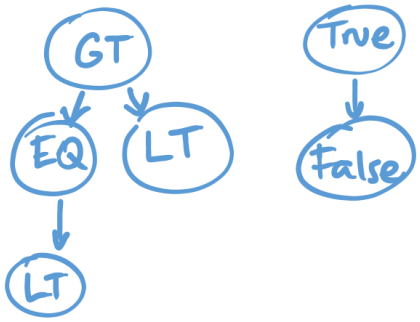
# elm-test guide to integrated shrinking

## How elm-explorations/test did things before #151: value-based shrinking

```
Fuzzer a = Generator (RoseTree a)
```




Each level is simpler than its parent, hidden behind a **Lazy** thunk.
Shrinking is just navigating the tree.
Value is fully shrunk when it has no more children in the tree.

Simplified:

```
Fuzzer a = { gen : Generator a, simplify : a -> List a }
```

Shrinking is done on the values.
Value is fully shrunk when the simplify function returns an empty list.

## After #151: integrated shrinking

Users no longer deal with shrinkers directly.

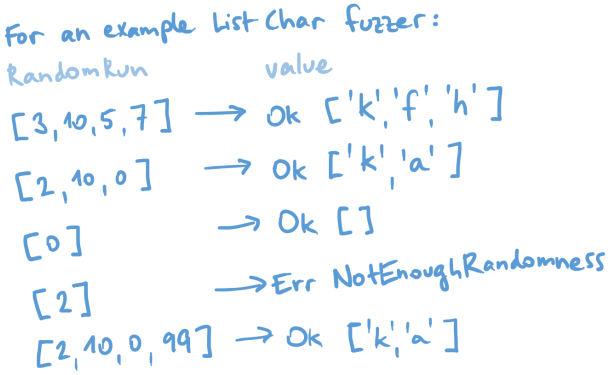
Added a layer of indirection: **RandomRun** (a history of drawn random integers).
Conceptually a `List Int`, but in practice a `Queue Int` was the most performant so we kept that.

Fuzzers take a **PRNG** source and try to convert the integers it gives into a value.

There are two types of PRNG sources: `PRNG = Random ... | Hardcoded ...`

**Random** corresponds to when you're drawing random ints (creating a **RandomRun** from scratch) when searching for the first failing value. This one can't run out of randomness, but it can run out of memory, so to speak - we're limiting max **RandomRun** length arbitrarily.

**Hardcoded** corresponds to you already having a **RandomRun** that generates a value that fails the test, and you're trying to shrink it. Can run out of randomness and frequently does (this is OK).




Simplified:

```
Fuzzer a : PRNG -> Result String (a, PRNG)
```

Shrinking is done on the **RandomRun**, and the goal of shrinking is shortening the **RandomRun** and minimizing its values to 0.

## Sensitivity to fuzzer definition

The way you write fuzzers matters: eg. there are multiple ways to write a `List` fuzzer:

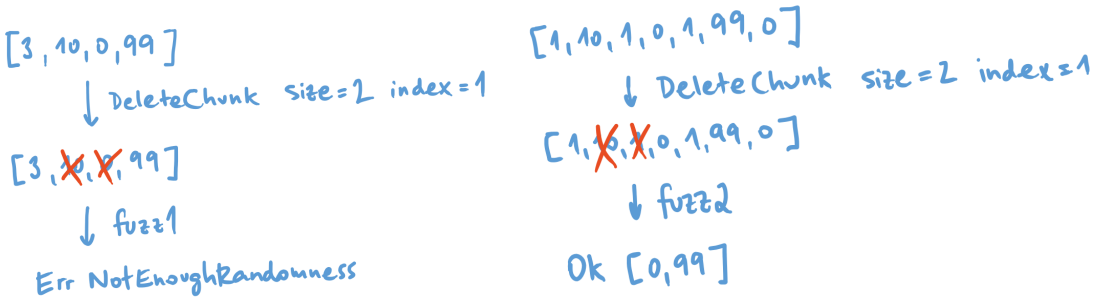
A) first generate an integer for length `|>` `andThen` generate that many items
B) generate a Bool for whether to generate an item `|>` `andThen` either stop or generate the item and go again

They will generate different `RandomRun`s for the same value: eg. value `[10,5,7]` could be created from a run `[3,10,5,7]` in case of the first one and from `[1,10,1,5,1,7,0]` in case of the second one.

There is a static set of `ShrinkCmd`s (actually, `SimplifyCmd`s but the naming habit is strong) that we try to shrink with, like eg. `DeleteChunk` or `MinimizeChoice` (choice being one integer drawn from the PRNG).

Different `RandomRun` patterns will shrink with different success rate.

The second fuzzer will have a different `RandomRun` for the same value (the list structure will be more like `[1,_,1,_,1,_,0]`, compared to `[3,_,_,_]` for the first fuzzer) which will line up with the `DeleteChunk` cmd nicely.




Note shrink cmds have no knowledge of what fuzzers were used to generate the `RandomRun`.[1]

So our goal as the library authors is to make the primitives synergize well with the `ShrinkCmd`s, to make it easy for the users to pick the performant approach. (Although they do have the freedom to do things from scratch their own way.)

Some more examples of `ShrinkCmd`s:
- Replace a chunk with 0
- Sort a chunk
- Swap a chunk with its neighbour
- Minimize an integer via binary search
- Redistribute between two neighbouring integers (we try to minimize `i1` while `i1+i2` stays constant)
- Decrement two integers together

Some of these terminate early based on the current values, and some are tailored or tweaked based on knowledge of what various fuzzers need. In general we try to a) maximize the success rate of the fuzzers (maximize their domain), and b) shrink towards the accepted `RandomRun`s (line up the range of shrink cmds with the domain of fuzzers as much as possible).

[1] This is perhaps our most promising avenue for optimization: if `Fuzz.float` wasn't used, we don't need to try `MinimizeFloat` shrink cmds at all, etc... Another optimization could be grouping ranges inside the `RandomRun` corresponding to specific fuzzers, and deleting/zeroing *these* instead of "random" chunks.

## Various tidbits

As an optimization, every `ShrinkCmd` also holds the minimal length the `RandomRun` must have before it can be ran. This helps with scenarios like:

- `RandomRun` has length 10
- `DeleteChunk` (size=8, index=2) ran successfully!
- `RandomRun` has length 2
- `DeleteChunk` (size=8, index=1) ran unsuccessfully...
- `DeleteChunk` (size=8, index=0) ran unsuccessfully...

In this case it's obvious the later two couldn't have succeeded, because their size exceeds the `RandomRun` length.

So after every successful shrink (that shortens the `RandomRun`), we filter the list of remaining `ShrinkCmd`s to only contain ones that aren't obviously useless.

After all `ShrinkCmd`s are finished, we try again if there was an improvement in any of them.
If we've reached a fixpoint, we're done shrinking.

---

A big part of the codebase now deals with floats and how to shrink them.

The binary64 format in the IEEE-754 standard has a sign bit as MSB, then the exponent (general magnitude of the number) and then the mantisa (the decimal number itself before scaling, in a sense). This combined with shrinkers trying to minimize the values means we frequently get values looking like `1.00000000082575` which, while being truly minimal in a sense, aren't very helpful or easy on the eyes.

The library Hypothesis from which #151 takes most of its inspiration does shuffle the bits in a float so that smaller values correspond to simpler fractions: `1.5` (3/2) is simpler than `1.25` (5/4) and so on.
So you'll get the "nicest" failing value possible, rather than something close to epsilon.
The relevant modules are `Fuzz.Float` and `MicroBitwiseExtra`.

---

There is an extensive test suite in `FuzzerTests` now; each exposed fuzzer has some tests.
There are also some shrinking challenges from [1] implemented in `ShrinkingChallengeTests`.

[1]: *https://github.com/jlink/shrinking-challenge*

# Map of the library

`Fuzz`
: Definitions for all the fuzzers.

  Building blocks:

  - `rollDice`: takes a `Generator Int`, draws an `Int` value and advances the `PRNG`
    - `uniformInt`: draws a `Random.int 0 n`
    - `weightedBool`: draws a `Random.float 0 1` and maps it to `0` or `1` with a given probability
    - `intFrequency`: draws a `Random.weighted` and returns a "bucket" index

  Everything else is built on top of these.

`Fuzz.Internal`
: Holds definition of `type Fuzzer`, so that the `Fuzz` module doesn't expose it to the users.

`Fuzz.Float`
: Holds `wellShrinkingFloat`, converting two 32bit integers to a float. The integers represent a custom encoding of the IEEE-754 data that shrinks towards nice values (small denominators).

`RandomRun`
: Holds definition of `type alias RandomRun` and various helpers to work with it.
  Mostly used by the `Simplify` module to interpret the cmds.

`PRNG`
: Randomness history for a given test run.
  Holds definition of `type PRNG`.

`GenResult`
: Just a `Result (String, PRNG) (a, PRNG)` with better naming.
  Holds definition of `type GenResult`.

`Simplify.Cmd`
: Operations as data. High-level descriptions of how we want to try shrink a given `RandomRun`.
  Holds definition of `type SimplifyCmd` and `cmdsForRun` as a way to create them.

`Simplify`
: Holds the function `simplify` which tries to improve upon the found failure for a given test.
  Contains implementations for all the `SimplifyCmd`s and a retry loop to simplify until a fixpoint.