26 Aug 2022. doi:10.14324/000.rp.10157010

Generating Random Infix Expressions for GNU coreutils expr

William B. Langdon

Department of Computer Science, University College London, Gower Street, WC1E 6BT, UK

E-mail: `w.langdon@cs.ucl.ac.uk`

Abstract. We use the recent random_tree() addition to GPquick [arXiv:2001.04505] to uniformly sample in linear time the space of binary trees. A unix gawk script transforms these to uniform random infix expressions, as used by Free Software Foundation GNU core utility `expr`. It converts from Lisp s-expression like prefix representation used by GPquick to bracketed infix expressions, e.g. `"(" 3050 "=" 5514 ")" "-" 3073`. gawk randomly labels internal tree nodes with the 14 functions known to `expr` and replaces leafs with randomly chosen positive integers up to 32768. About 80% of random expressions are rejected, since they cause `expr` to fail, typically due to division by zero.

Keywords prefix, infix, software under test, SUT, empirical software robustness, failed error propagation, FDP

In order to measure the robustness of software [1], in *cora* we use a thousand or more tests drawn at random from a known distribution (and hence with a known entropy). We record the program's normal output, disrupt the software under test and compare the disrupted program's answer on each test with its undisturbed answer. We find many cases of *failed error propagation* [2], where despite potentially severe internal change, the program's output is unchanged.

The current version (9.1 released 15 April 2022) of the GNU C coreutils was downloaded from `https://ftp.gnu.org/gnu/coreutils/coreutils-9.1.tar.gz`. Coreutils contains many utilities, including `expr`. `expr.c` comprises 779 lines of C code (excluding include header files).

Our new random_tree() function [3] was run on all legal tree sizes up to 400 nodes (i.e. from 1, a single leaf, to 399 = 199 functions and 200 leafs). For each size, 60 random binary trees were generated and then randomly labeled. We tested each random expression by seeing if `expr` could evaluate it. We discarded those random expressions for which `expr` failed. So although $200 \times 60 = 12\,000$ random expression were generated, only 2430 were suitable as test cases (see Figure 1).

`expr` allows 14 different functions: `+ - * / % | & < <= = == != >= >`. Each internal node in the randomly generated tree was labeled with one of these 14 chosen uniformly at random. In our random expressions, data values were randomly selected from the first 32768 positive integers. Each binary tree generated by random_tree() was converted into an infix expression by a gawk script, see the appendix. Our gawk function eval() steps through the binary tree sequentially. Effectively doing a depth first pass. On internal nodes eval() calls itself twice (once per branch), generating opening and closing brackets, and random functions and numbers as it goes. Extensive use of round brackets is needed to ensure the order of evaluation follows the operator precedence set by the choice of random binary tree, rather than the implicit precedence of the 14 available operators.

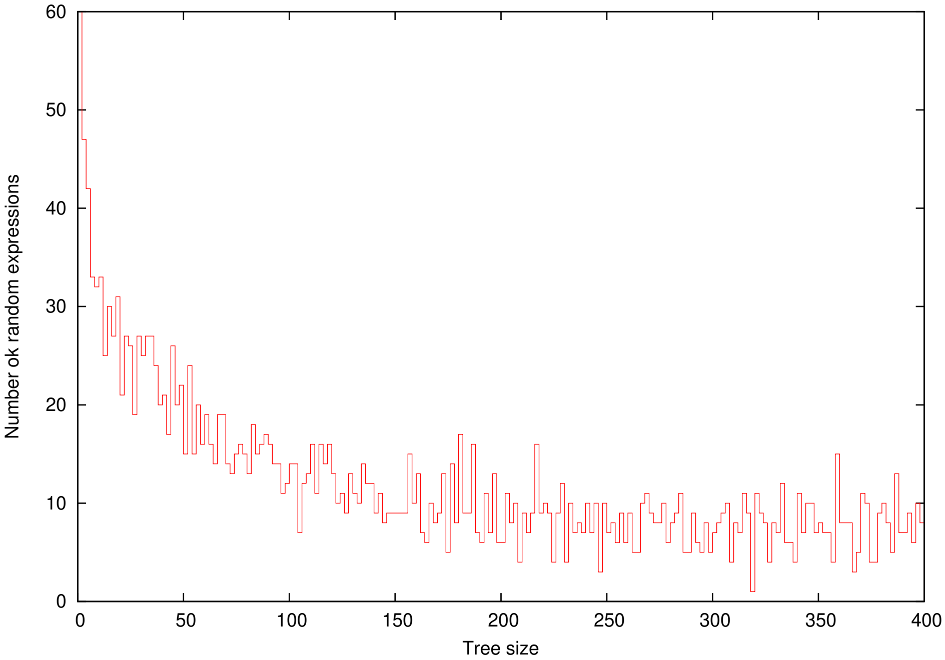

Figure 1. Number of random infix expressions that can be processed by unix expr command. (Most failures are due to divide by zero errors.) 60 random expressions were created for each odd tree size. (Tree size refers to the base binary tree and so excludes brackets.)

Acknowledgements

Funded by the Meta OOPS project. It is intended that the `expr` test dataset will be released as part of the cora GitHub project. Until then please contact the author via email.

References

[1] Langdon, W.B., Petke, J.: Software is not fragile. In: Parrend, P., et al. (eds.) Complex Systems Digital Campus E-conference, CS-DC'15. pp. 203–211. Proceedings in Complexity, Springer (Sep 30-Oct 1 2015), `http://dx.doi.org/10.1007/978-3-319-45901-1_24`, invited talk

[2] Petke, J., et al.: Software robustness: A survey, a theory, and some prospects. In: Avgeriou, P., Zhang, D. (eds.) ESEC/FSE 2021, Ideas, Visions and Reflections. pp. 1475–1478. ACM, Athens, Greece (23-28 Aug 2021), `http://dx.doi.org/10.1145/3468264.3473133`

[3] Langdon, W.B.: Fast generation of big random binary trees. Tech. Rep. RN/20/01, Computer Science, University College, London, Gower Street, London, UK (13 Jan 2020), `https://arxiv.org/abs/2001.04505`

Fragment of gawk Script to Convert from Prefix to Infix

We start with a single line of text containing a flattened binary tree in GPquick format (i.e. in prefix representation). To avoid changes to existing C++ code, all the internal nodes are represented by the binary function MUL. `rand_function()` replaces MUL with one of 14 function symbols known to `expr` chosen at random. Anything which is not MUL is a leaf, and the `else` branch in function Print(text) replaces it with a randomly chosen integer.

Function eval() processes the binary tree in depth first (left–right) order. It is initially called with it's input `I` pointing to the tree's root node, which is the left most part of the tree in the input line. At each step the input `I` is moved on to the next node in the tree. At tree branch points, eval() recursively calls itself twice (one per branch). On tree leaf nodes, it prints a random integer. In both cases eval() returns the new value of `I`.

Note pushing and popping is handled implicitly by gawk. Declaring the variables `I_, nargs, i` and `simple` ensures they are treated as local, rather than global, and so each eval() call gets it own clean copy, which is deleted when eval() returns.

```
function rand_function( n,t) {
  n = split("+ - * / % | & < <= = == != >= >",t);
  return t[1+int(n * rand())];
}
function Print(text) {
  if(index(text,"MUL")) printf("\"%s\" ",rand_function())
  else                  printf("%d ",int(1+32768*rand()));
}
function eval(I,  I_,nargs,i,simple){
  I++;
  I_ = I;
  nargs = index($I,"MUL")? 2 : 0;
  if(nargs==2) {
    simple = index($(I+1),"MUL") == 0;
    if(!simple) printf("\"(\" ");
    I = eval(I);
    if(!simple) printf("\")\" ");
    Print($I_);
    simple = index($(I+1),"MUL") == 0;
    if(!simple) printf("\"(\" ");
    I = eval(I);
    if(!simple) printf("\")\" ");
  } else Print($I);
  return I;
}
```