

Intro to PBT



EBT

PBT

ScalaCheck

Choosing Properties

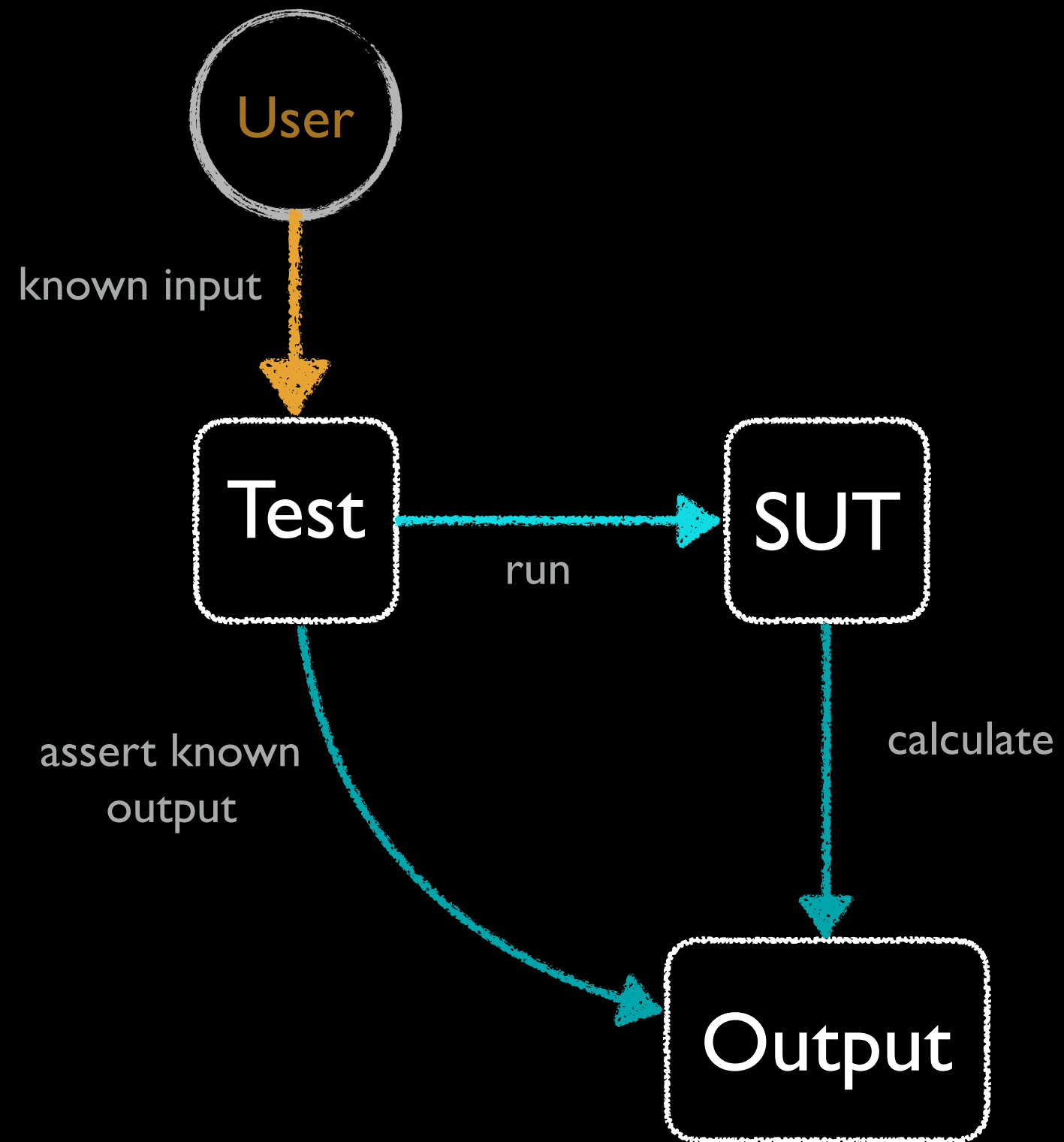
Examples

Summary

Testing shows the presence,
not the absence of bugs

Dijkstra

Example-Based Testing



known inputs

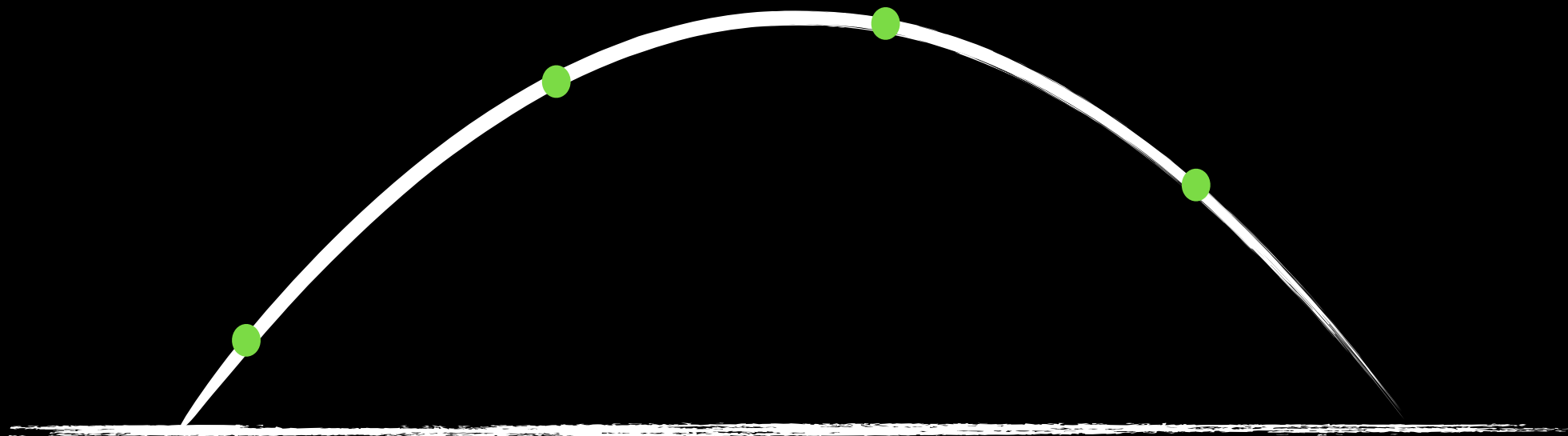
known output



add(1, 2) should be 3

The diagram illustrates the concept of known inputs and outputs for a function. It features the text "add(1, 2) should be 3" in white. Above the first two digits, "1" and "2", are the words "known inputs" in white. Two orange arrows point from "1" and "2" down to the text. Above the digit "3" is the phrase "known output" in white, with an orange arrow pointing from it down to the text. The digits "1" and "2" are colored cyan, and the digit "3" is colored green.

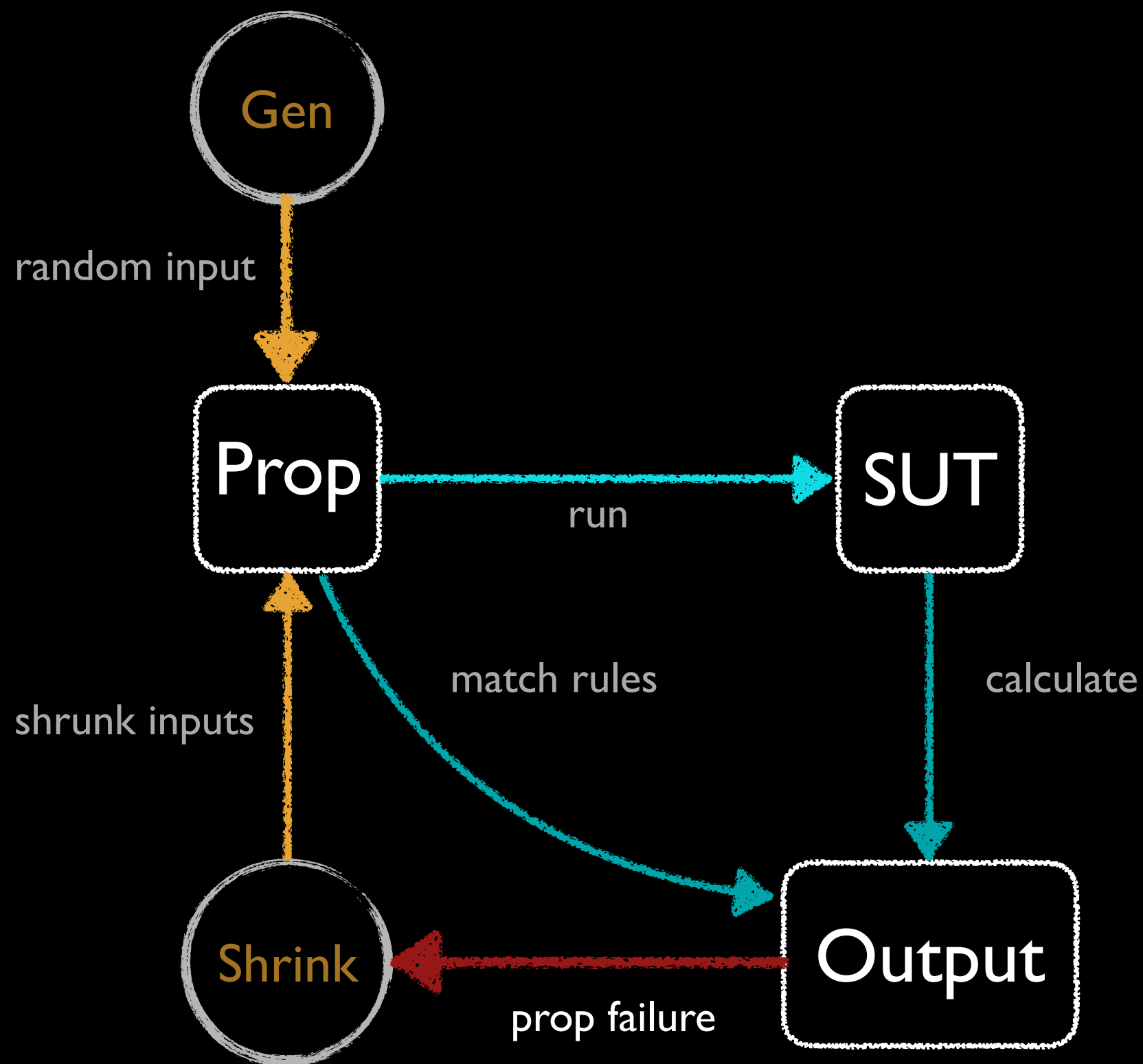




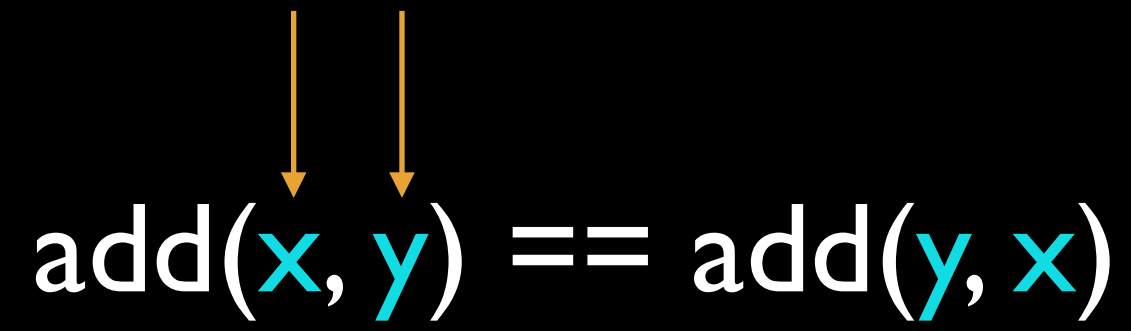
Input Sample

Property-Based Testing

x100



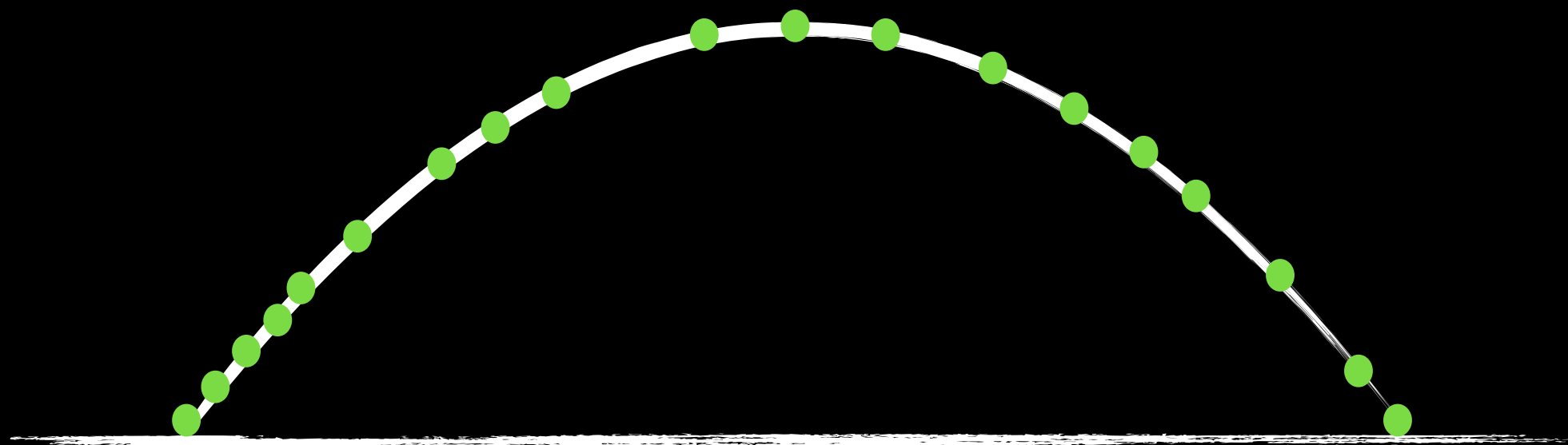
any inputs



The diagram illustrates the commutativity of the `add` function. It features the equation `add(x, y) == add(y, x)` in white text. The variables `x` and `y` in the first function call are highlighted in cyan. Two orange arrows point downwards from the text "any inputs" to these cyan variables, indicating that the equation holds for any values of `x` and `y`. In the second function call, the variables `y` and `x` are also highlighted in cyan.

$$\text{add}(x, y) == \text{add}(y, x)$$






Input Sample

ScalaCheck

Gen[A]



Arbitrary[A]



Arbitrary[String]

ꠏꠐꠑꠒꠓꠔꠕꠖꠗꠘꠙꠚꠛꠜꠝꠞꠟꠠꠡꠢꠣꠤꠥꠦꠧ꠨꠩꠪꠫꠬꠭꠮꠯꠰꠱꠲꠳꠴꠵꠶꠷꠸꠹꠺꠻꠼꠽꠾꠿ꡀꡁꡂꡃꡄꡅꡆꡇꡈꡉꡊꡋꡌꡍꡎꡏꡐꡑꡒꡓꡔꡕꡖꡗꡘꡙꡚꡛꡜꡝꡞꡟꡠꡡꡢꡣꡤꡥꡦꡧꡨꡩꡪꡫꡬꡭꡮꡯꡰꡱꡲꡳ꡴꡵꡶꡷꡸꡹꡺꡻꡼꡽꡾꡿ꢀꢁꢂꢃꢄꢅꢆꢇꢈꢉꢊꢋꢌꢍꢎꢏꢐꢑꢒꢓꢔꢕꢖꢗꢘꢙꢚꢛꢜꢝꢞꢟꢠꢡꢢꢣꢤꢥꢦꢧꢨꢩꢪꢫꢬꢭꢮꢯꢰꢱꢲꢳꢴꢵꢶꢷꢸꢹꢺꢻꢼꢽꢾꢿꣀꣁꣂꣃ꣄ꣅ꣆꣇꣈꣉꣊꣋꣌꣍꣎꣏꣐꣑꣒꣓꣔꣕꣖꣗꣘꣙꣚꣛꣜꣝꣞꣟꣠꣡꣢꣣꣤꣥꣦꣧꣨꣩꣪꣫꣬꣭꣮꣯꣰꣱ꣲꣳꣴꣵꣶꣷ꣸꣹꣺ꣻ꣼ꣽꣾꣿ꤀꤁꤂꤃꤄꤅꤆꤇꤈꤉ꤊꤋꤌꤍꤎꤏꤐꤑꤒꤓꤔꤕꤖꤗꤘꤙꤚꤛꤜꤝꤞꤟꤠꤡꤢꤣꤤꤥꤦꤧꤨꤩꤪ꤫꤬꤭꤮꤯ꤰꤱꤲꤳꤴꤵꤶꤷꤸꤹꤺꤻꤼꤽꤾꤿꥀꥁꥂꥃꥄꥅꥆꥇꥈꥉꥊꥋꥌꥍꥎꥏꥐꥑꥒ꥓꥔꥕꥖꥗꥘꥙꥚꥛꥜꥝꥞꥟ꥠꥡꥢꥣꥤꥥꥦꥧꥨꥩꥪꥫꥬꥭꥮꥯꥰꥱꥲꥳꥴꥵꥶꥷꥸꥹꥺꥻꥼ꥽꥾꥿ꦀꦁꦂꦃꦄꦅꦆꦇꦈꦉꦊꦋꦌꦍꦎꦏꦐꦑꦒꦓꦔꦕꦖꦗꦘꦙꦚꦛꦜꦝꦞꦟꦠꦡꦢꦣꦤꦥꦦꦧꦨꦩꦪꦫꦬꦭꦮꦯꦰꦱꦲ꦳ꦴꦵꦶꦷꦸꦹꦺꦻꦼꦽꦾꦿ꧀꧁꧂꧃꧄꧅꧆꧇꧈꧉꧊꧋꧌꧍꧎ꧏ꧐꧑꧒꧓꧔꧕꧖꧗꧘꧙꧚꧛꧜꧝꧞꧟ꧠꧡꧢꧣꧤꧥꧦꧧꧨꧩꧪꧫꧬꧭꧮꧯ꧰꧱꧲꧳꧴꧵꧶꧷꧸꧹ꧺꧻꧼꧽꧾ꧿ꨀꨁꨂꨃꨄꨅꨆꨇꨈꨉꨊꨋꨌꨍꨎꨏꨐꨑꨒꨓꨔꨕꨖꨗꨘꨙꨚꨛꨜꨝꨞꨟꨠꨡꨢꨣꨤꨥꨦꨧꨨꨩꨪꨫꨬꨭꨮꨯꨰꨱꨲꨳꨴꨵꨶ꨷꨸꨹꨺꨻꨼꨽꨾꨿ꩀꩁꩂꩃꩄꩅꩆꩇꩈꩉꩊꩋꩌꩍ꩎꩏꩐꩑꩒꩓꩔꩕꩖꩗꩘꩙꩚꩛꩜꩝꩞꩟ꩠꩡꩢꩣꩤꩥꩦꩧꩨꩩꩪꩫꩬꩭꩮꩯꩰꩱꩲꩳꩴꩵꩶ꩷꩸꩹ꩺꩻꩼꩽꩾꩿꪀꪁꪂꪃꪄꪅꪆꪇꪈꪉꪊꪋꪌꪍꪎꪏꪐꪑꪒꪓꪔꪕꪖꪗꪘꪙꪚꪛꪜꪝꪞꪟꪠꪡꪢꪣꪤꪥꪦꪧꪨꪩꪪꪫꪬꪭꪮꪯꪰꪱꪴꪲꪳꪵꪶꪷꪸꪹꪺꪻꪼꪽꪾ꪿ꫀ꫁ꫂ꫃꫄꫅꫆꫇꫈꫉꫊꫋꫌꫍꫎꫏꫐꫑꫒꫓꫔꫕꫖꫗꫘꫙꫚ꫛꫜꫝ꫞꫟ꫠꫡꫢꫣꫤꫥꫦꫧꫨꫩꫪꫫꫬꫭꫮꫯ꫰꫱ꫲꫳꫴꫵ꫶꫷꫸꫹꫺꫻꫼꫽꫾꫿꬀ꬁꬂꬃꬄꬅꬆ꬇꬈ꬉꬊꬋꬌꬍꬎ꬏꬐ꬑꬒꬓꬔꬕꬖ꬗꬘꬙꬚꬛꬜꬝꬞꬟ꬠꬡꬢꬣꬤꬥꬦ꬧ꬨꬩꬪꬫꬬꬭꬮ꬯ꬰꬱꬲꬳꬴꬵꬶꬷꬸꬹꬺꬻꬼꬽꬾꬿꭀꭁꭂꭃꭄꭅꭆꭇꭈꭉꭊꭋꭌꭍꭎꭏꭐꭑꭒꭓꭔꭕꭖꭗꭘꭙꭚ꭛ꭜꭝꭞꭟꭠꭡꭢꭣꭤꭥꭦꭧꭨꭩ꭪꭫꭬꭭꭮꭯ꭰꭱꭲꭳꭴꭵꭶꭷꭸꭹꭺꭻꭼꭽꭾꭿꮀꮁꮂꮃꮄꮅꮆꮇꮈꮉꮊꮋꮌꮍꮎꮏꮐꮑꮒꮓꮔꮕꮖꮗꮘꮙꮚꮛꮜꮝꮞꮟꮠꮡꮢꮣꮤꮥꮦꮧꮨꮩꮪꮫꮬꮭꮮꮯꮰꮱꮲꮳꮴꮵꮶꮷꮸꮹꮺꮻꮼꮽꮾꮿꯀꯁꯂꯃꯄꯅꯆꯇꯈꯉꯊꯋꯌꯍꯎꯏꯐꯑꯒꯓꯔꯕꯖꯗꯘꯙꯚꯛꯜꯝꯞꯟꯠꯡꯢꯣꯤꯥꯦꯧꯨꯩꯪ꯫꯬꯭꯮꯯꯰꯱꯲꯳꯴꯵꯶꯷꯸꯹꯺꯻꯼꯽꯾꯿가각갂갃간갅갆갇갈갉갊갋갌갍갎갏감갑값갓갔강갖갗갘같갚갛개객갞갟갠갡갢갣갤갥갦갧갨갩갪갫갬갭갮갯갰갱갲갳갴갵갶갷갸갹갺갻갼갽갾갿걀걁걂걃걄걅걆걇걈걉걊걋걌걍걎걏걐걑걒걓걔걕걖걗걘걙걚걛걜걝걞걟걠걡걢걣걤걥걦걧걨걩걪걫걬걭걮걯거걱걲걳건걵걶걷걸걹걺걻걼걽걾걿검겁겂것겄겅겆겇겈겉겊겋게겍겎겏겐겑겒겓겔겕겖겗겘겙겚겛겜겝겞겟겠겡겢겣겤겥겦겧겨격겪겫견겭겮겯결겱겲겳겴겵겶겷겸겹겺겻겼경겾겿곀곁곂곃계곅곆곇곈곉곊곋곌곍곎곏곐곑곒곓곔곕곖곗곘곙곚곛곜곝곞곟고곡곢곣곤곥곦곧골곩곪곫곬곭곮곯곰곱곲곳곴공곶곷곸곹곺곻과곽곾곿관괁괂괃괄괅괆괇괈괉괊괋괌괍괎괏괐광괒괓괔괕괖괗괘괙괚괛괜괝괞괟괠괡괢괣괤괥괦괧괨괩괪괫괬괭괮괯괰괱괲괳괴괵괶괷괸괹괺괻괼괽괾괿굀굁굂굃굄굅굆굇굈굉굊굋굌굍굎굏교굑굒굓굔굕굖굗굘굙굚굛굜굝굞굟굠굡굢굣굤굥굦굧굨굩굪굫구국굮굯군굱굲굳굴굵굶굷굸굹굺굻굼굽굾굿궀궁궂궃궄궅궆궇궈궉궊궋권궍궎궏궐궑궒궓궔궕궖궗궘궙궚궛궜궝궞궟궠궡궢궣궤궥궦궧궨궩궪궫궬궭궮궯궰궱궲궳궴궵궶궷궸궹궺궻궼궽궾궿귀귁귂귃귄귅귆귇귈귉귊귋귌귍귎귏귐귑귒귓귔귕귖귗귘귙귚귛규귝귞귟균귡귢귣귤귥귦귧귨귩귪귫귬귭귮귯귰귱귲귳귴귵귶귷그극귺귻근귽귾귿글긁긂긃긄긅긆긇금급긊긋긌긍긎긏긐긑긒긓긔긕긖긗긘긙긚긛긜긝긞긟긠긡긢긣긤긥긦긧긨긩긪긫긬긭긮긯기긱긲긳긴긵긶긷길긹긺긻긼긽긾긿김깁깂깃깄깅깆깇깈깉깊깋까깍깎깏깐깑깒깓깔깕깖깗깘깙깚깛깜깝깞깟깠깡깢깣깤깥깦깧깨깩깪깫깬깭깮깯깰깱깲깳깴깵깶깷깸깹깺깻깼깽깾깿

`posNum[T](implicit Numeric[T]): Gen[T]`

`posNum[Int]`

25, 6, 56, 19, 9, 86, 94, 8, 20, 68

`frequency[T]((Int, Gen[T])*): Gen[T]`

```
frequency(  
  2 -> Gen.choose('A', 'Z'),  
  3 -> Gen.choose(1, 10)  
)
```

3, C, 8, 2, 9, D, 6, U, S, 4

`choose[T](min:T, max:T)`
`(implicit Choose[T]):`
`Gen[T]`

`choose('a', 'z')`

`d, u, f, z, b, m, f, z, f, m`

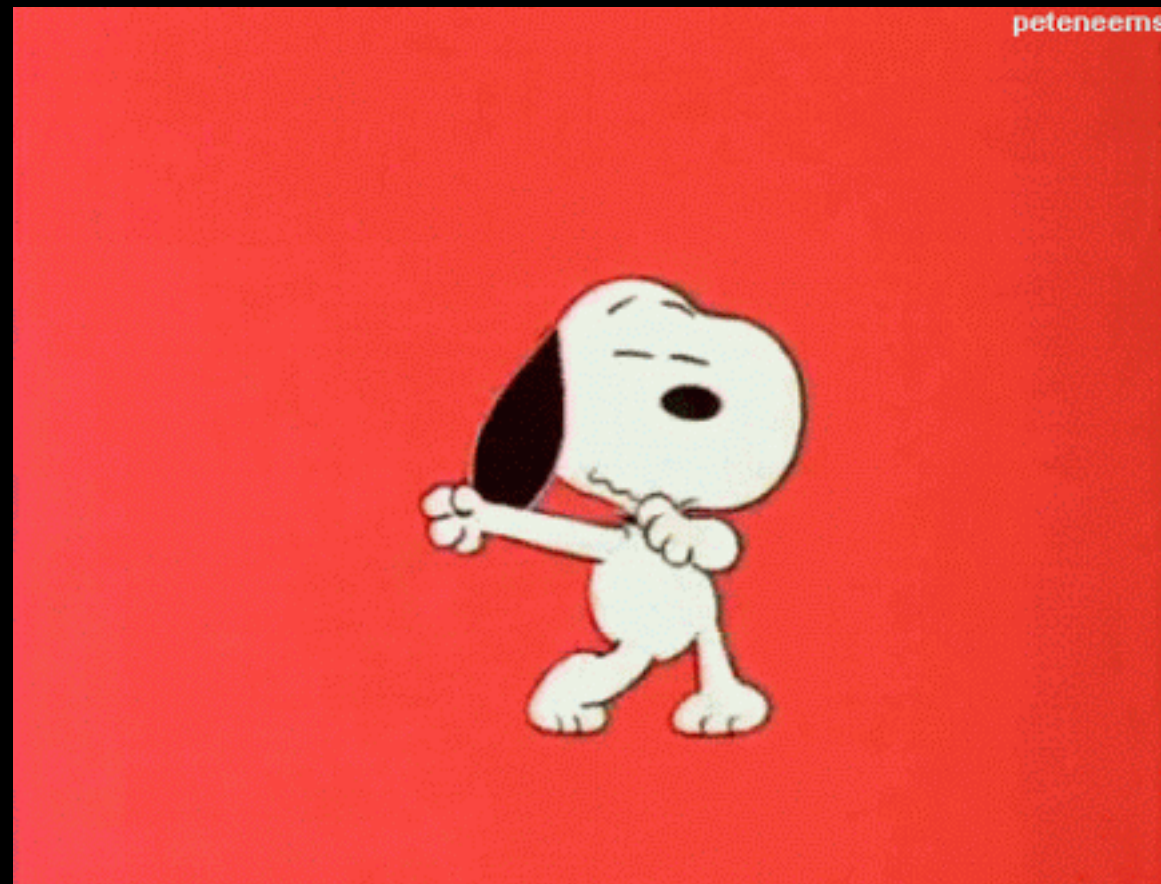
`Choose[Byte], Choose[Short], Choose[Char], Choose[Int], Choose[Long], Choose[Float],`
`Choose[Double],`

`option[T](Gen[T]): Gen[Option[T]]`

`option(Gen.posNum[Int])`

`None, Some(2), None, Some(67), None, None, None, Some(97), Some(3), None`

NameGenerator



There are **many** more

<http://bit.ly/2oxLFLV>

Can be used with EBT

implicitly[Gen[T]].sample.get

```
Shrink[A] {  
  def shrink(x:T): Stream[T]  
}
```

```
implicitly[Shrink[Int]].shrink(100)  
List(50, -50, 25, -25, 12, -12, 6, -6, 3, -3, 1, -1, 0)
```

[U]niversally Quantified Properties

=> Prop

=> Boolean

`forall[T I, P](gI: Gen[T I])`
`(T I \Rightarrow P)`

`(implicit p: (P) \Rightarrow Prop,`

`sl: Shrink[T I],`

`ppI: (T I) \Rightarrow Pretty)`

`:Prop`

forall[T I, P](T I \Rightarrow P)

(implicit p: (P) \Rightarrow Prop,

 a I: Arbitrary[T I],

 s I: Shrink[T I],

 pp I: (T I) \Rightarrow Pretty)

:Prop

Choosing Properties

Math



Laws

Associativity	$(a+b)+c$	$==$	$a+(b+c)$
Commutativity	$(a+b)$	$==$	$(b+a)$
Identity	$(a+\emptyset)$	$==$	a
	$(\emptyset+a)$	$==$	a
Distribution	$x(a+b)$	$==$	$xa+xb$

?ing



What does it **do**?

How is this **similar** to ...?

How is this **different** from ...?

Can I verify this **without** duplicating the CUT?

What will make it **fail**?

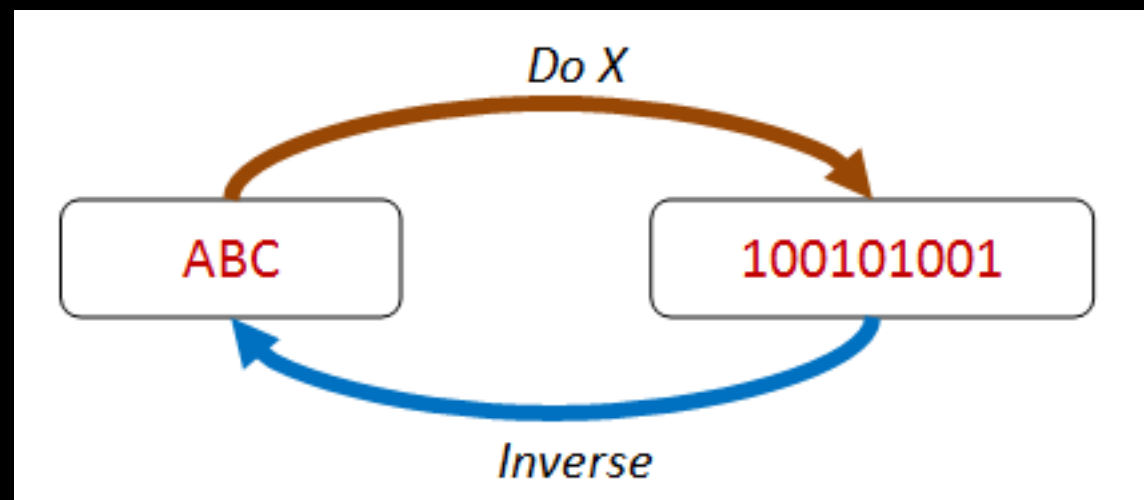
Patterns

Invariants

Length
Contents

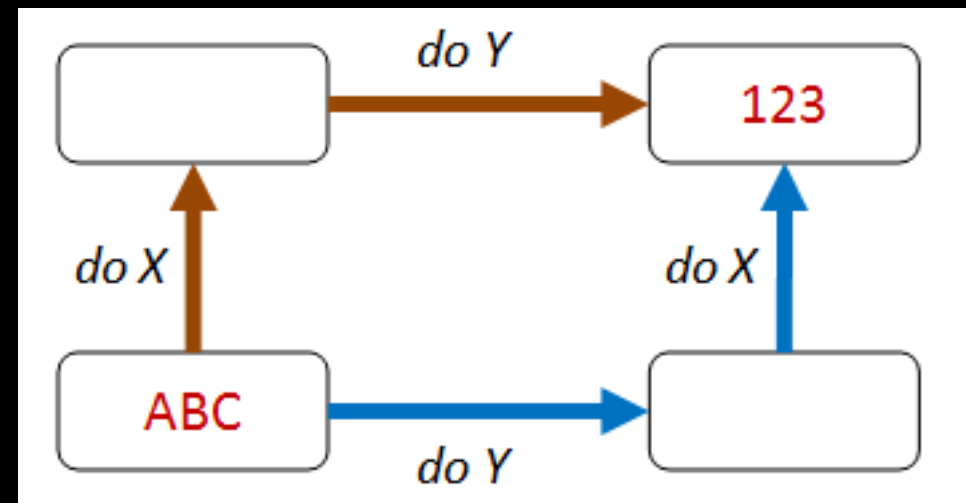
`list.sorted.length == list.length`

Round-tripping



`json.parse.toJson == json`

Different Order Same Result



`list.map(_ + 1).sorted == list.sorted.map(_ + 1)`

Compose Methods

```
list2.reverse ++ list1.reverse  
==  
(list1 ++ list2).reverse
```

Test Oracle

Verify against another implementation

multithreaded result == single-threaded result
jsonLibX result == jsonLibY result



There are others

<http://bit.ly/2o6DKsy>

Examples

Addition

ToAsciiUPPERCase

PBT's got your Back

```
[info] ! ToUpperCase.All non lowercase characters must be at the same positions: Falsified after 43 passed tests.  
[info] > ARG_0: "꺅"  
[info] > ARG_0_ORIGINAL: "램 다 굶 ㄴ뵤 襪 ㄹ꺅 簞 順 畧 ㄱ"
```

REA Robot

dets



Before



After



- Files over 1GB?
- Rehashing?
- > 6 weeks of effort!

- Database with *one* record!
- 5—6 calls to reproduce
- < 1 day to fix



Clojure/West

March 24-26 2014
The Palace Hotel San Francisco





Bug #4

Prefix:

```
open_file(dets_table, [{type, bag}]) --> dets_table  
close(dets_table) --> ok  
open_file(dets_table, [{type, bag}]) --> dets_table
```

Parallel:

1. `lookup(dets_table, 0)` --> []
2. `insert(dets_table, {0, 0})` --> ok
3. `insert(dets_table, {0, 0})` --> ok

Result: ok



March 24-26 2014

The Palace Hotel San Francisco



Summary

EBT

Easy to understand

Quick to implement

Good for implementations with few combinations

Needed for regression

Limited by developers imagination

Hard to test complex implementations

Boring to write

PBT

Edge cases for free

Hundreds of tests

Reusable Generators

More thinking involved

Good for complex implementations

Requires investment in learning techniques

More thinking involved

Have to write Generators/Shrinkers

Not good for regression



EBT + PBT = WIN



EBT

Basic cases

Regression (bugs)

PBT

Edge cases



Links

[The lazy programmer's guide to writing 1000's of tests - Scott Wlaschin](#)

[Practical Property-Based Testing - Charles O'Farrell](#)

[Property-Based Testing for Better Code - Jessica Kerr](#)

[Testing the Hard Stuff and Staying Sane - John Hughes](#)

Thank You!
