, 94041, USA.

“yes.c” is licensed under the GNU General Public License v3.

## Introduction

I have spent a significant amount of my time over the past two years looking at a computer program that could be considered a sort of toy problem [1].

What is a toy problem? Let me Google that for you. I find what is up on Wikipedia at the moment on the topic to be pretty useful here:

In scientific disciplines, a *toy problem* is a problem that is not of immediate scientific interest, yet is used as an expository device to illustrate a trait that may be shared by other, more complicated, instances of the problem, or as a way to explain a particular, more general, problem solving technique.

For instance, while engineering a large system, the large problem is often broken down into many smaller toy problems which have been understood in good detail. Often these problems distill a few important aspects of complicated problems so that they can be studied in isolation. Toy problems are thus often very useful in providing intuition about specific phenomena in more complicated problems. [2]

This is essentially an elaboration of some text that, according to the history page, was originally written back in 2004 by a nameless individual at IP address 80.61.85.16, which is assigned to some computer in the Netherlands. It captures the essence of toy problems, which is that they do not directly deal with “real world” matters and are not of the same complexity as problems that do, but are nevertheless are interesting, educational, and thought-provoking.

A trivial problem is uninteresting. How can you determine whether all the elements in a series of length zero are prime? Well, of *course* they’re all prime. And they’re also all composite. And they’re also all blue. There are *no* elements. The problem is trivial and can be solved without even knowing what “prime” means. If we had some elements, it might not be, but without any elements, there’s nothing to talk about.

A toy problem is not trivial. It is useful because it allows one to play with something interesting that scales up to a more complex and directly significant problem. A toy problem is bounded, abstracted, made simpler and neater than the really useful-to-consider but also more tricky problems. But it captures something interesting. There is something to talk about in the toy problem, which as 80.61.85.16 and his or her collaborators tell us can be used as an “expository device” and which manages to “distill a few important aspects of complicated problems so that they can be studied in isolation.” The toy problem is often a very necessary step that allows scientists, engineers, mathematicians and others to explore and learn – before they put away childish things (in that particular case) and move on to some serious problem.

Furthermore, if one considers a toy problem not only as a mental exercise and learning experience, going on instead to understand why the toy problem was constructed and what it represents about cognition and our perspective on the world, the toy problem can be particularly valuable. For instance, a one-line BASIC program that generates random mazes can shed light on cultural perspectives on the maze, on how randomness and regularity come together in visual design, and on the history of the computer and programming language that this program runs on.

Enough of these toy programs, at least for the moment: That book will be out soon. For now, consider the trivial. Programs of this sort, utterly uninteresting from a computational standpoint, nevertheless can tell us about culture and computing.

### Getting to “yes”

Any modern Unix-like system – and for many people today this means Linux (or, to be politically correct, GNU/Linux) or OS X – features the command “yes,” just as it implements “ls” to see what is in the current directory, and “cd” to change to a different directory, and so on. It’s run from a terminal window or, in the rare case in which one is not running a GUI at all, from the command line on one’s textual screen.

When one types “yes” at one’s shell prompt, a line of y’s rapidly pushes whatever is displayed upwards, so that the terminal features nothing but a line of y along the left edge. In some cases, one might see the y on the bottom flickering slightly, indicating that a boundless supply of y’s is being rapidly produced. The result might be particularly pleasing to Norwegian concrete poet Ottar Ormstad [3].

That is all the program “yes” does when invoked in that way, the default way. It continues producing the letter y again and again unless some event outside of the program itself interrupts it: CONTROL-C being pressed to interrupt the program, for instance, or the power being cut off, or the whole operating system crashing. There is a bit more to the program “yes,” as will be explained later, but not much more.

The obvious question to ask about this program is “why, why, why?” What’s the meaning of something like this, the purpose? How can it be interpreted? What does it relate to?

As Eldon Tyrell says in *Blade Runner*, I want to see a negative before I provide you with a positive. So let’s consider that “yes” may have something to do with a long, famous, yes-filled soliloquy, the one that ends like this:

... and the figtrees in the Alameda gardens yes and all the queer little streets and the pink and blue and yellow houses and the rosegardens and the jessamine and geraniums and cactuses and Gibraltar as a girl where I was a Flower of the mountain yes when I put the rose in my hair like the Andalusian girls used or shall I wear a red yes and how he kissed me under the Moorish wall and I thought well as well him as another and then I asked him with my eyes to ask again yes and then he asked me would I yes to say yes my mountain flower and first I put my arms around him yes and drew him down to me so he could feel my breasts all perfume yes and his heart was going like mad and yes I said yes I will Yes. [4]

Molly Bloom’s use of “yes” throughout this “Penelope” chapter of *Ulysses* has been seen as part of her linguistic ravelling and unravelling: “... her strategy is to knot in order to unknot and to ‘not’ in order to ‘un-not.’ The first word of her soliloquy – Yes – may represent the removal of the first of these knots and every one of the many yeses that follows is another ‘not’ undone.” [5] Is the more repetitive soliloquy produced by “yes” also an example of undoing the “not” and unraveling something that has been put in place for purposes of Penelopistic delay?

O Jamesy let me up out of this ... [6] The purpose of this program is not all that difficult to explain, actually, but to figure out what it is for, it simply will not work to attack it directly and immediately with the canon. One has to first set down the novel, take a step back, see how Unix-like operating systems function, and consider some of the practical concerns that arose in the use of these systems – concerns that led to the development of “yes.”

## Scripts, Pipes, and Code Machines

It's often the case in working with a Unix-like system, and in doing even the most routine maintenance and installation of programs, that a person has to confirm something a script is doing. A script will often print a question and require confirmation from the user before it proceeds. What the user needs to type is "y."

Furthermore, Unix and the operating systems like it are set up to allow command-line programs to communicate with one another and to read from and write to files. They do so using input redirection, output redirection, and pipes, implemented on the command line using the symbols `<`, `>`, and `|`. The `|` symbol, the "pipe," is used to connect programs together as they run, sending the output from one to the next one in the pipeline.

Pipes specifically were a major innovation that allowed for a simple, flexible form of interprocess communication. Since the way each Unix program provided output and accepted output was standardized, it was possible to hook these programs to each other in a pipeline. For instance, if one wants to see who is logged to the system, the command "who" can be used; if one wants to sort some text, "sort" will serve to do this; and if one wants to see output a page at a time rather than having it all scroll past the viewable area of the terminal, the program "more" is one that will accomplish this. In Unix and the operating systems that followed it. With pipes, it's possible to output all the users currently logged onto the system, sort that list alphabetically, and send the result to a pager so that one can read through it a window at a time – simply by typing:

```
$ who | sort | more
```

In this case, the user will certainly want to exit the "more" program at some point after surveying the alphabetized users. Doing so breaks the pipe between "sort" and "more," which causes "sort" to exit, which breaks the pipe between "sort" and "who," which causes "who" to exit. What sounds at first like a plumbing disaster is actually exactly the desired behavior. There is no need for "sort" and "who" to keep running after the user has chosen to exit "more." It's cleanest for everything in the original pipeline to shut down.

Pipes were suggested by M. Doug McIlroy at Bell Labs, where Unix was originally developed, and were incorporated into that system in 1973 by Ken Thompson [7].

A general justification for pipes is that programs are often taking the output of other programs as input. But programs also work on other programs in all sorts of ways. A shell script, for instance, can be used to install a new program on the system. (There are more standardized ways of installing a new program, but individual shell scripts have often been used for this purpose and are still used at times.) A script shell such as "install.sh," run using the shell "sh," may need to modify some existing files, and it may ask the user if it should continue the installation and make these changes. Unless the script was run by accident, the answer is usually "yes," or "y." Instead of waiting through the entire installation procedure and pressing "y" every time a question mark appears, it can be very convenient to supply a continue stream of y's to one's install script. The program developed for this very purpose is "yes," which can be run as follows:

```
$ yes | sh install.sh
```

While “yes” can be run by itself and will run until interrupted by `CONTROL-C`, or by closing the terminal, or by something else, when “yes” is run as part of a pipeline it only keeps running as long as the next program is running. When the installation script terminates, “yes” does as well.

The idea here is not to try to ban any mention of James Joyce in code studies. Discussion of code and Joyce has already been undertaken quite productively [8]. Furthermore, “yes” is in fact code that is written to help unravel a series of yes-or-no questions that other code has raveled together, a litany of questions that is constructed to require time, attention, and some sort of effort (even if it is just a keystroke per question and a press of the `ENTER` key) from the user. Sternlieb’s discussion of Molly Bloom’s soliloquy may in fact lead to a useful way to view this program [9].

The point here is rather that a program like this, taken up in isolation and held against a well-known and often interpreted literary work, is not going to be understood nearly as well as when that program is considered in its context of development and use.

### Related Recreational Programs

It’s possible to locate programs that are written for fun, rather than as utilities, and that relate to “yes.” One of them is a BASIC program I wrote in middle school, one that was, in different versions, spontaneously discovered by numerous young programmers during the home computer version. My version of it looks like this:

```
10 PRINT "NICK RULES" : GOTO 10
```

You can see from one of Cat Keynes’s blog posts, entitled `10 PRINT "CAT IS GREAT " ; GOTO 10`, that I was not alone in writing a program like that at some point [10].

(Keynes’s version, which adds two spaces at the end and a semicolon after the end of the string, fills the screen instead of printing a single column.) Matt Dowell has a blog named `10 PRINT "MATT", 20 GOTO 10` [11]. The comma, no doubt, is meant in this title to show where a line break should go. In a 2002 interview, Adrian O’Grady responds to the question “What was your first program?” [12] with:

```
10 PRINT "ADRIAN"
20 GOTO 10
```

There are dozens of instances of similar programs that print names and that print more conventional and neutral phrases such as “HELLO WORLD.” That phrase, of course, is seen in another famous trivial program which is often used to introduce programming languages [13].

The computer that I now use every day does not have a built-in BASIC interpreter, but my version of this program can be implemented very easily using “yes.” One simply types the following rather satisfying line at the Linux shell prompt:

```
% yes NICK RULES
```

And enjoys the anticipated result that is depicted in figure 1.

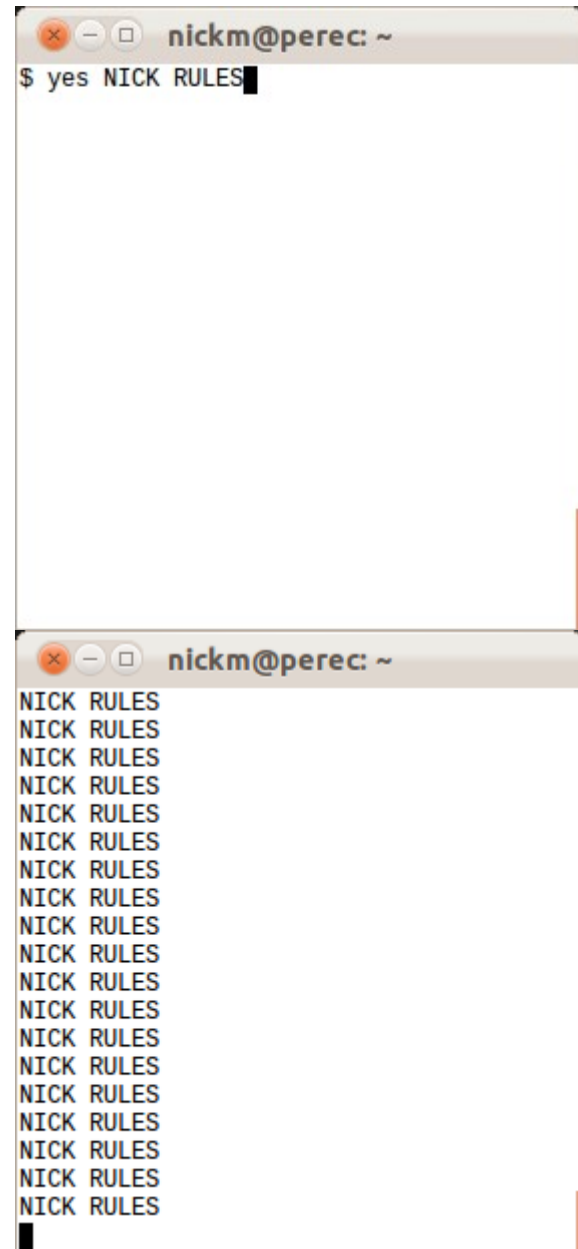
It must be that “yes” resonates profoundly with some digital media artists – or at least with me. For the May 21-22, 2008 Software Studies Workshop at UC San Diego, I developed a Python program named `yes_voices.py` that did nothing but repeatedly affirm, doing so in 15 different voices and then displaying its own code during my Pecha-Kucha-format talk [14]. Cicero Inacio da Silva, who must have been a big fan of *Sim City*, translated this to Portuguese as `vozes_sim.py` [15].

This raises the issue of how easy it would be to translate the program “yes” to various other languages. Without trying to tackle that problem comprehensively, or even for a wide variety of languages, it can be observed that translators will encounter difficulties in translating “yes” to certain languages – for instance, to Latin. In that language, there is no general-purpose word that means “yes.” One typically affirms by repeating the verb from the original question in the appropriate conjugation: “*Stare?*” (“Do you stand?”) “*Sto.*” (“I stand.”) Words such as “*certe*” (“certainly”) have been pressed into service by contemporary Latin-speakers desperate to let their telephonic interlocutors know that they are listening, and can also be found used in yes-like ways in some ancient writings, but these do not map exactly onto the English “yes.” So, even this trivial use of language, pure affirmation by means of a single repeated word, is linguistically situated and can be a starting point for encounters of important differences between languages.

### An Automated Yes Man

A further way that this program is situated in culture is seen in how it affirms by default. The program “yes” can of course be made to do otherwise: “yes no” prints “no” endlessly, “yes n” prints innumerable n’s, and one is even welcome to type “yes maybe.” But the program is called “yes,” not “no” or “maybe,” and what it does by default – without an argument of any sort – is to continually affirm.

We can imagine a mirror-universe in which the world is more or less the same, but where the program “no” is standard instead of “yes” and I have a beard. In this uncanny



```

nickm@perec: ~
$ yes NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES
NICK RULES

```

Figure 1. A particular invocation of “yes” and the resulting output.

alternate reality, installation scripts would usually ask questions such as “/home/nickm/.config/queso.conf already exists -- mind if I modify it?” and would need to be supplied with the answer “n” or “no” in order to continue. Presumably sailors in this world would be expected to usually say “no, sir” and “nay-nay” in response to appropriately-worded inquiries and commands from their captain. There might be informational campaigns to remind people that “yes means yes.” That is, everything could go on as it does now if the role of “yes” and “no” were flipped. No men would be yes men.

So, perhaps it makes more sense to concentrate on what all those y’s are doing instead of trying to understand “yes” as a system for automated affirmation. It’s a more general program than that, for one thing, and as just shown, it could work identically, in a different context, if all it did was spout n’s.

As it is actually used to help accomplish tasks, “yes” replaces a person who is supposed to read and needs to confirm many steps of an installation or other process. By converting an interactive process with prompts to one that is done without human intervention, “yes” acts less like an affirmation system specifically and more as a trivial tool for automation – specifically, the automation of a process that was intentionally made to require human intervention.

### **An Obligatory Program within Unix, Linux, and OS X**

Many programs that normally require confirmation when they run implement a flag that allows the user to circumvent the repeated typing of “y.” For instance, “rm,” which is used to delete files and directories, will not ask for confirmation if the “-f” or “force” flag is used. As programs are converted to offer such conveniences, it’s possible to imagine a point at which every program offers such a flag and “yes” becoming obsolete.

No such luck.

It’s unlikely that it will ever be possible to remove “yes” from Unix-like systems, even if all the core parts of a system – the other utilities – will no longer possibly need it. (It’s not clear to me that they could possibly need to be fed the output of “yes” even now, but I have not checked exhaustively.) In fact, even if OS X or a Linux distribution offered flags to override the interactive inquiries of every program in the distribution, there would still be scripts that invoke “yes” and depend on it. While “yes” wouldn’t be strictly necessary any more, removing “yes” would break those scripts, scripts that would otherwise run without any trouble [16].

However trivial this program is, now that it has been issued as a standard part of Unix-like systems, “yes,” like bullets, cannot be recalled.

### **Obligatory and Conventional Code within “yes”**

Clearly there’s plenty to discuss surrounding this trivial program, “yes.” But we can also discuss the program itself in greater depth. We’ve said more or less everything it does, but since there are plenty of free software implementations of yes, such as the one by the GNU (GNU’s Not Unix) project, we can also very easily crack open the code and take a look [17].

Looking in coreutils-8.15.tar.xz, the version of the core utilities from January 6, 2012, reveals this code for “yes” in the file yes.c (line numbers have been added):

```

1  /* yes - output a string repeatedly until killed
2  Copyright (C) 1991-1997, 1999-2004, 2007-2012 Free Software Foundation, Inc.
3
4  This program is free software: you can redistribute it and/or modify
5  it under the terms of the GNU General Public License as published by
6  the Free Software Foundation, either version 3 of the License, or
7  (at your option) any later version.
8
9  This program is distributed in the hope that it will be useful,
10 but WITHOUT ANY WARRANTY; without even the implied warranty of
11 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 GNU General Public License for more details.
13
14 You should have received a copy of the GNU General Public License
15 along with this program. If not, see <http://www.gnu.org/licenses/>. */
16
17 /* David MacKenzie <djm@gnu.ai.mit.edu> */
18
19 #include <config.h>
20 #include <stdio.h>
21 #include <sys/types.h>
22 #include <getopt.h>
23
24 #include "system.h"
25
26 #include "error.h"
27 #include "long-options.h"
28
29 /* The official name of this program (e.g., no `g' prefix). */
30 #define PROGRAM_NAME "yes"
31
32 #define AUTHORS proper_name ("David MacKenzie")
33
34 void
35 usage (int status)
36 {
37     if (status != EXIT_SUCCESS)
38         fprintf (stderr, _("Try `%s --help' for more information.\n"),
39                 program_name);
40     else
41     {
42         printf (_("\
43 Usage: %s [STRING]...\n\
44 or: %s OPTION\n\
45 "),
46                 program_name, program_name);
47
48         fputs (_("\
49 Repeatedly output a line with all specified STRING(s), or `y'.\n\
50 \n\
51 "), stdout);
52         fputs (HELP_OPTION_DESCRIPTION, stdout);
53         fputs (VERSION_OPTION_DESCRIPTION, stdout);
54         emit_ancillary_info ();
55     }
56     exit (status);
57 }
58
59 int
60 main (int argc, char **argv)
61 {
62     initialize_main (&argc, &argv);
63     set_program_name (argv[0]);
64     setlocale (LC_ALL, "");
65     bindtextdomain (PACKAGE, LOCALEDIR);
66     textdomain (PACKAGE);
67
68     atexit (close_stdout);
69

```



```

70     parse_long_options (argc, argv, PROGRAM_NAME, PACKAGE_NAME, Version,
71                         usage, AUTHORS, (char const *) NULL);
72     if (getopt_long (argc, argv, "+", NULL, NULL) != -1)
73         usage (EXIT_FAILURE);
74
75     if (argc <= optind)
76     {
77         optind = argc;
78         argv[argc++] = bad_cast ("y");
79     }
80
81     while (true)
82     {
83         int i;
84         for (i = optind; i < argc; i++)
85             if (fputs (argv[i], stdout) == EOF
86                 || putchar (i == argc - 1 ? '\n' : ' ') == EOF)
87                 error (EXIT_FAILURE, errno, _("standard output"));
88     }
89 }

```

The fascinating thing here is that an 89-line program has been written to do more or less what the one-line or two-line “NICK RULES” and “HELLO” BASIC programs do. This program “yes” is just slightly more general – although the BASIC programs are pretty general, too, since it’s possible to simply edit the code and change the string that they print. Another way to put this into perspective is to consider how else one could get a repeating “y” to be printed on a Unix-like system. It’s possible to do it with the following one-line Perl program, invoked like so on the command line:

```
$ perl -le '{print"y";redo}'
```

The entire text that needs to be typed here is only 25 characters long – so, only 22 characters more than typing “yes.” And one can easily change the “y” in this code to whatever arbitrary string one wishes to be repeatedly printed. So this 89-line C program “yes” only results in a few keystrokes saved.

Consider another C program, “y.c,” that does not provide for printing arbitrary strings but does the basic task of printing a “y” repeatedly:

```

1     #include <stdio.h>
2
3     int main(void)
4     {
5         while (1)
6         {
7             printf("y\n");
8         }
9     }

```

The full yes.c has 2409 characters; y.c, on the other hand, which is not particularly compressed and which follows the formatting conventions of yes.c while doing the same core task, has only 82. It is much smaller – less than 3.5% the size of yes.c. Although it certainly costs some extra code to generalize this program so that it can repeatedly output any string, it seems that there is still a tremendous difference – more than can be accounted for by this generalization – in the size of the two programs.

It may be possible to go through and determine the purpose of each character in yes.c, to quantitatively divide the code. It’s not clear what the value of this character-counting would be, though, even if each character could be assigned to one unambiguous purpose. It seems interesting enough to just determine what all that

other code is for – what purposes, including generalization of the nine-line program but going beyond that, are at work here.

Some of it is clearly for generality. Consider “`int argc, char **argv`” in line 60, which allows the program to accept arguments from the command line. This, along with the other references to `argc` and `argv`, allow `yes.c` to accept command-line parameters and repeatedly output whatever is typed after “`yes`.”

But the 17 lines of comments at the beginning are there for a different reason. They – along with the `#define` directives on lines 30 and 32 – are there because they are obligatory for GNU programs, particularly GNU core utilities. This is standard information that must be provided. The “`usage`” function, lines 34-57, has a concrete effect on how the program operates but is also there for obligatory purposes, so that when one adds an undefined flag (something other than “`--help`” or “`--version`”) to the program’s name on the command line, an informative response will be provided.

```
$ yes --whatever
yes: invalid option -- '-'
Try `yes --help' for more information.
```

Following this suggestion leads to some additional text:

```
$ yes --help
Usage: yes [STRING]...
  or: yes OPTION
Repeatedly output a line with all specified STRING(s), or `y'.

    --help      display this help and exit
    --version   output version information and exit

Report yes bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
For complete documentation, run: info coreutils 'yes invocation'
```

And, it’s possible to try the “`--version`” flag and see the other possible non-repeating output:

```
$ yes --version
yes (GNU coreutils) 8.5
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Written by David MacKenzie.

The outputs from “`yes`” with the last two flags are generated from some data in `yes.c` and from the included file `system.h` – yet more code, in the form of a 591-line header file. What those texts say could possibly be nice to know, but above all, they are standard to have these options on all GNU core utilities. They are obligatory, a sort of boilerplate. Much of the code that isn’t directly providing a string is of this sort: standard, expected, conventional, and included in obedience to best practices.

While a trivial program may be uninteresting as computation, it can be very

interesting to look at if one wants to identify how much of this obligatory, conventional code there is across all the programs in a project such as the GNU core utilities.

### Conclusion

In the case of “yes,” what is clearly a trivial program (easy enough for a child to a program in BASIC) is nevertheless a program that has (or at least had) some use. Since the trivial program “yes” became standard long ago [18], it seems that it is now impossible to extricate it from Unix-like systems. And, the program, which can be implemented very briefly either in Perl (now standard on all Unix-like systems) or using a different bit of C code, reveals the large amount of obligatory and conventional code that many programs have.

The word “yes” by itself, completely in isolation, means nothing. Without the rest of the discourse, it answers no question and affirms no statements. Similarly, the program “yes.c” means little by itself. It points, however, toward important facts about computer programs and their use, in technical terms and within culture. A trivial program may be important because it is part of standard distribution, an operating system, a set of utilities, a pipeline. It may be invoked in practices of software installation, to convert what was intended to be a manual process to an automatic one. It can help us see that programs live within systems of development and use.

### Notes and References

1. (I’ve been writing a single-voice book with nine other authors about a one-line Commodore 64 BASIC program: Montfort, Nick, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter, *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*, forthcoming from MIT Press, 2012. The title of the book is the program itself. This program draws a maze pattern on the screen by printing, at each step, one of two different PETSCII characters corresponding to the two diagonal lines. Our project represents a new type of scholarship (a 10-author collaboration to produce something that reads like a standard monograph, facilitated by MediaWiki and other software and systems) and explodes the interpretation of code in a way that I believe should profoundly change digital media studies, opening up new understandings of how code exists in culture.)
2. 80.61.85.16 et al., “Toy Problem,” *Wikipedia*. Created December 27, 2004. Last updated by Slightsmile June 4, 2011 23:44. [http://en.wikipedia.org/wiki/Toy\\_problem](http://en.wikipedia.org/wiki/Toy_problem)
3. Ormstad, Ottar. *y (gul poesi)*. 2005. [http://www.nokturno.org/files/ottar-ormstad/ottar-ormstad\\_y.html](http://www.nokturno.org/files/ottar-ormstad/ottar-ormstad_y.html)
4. Joyce, James. *Ulysses: The Corrected Text*. 1986. Ed. Hans Walter Gabler with Wolfhard Steppe and Claus Melchior. New York: Random House. pp. 643-644, lines 18.1599-1609.
5. Sternlieb, Lisa. “Molly Bloom: Acting Natural.” *English Literary History* 65: 3, 757-778. Fall, 1998. p. 766.
6. Joyce. p. 633, lines 18.1128-29.
7. The Linux Information Project. “Pipes: A Brief Introduction.” Created April 29, 2004. Last updated August 23, 2006. <http://www.linfo.org/pipe.html>
8. Black, Maurice J. “The Art of Code.” PhD Dissertation, University of Pennsylvania, Department of English. 2002.
9. (After all, Tyrell did in fact present Deckard initially with a “positive,” an actual

- replicant, in *Blade Runner*.)
10. <http://catkeynes.com/CS00060.html>
  11. <http://mattdowell.blogspot.com>
  12. [http://dcemulation.org/?title=Interviews:Adrian\\_0%27Grady](http://dcemulation.org/?title=Interviews:Adrian_0%27Grady)
  13. Marino, Mark. "Critical Code Studies." *Electronic Book Review*. 12-04-2006.  
<http://www.electronicbookreview.com/thread/electropoetics/codology>
  14. Montfort, Nick. "My Generation about Talking." *nickm.com*. May 21, 2008.  
[http://nickm.com/if/yes\\_voices.py](http://nickm.com/if/yes_voices.py)
  15. Montfort, Nick. "Geração sobre a fala." Portuguese translation of "My Generation about Talking," trans. Cicero Inacio da Silva. February 13, 2010. [http://nickm.com/if/vozes\\_sim.py](http://nickm.com/if/vozes_sim.py)
  16. (People might find it annoying, too – it's possible that some programmers are out there who habitually fire up "yes" themselves, on the command line, to override the questions asked by particular programs or scripts. But people can't break the way that scripts do. A change that inconveniences a user can be implemented with a shrug and perhaps the word "deprecated." One that breaks scripts will elicit bug reports and cannot be undertaken as lightly.)
  17. (There is a reason that some people insist on calling the operating system commonly known as "Linux" "GNU/Linux." Much of what makes it work is not the Linux kernel but the programs that surround it, such as the core utilities, which were developed by the GNU project.)
  18. (I do not know how long ago. The GNU "yes" seems to have been first written in 1991, based on the copyright information, but this was a reimplementaion of the Unix "yes.")