Balanced Sampling as a Tool for Useful PBT Random Tree Generation

Jim E. Newton
and Ghiles Ziat

EPITA Research Laboratory

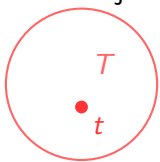
September 22, 2026

Motivation

- Property-Based Testing (PBT) requires generating objects of a target type.

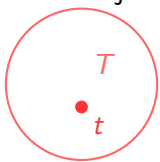
Motivation

- Property-Based Testing (PBT) requires generating objects of a target type.
- Select objects from an infinite set, T.



Motivation

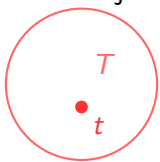
- Property-Based Testing (PBT) requires generating objects of a target type.
- Select objects from an infinite set, T.



- Selection should be *surjective*.
  I.e. any conceivable object of the target type should be possible.

Motivation

- Property-Based Testing (PBT) requires generating objects of a target type.
- Select objects from an infinite set, T.



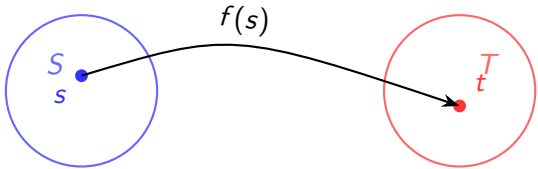
- Selection should be *surjective*.
  I.e. any conceivable object of the target type should be possible.
- Selection should be *uniform*. controversial
  I.e., any two objects are equally likely.

Motivation

- Selecting surjectively and uniformly in T may be difficult.

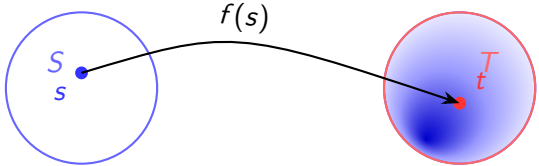
Motivation

- Selecting surjectively and uniformly in T may be difficult.
- We may prefer to sample in another space S.
- $f : S \to T$ ... and then convert object $s \in S$ to $f(s) = t \in T$.



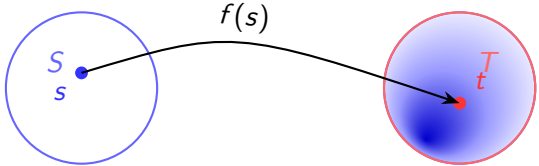
Motivation

- Sadly uniform selection in S may not deliver uniformity in T.



Motivation

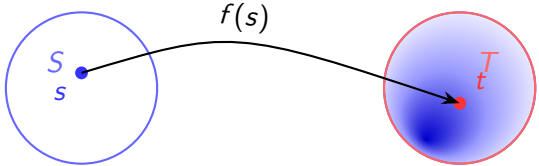
- Sadly uniform selection in S may not deliver uniformity in T.



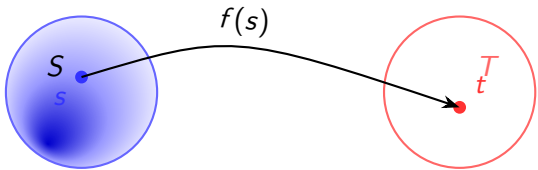
- OBSERVATION: When we select $s \in S$ then generate $f(s) \in T$ we often find that $f(s)$ is trivial.

Motivation

- Sadly uniform selection in S may not deliver uniformity in T.



- OBSERVATION: When we select $s \in S$ then generate $f(s) \in T$ we often find that $f(s)$ is trivial.
- GOAL: improve uniformity in T by biasing the selection in S.

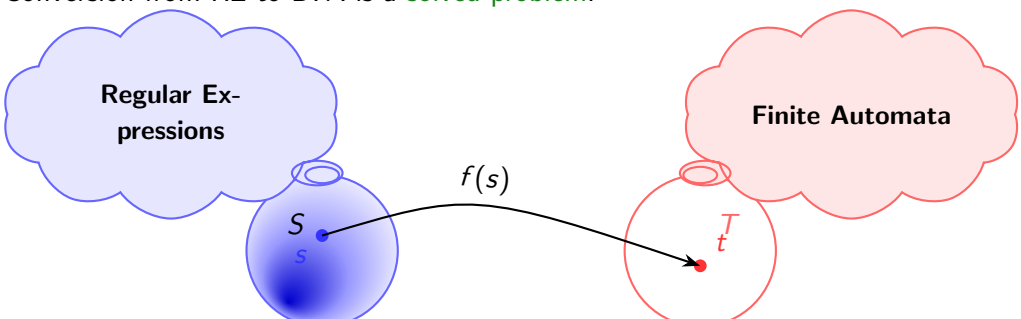


Motivation

CONCRETELY: uniformly sample Finite Automata by:

1. Biased sample of Regular Expressions
2. Convert Regular Expression to Finite Automaton

Uniform generation of Deterministic Finite Automata (DFA) is difficult.
Conversion from RE to DFA is a solved problem.



Overview

What are Regular Type Expressions (RTEs), viewed as ASTs.

How do RTEs relate to Deterministic Finite Automata (DFAs)?

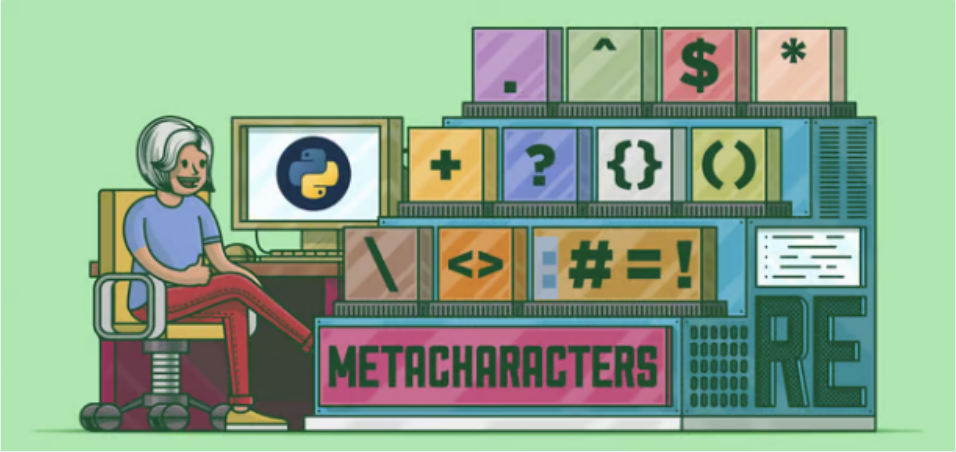
Generating random DFA

Algorithms for Balanced Generation

Experimental Results

Final Thoughts

What are Regular Type Expressions (RTEs), viewed as ASTs.



Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

[1, 2, 2.3, 9.3, 3, 1.5F, 6.5, 4.8F, 2, 2.3]

Regular Type Expressions (RTEs)

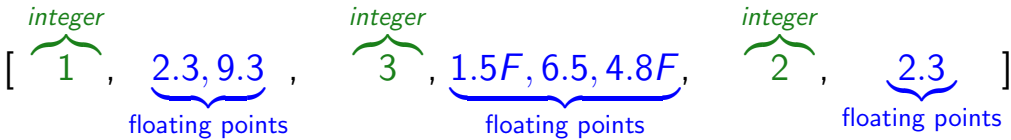
We'd like to recognize sequences with *regular patterns*.

[1, 2, 2.3, 9.3, 3, 1.5F, 6.5, 4.8F, 2, 2.3]

What's the pattern?

Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.



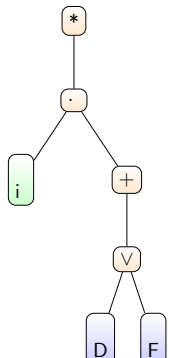
What's the pattern?

Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.

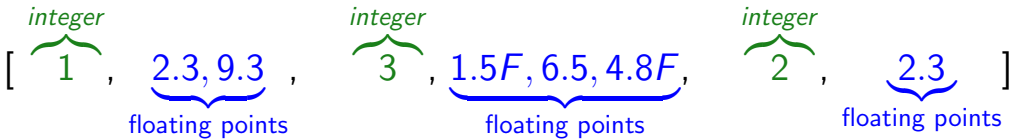
String-based regular expressions

- Match strings like: `"iDDiFDFiD"`,
- ... we use surface syntax: `"(i(D|F)+)*"`.
- ... representing expression: $(i \cdot (D \vee F)^{+})^{*}$,



Regular Type Expressions (RTEs)

We'd like to recognize sequences with *regular patterns*.



Type-based regular expressions

$$(Integer \cdot (Double \vee Float)^{+})^{*}$$

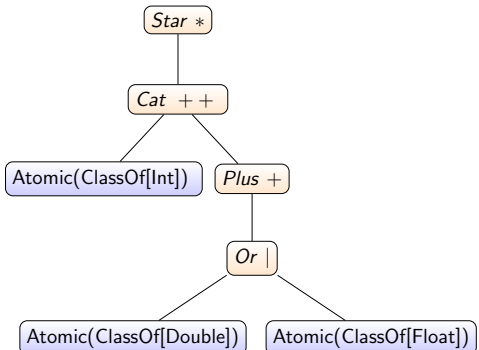
RTEs

Surface Syntax and AST

- Mathematical notation:
  $(Int \cdot (Double \cup Float)^{+})^{*}$
- Scala notation: AST

```
1  val I:Rte = Atomic(classOf[Int])
2  val F:Rte = Atomic(classOf[Float])
3  val D:Rte = Atomic(classOf[Double])
4
5  val re:Rte = (I ++ (D | F).+ ).*
```

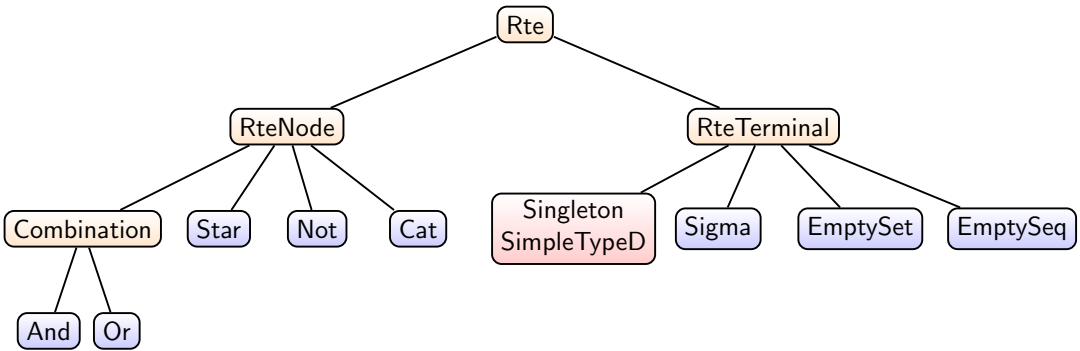
- Leaf nodes interface to Scala Type System



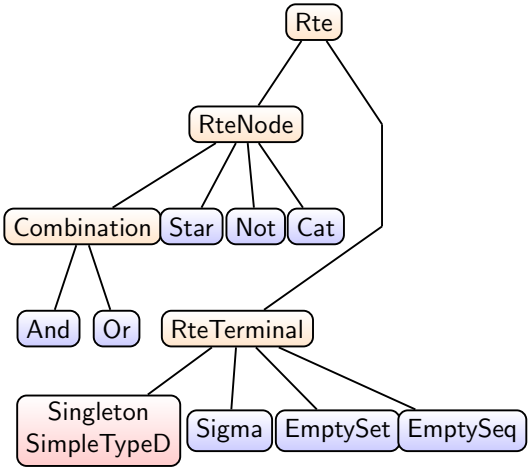
RTE Library

- RTE operations include union, intersection, complement, Kleene-star
- Habitation, Subset, Emptiness checks
- Membership check (RE-matching)
- Need to test RTE library
  - Random generation of RTE/AST

RTE class quasi-ADT



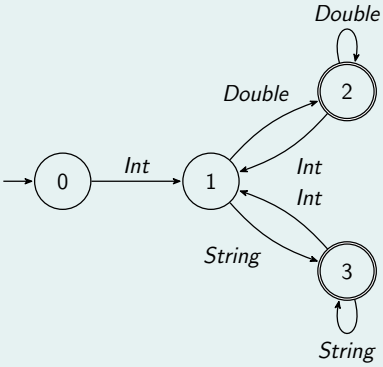
Generate a Random RTE/AST (depth-based)




```
def genRte(depth: Int): Rte = {
 val terms= Seq(Sigma,EmptySeq,EmptySet)
 def rte = genRte(depth-1)
 val generators: = Seq(
      () => randomChoice(terms),
      () => Singleton(genType()),
      () => Or(rte(), rte()),
      () => And(rte(), rte()),
      () => Cat(rte()),
      () => Star(rte()),
      () => Not(rte()))

 if (depth <= 0)
   Singleton(genType())
 else
   randomChoice(generators)()}
```

How do RTEs relate to Deterministic Finite Automata (DFAs)?



RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

Does the sequence follow the pattern? $(Int \cdot (Double^{+} \vee String^{+}))^{+}$

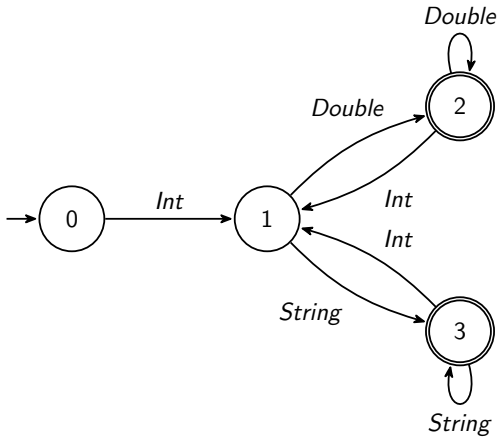
RTE Matching

How does a pattern predicate work?

`[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]`

Does the sequence follow the pattern? $(Int \cdot (Double^+ \vee String^+))^+$

We construct a *deterministic* finite automaton (DFA).

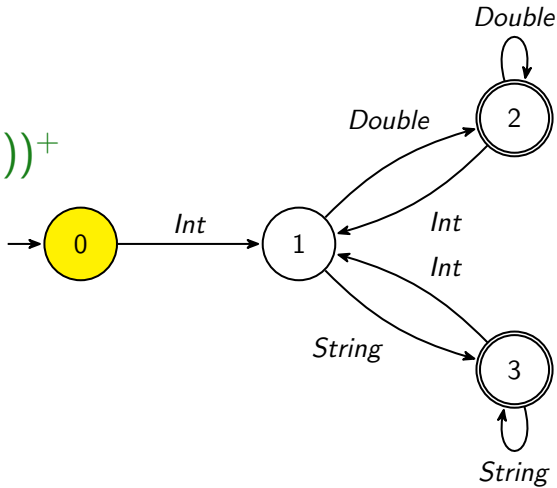


RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



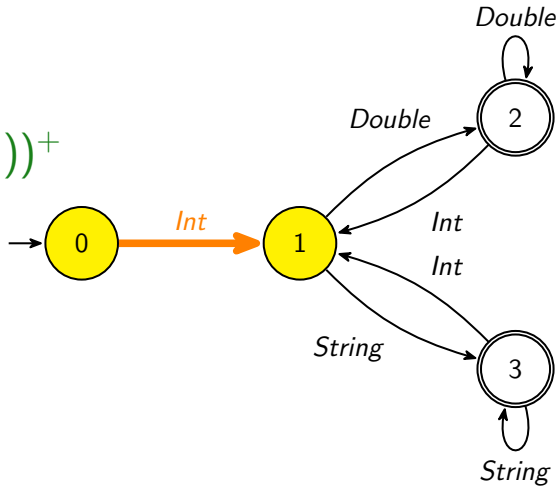
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



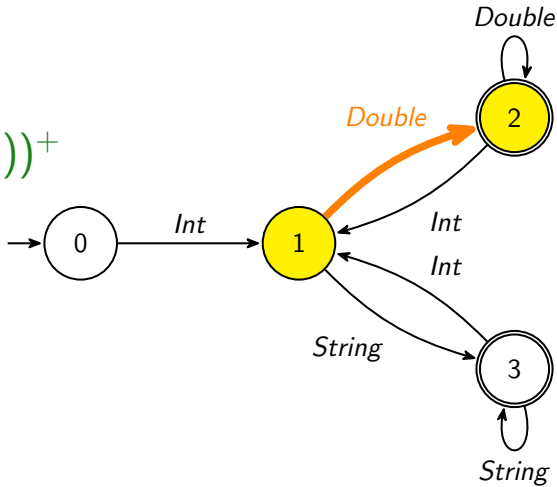
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



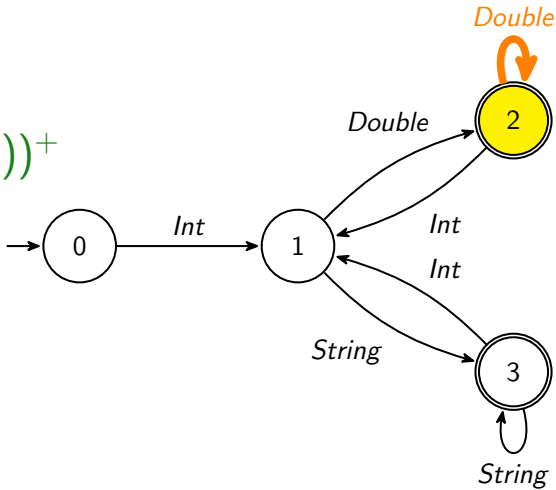
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



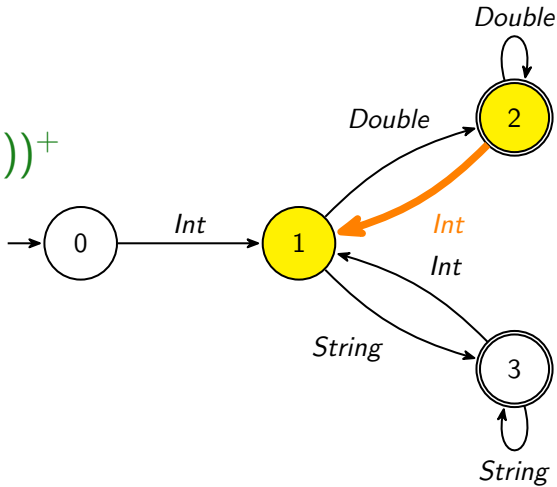
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



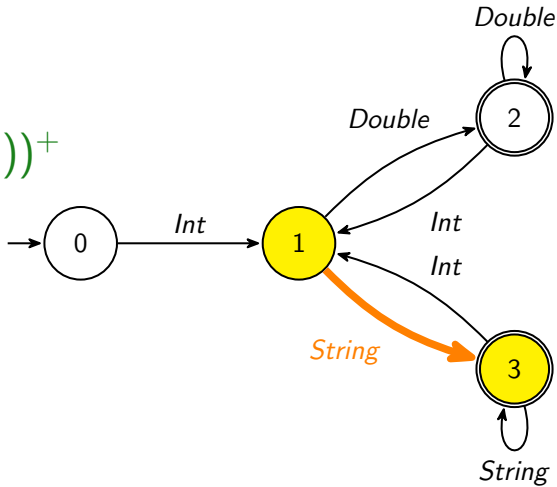
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



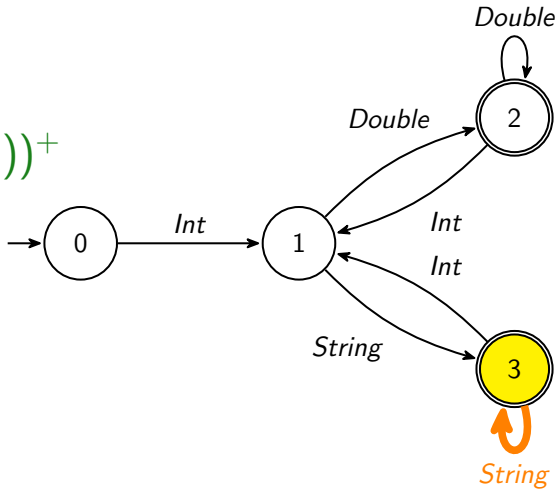
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$



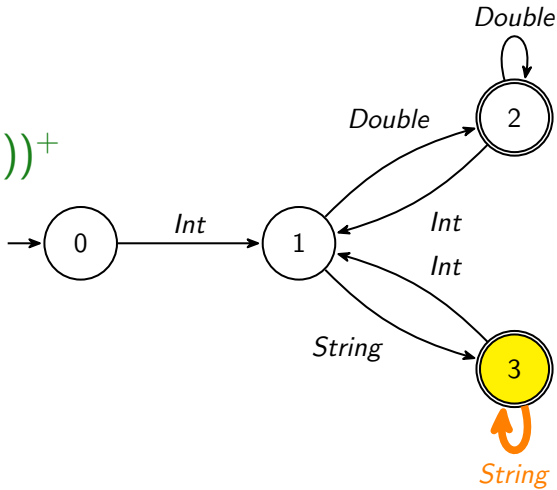
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



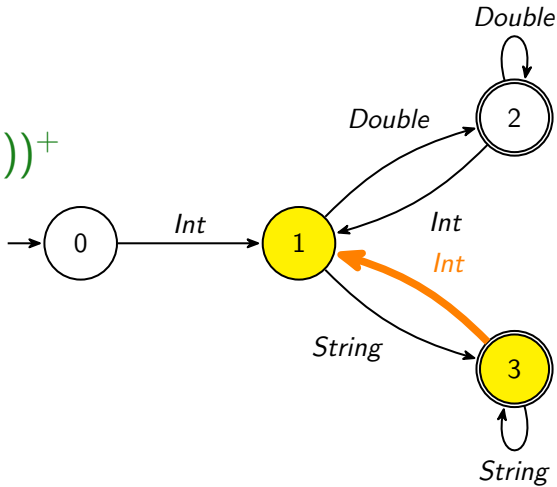
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



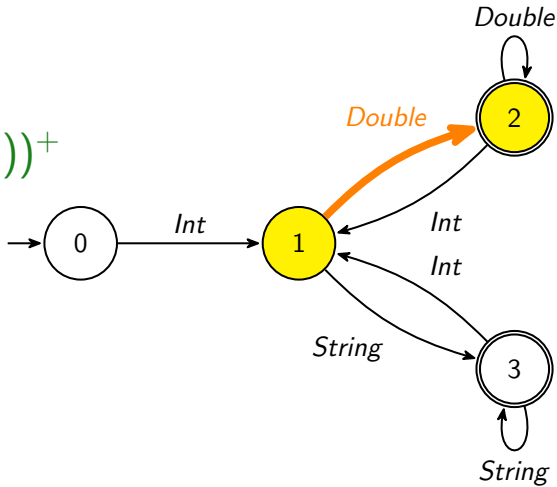
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$(Int \cdot (Double^{+} \vee String^{+}))^{+}$



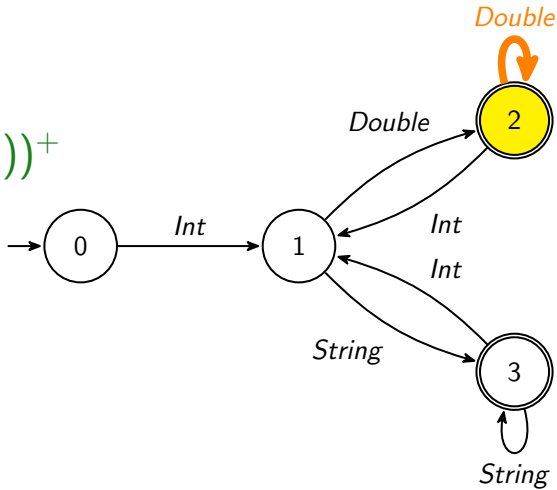
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



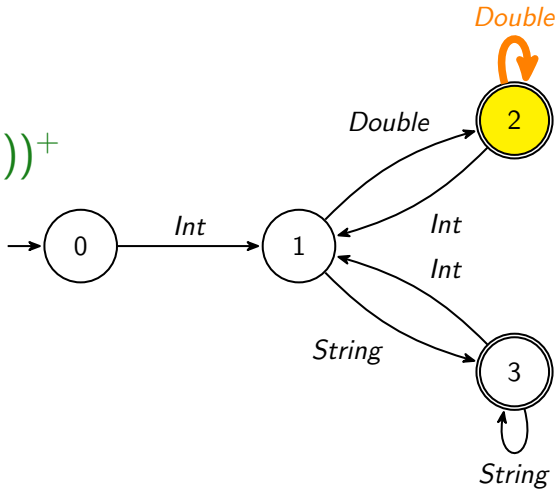
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



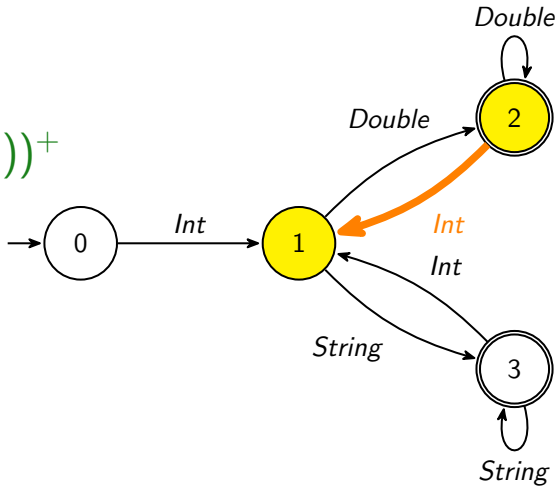
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$



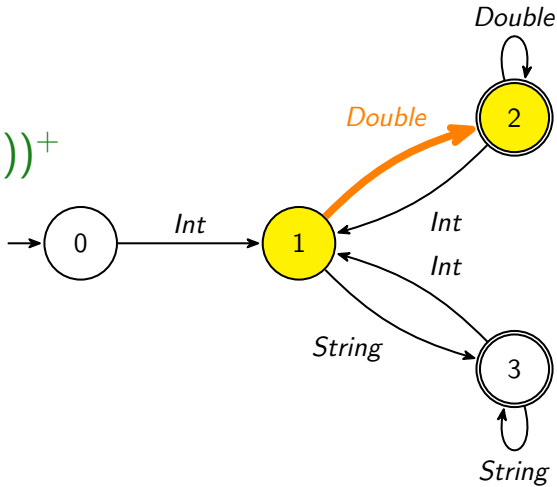
Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

$$(Int \cdot (Double^{+} \vee String^{+}))^{+}$$

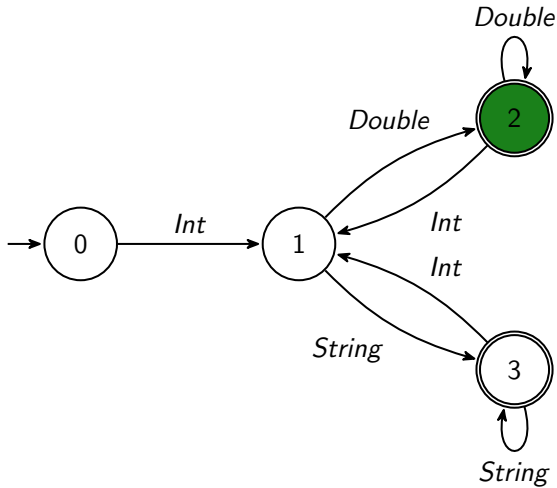


Does it match?

RTE Matching

How does a pattern predicate work?

```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```

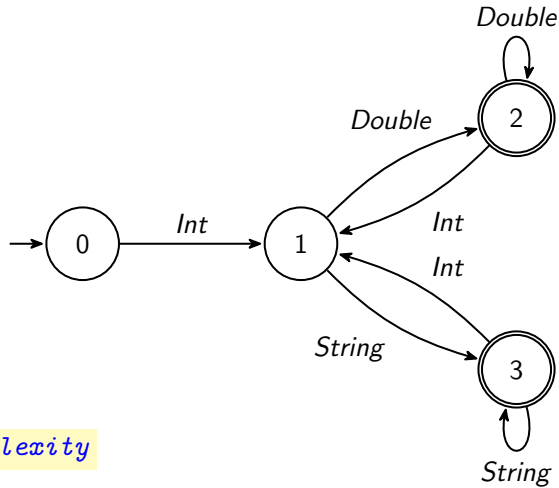


Yes, it's a match!

RTE Matching

How does a pattern predicate work?

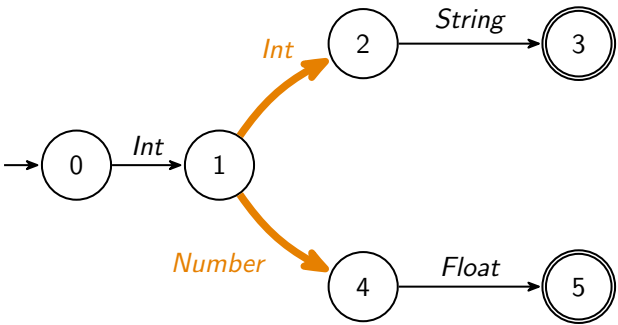
```
[13 2.0 6.0 4 "a" "an" "the" -5 2.0 3.0 4.0 7 8.0]
```



Decision procedure is $O(n)$,
independent of syntactical complexity
of the RTE.

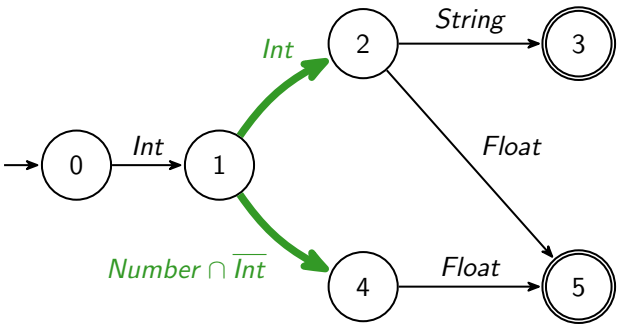
Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`



Deterministic (DFA) vs Non-deterministic (NFA)

Suppose sequence = `[2, 3, 5.6F]`



DFA Library `xymbolyco`

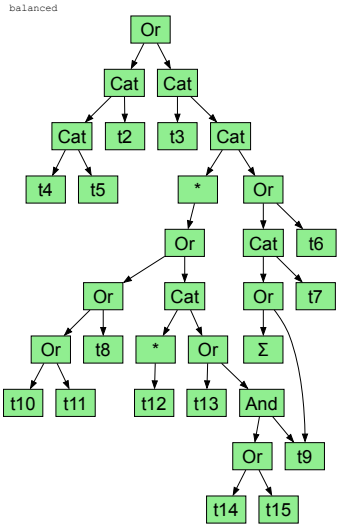
Symbolic Finite Automata Library

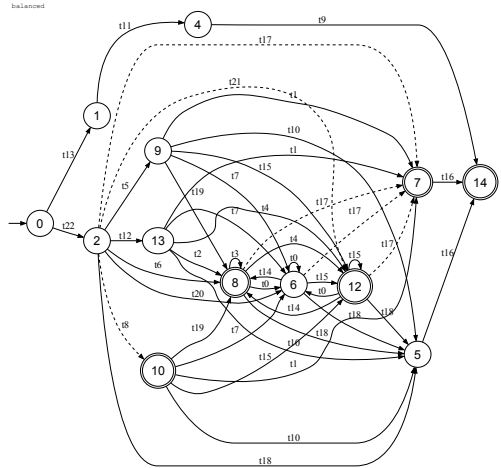
- RTE to DFA
- DFA to RTE
- Pattern matching
- Minimization
- Union, Intersection, Emptiness check
- Need to test `xymbolyco`
  - Random generation of DFA

Generating random DFA

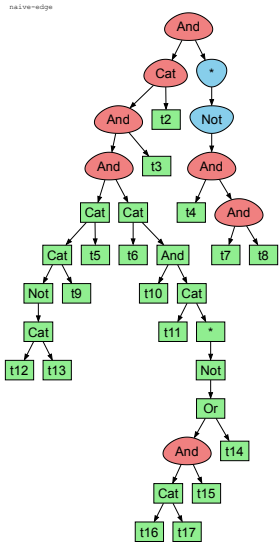
- We can construct DFA from RTE
- ... so generate random RTE (as seen earlier)
- ... and construct DFA

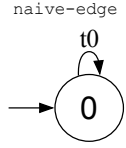
Successful: Generation of DFA from RTE





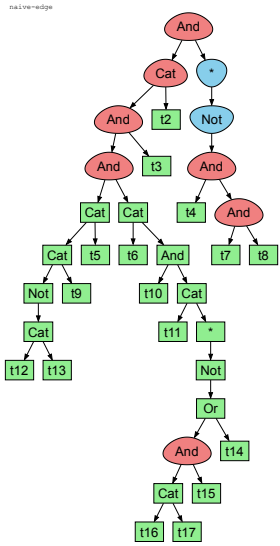
Dubious: Generation of DFA from RTE

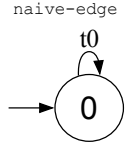




- Trivial DFA.

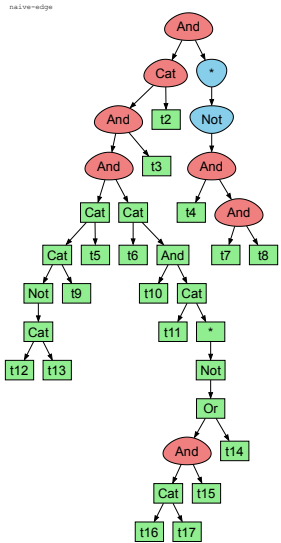
Dubious: Generation of DFA from RTE

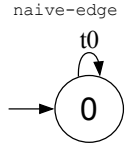




- Trivial DFA.
- Non-uniformity caused by so-called annihilators

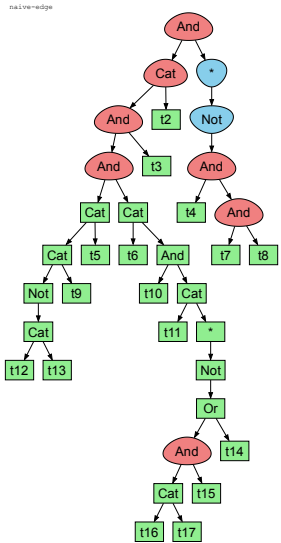
Dubious: Generation of DFA from RTE

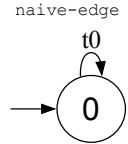




- Trivial DFA.
- Non-uniformity caused by so-called annihilators
- Whenever an input $X = \emptyset$ then $X \cap Y = \emptyset$.

Dubious: Generation of DFA from RTE



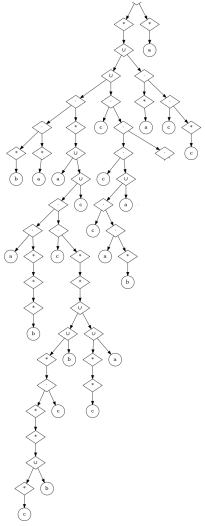


- Trivial DFA.
- Non-uniformity caused by so-called annihilators
- Whenever an input $X = \emptyset$ then $X \cap Y = \emptyset$.
- Thus entire tree generating Y is in vain.

Proposed fix: Balanced Tree Generation

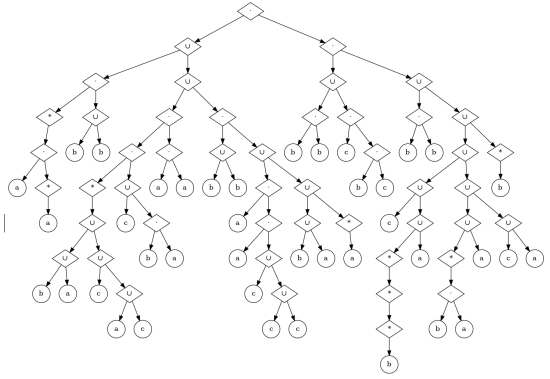
UNIFORM

Portrait — narrow and deep



BALANCED

Landscape — shallow and wide



Requirements

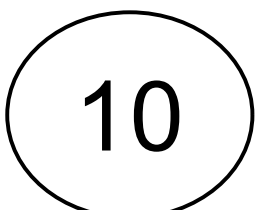
- If we want to generate *large* DFAs, we necessarily should generate *large* RTE (AST).
- However, this is not sufficient. Many large RTEs generate small/trivial DFAs.
- We must generate Portrait not Landscape aspect ratios.

Algorithms for Balanced Generation

Tree-Split Algorithm

Divide and Conquer

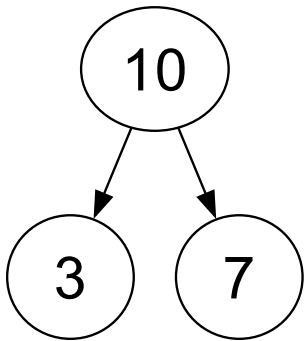
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

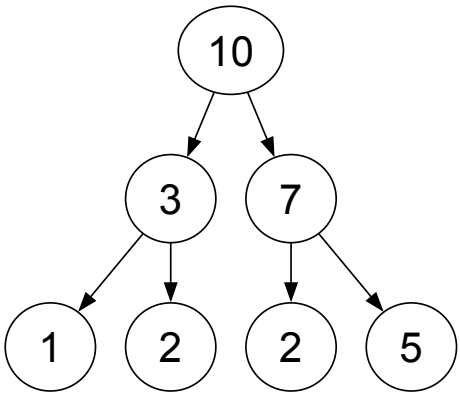
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

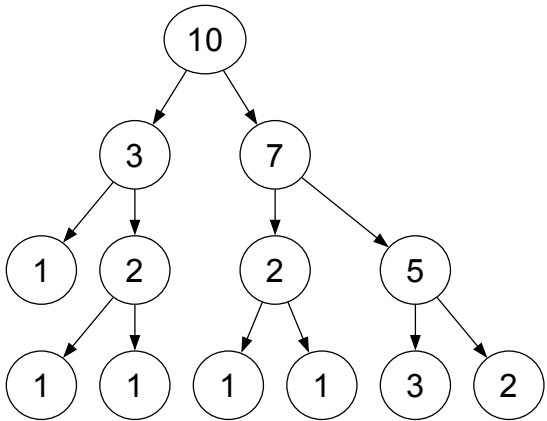
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

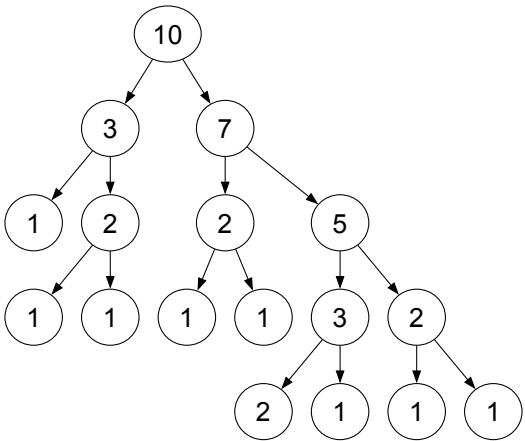
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

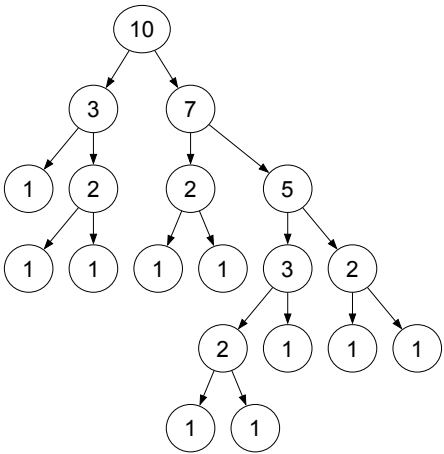
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

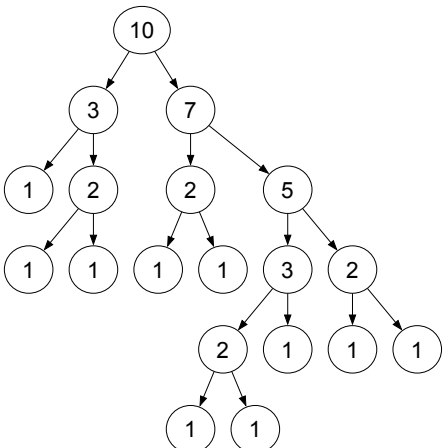
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Question is how to produce the pivot?

- Gaussian
- Inverse Gaussian
- Linear
- Lopsided Comb (*de guingois*, *traviole*)

Rte Tree-Split Generator

```
1  def treeSplitRte(leaves:Int, pivot:Int => Int):Rte =
2    if (leaves == 1)
3      randElement(inhabitedLeaves)
4    else {
5      val leftSize = pivot(leaves)  // choose 1 <= N < leaves
6
7      lazy val left  = treeSplitRte(leftSize, pivot)
8      lazy val right = treeSplitRte(leaves - leftSize, pivot)
9      lazy val mid   = treeSplitRte(leaves, pivot)
10
11     weightedCase((20, () => Cat(left, right)),
12                  (30, () => And(left,right)),
13                  (30, () => Or(left,right)),
14                  (5,  () => Star(mid)),
15                  (15, () => Not(mid)))}
```

Rte Tree-Split 4-variants

```
1  // lopsided comb
2  def treeCombRte(leaves:Int):Rte =
3    treeSplitRte(leaves, (n) => {
4      if (n == 2 || n == 3) 1
5      else                  2  })
6
7  // linear split
8  def treeSplitLinear(leaves:Int):Rte =
9    treeSplitRte(leaves, (n) => between(1, n))
10
11 // Gaussian split
12 def treeSplitGaussian(leaves:Int):Rte =
13   treeSplitRte(leaves, (n)=> between(1, n/2) + between(1, n/2))
14
15 // inverse Gaussian split
16 def treeSplitInvGaussian(leaves:Int):Rte =
17   treeSplitRte(leaves, (n) => invertedGaussian(1, n))
```

Flajolet Algorithm

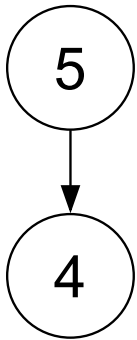
Generate a Skeleton

Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`

Flajolet Algorithm

Generate a Skeleton

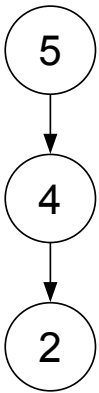
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

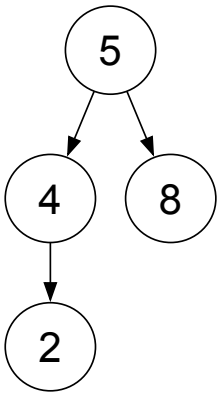
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

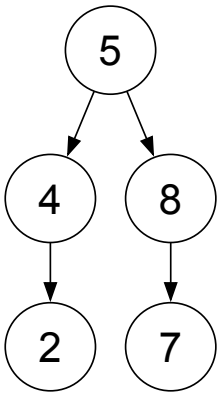
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

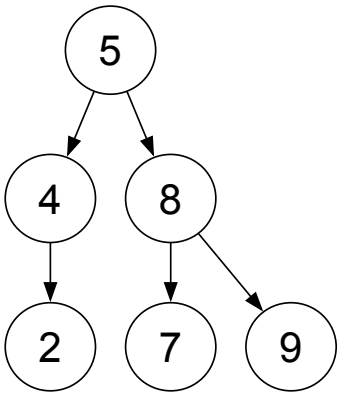
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

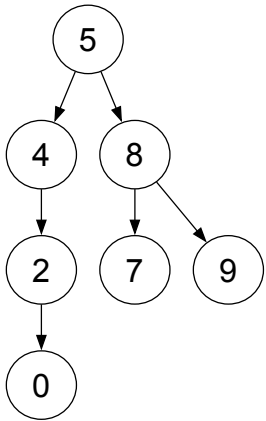
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

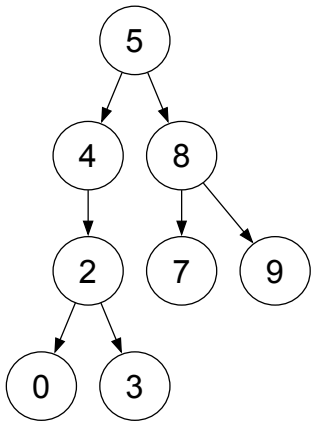
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

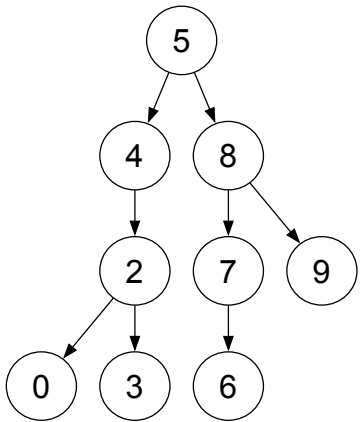
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

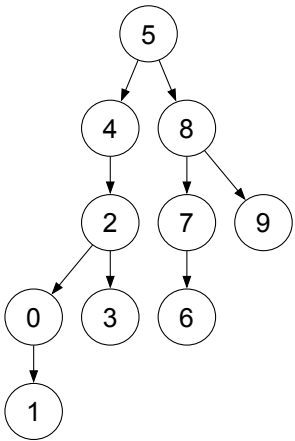
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

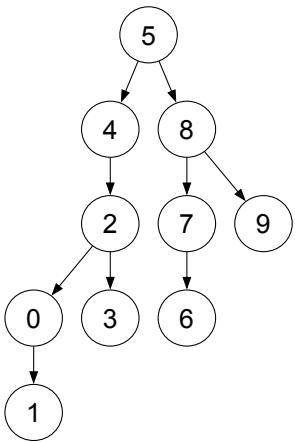
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`

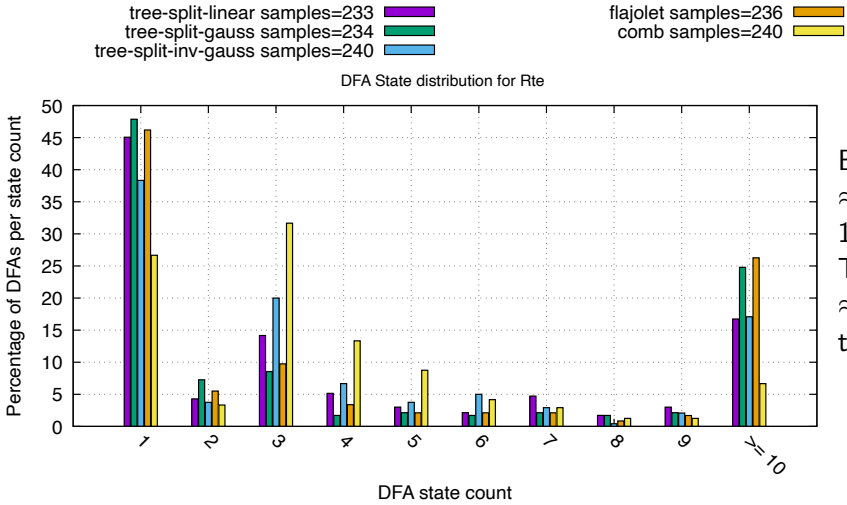


After generating skeleton, we fit an RTE to that skeleton.

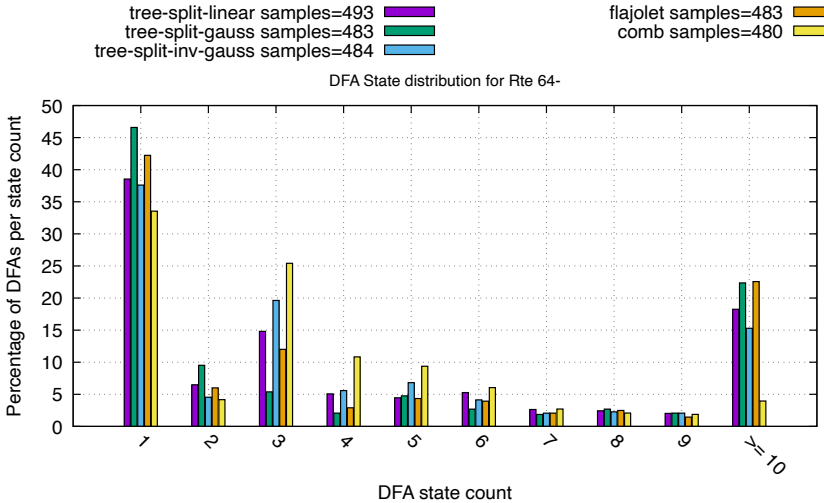
RTE Flajolet Generator

```
1  def flajoletRte(leaves:Int):Rte =
2    def recur(node:Node):Rte =
3      weightedCase(
4        (5, () => Star(recur(node))),
5        (5, () => Not(recur(node)))),
6        (90, () => node match {
7          case InternalNode(_,left,right) =>
8            weightedCase((34, () => Cat(recur(left), recur(right))),
9                         (33, () => And(recur(left), recur(right))),
10                        (33, () => Or( recur(left), recur(right))))
11         case LeafNode() =>
12           randElement(inhabitedLeaves)      }))
13
14   val skeleton = balancedRandTree(leaves - 1)
15   recur(skeleton)
```

Experimental Results

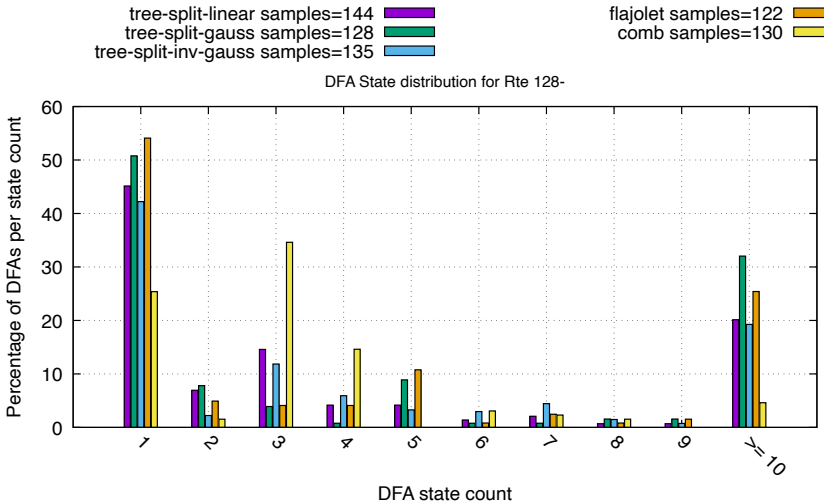


Balanced sampling results $\approx$ 24 to 26% of DFAs with 10 states or larger. Tree-split (uneven) results $\approx$ 27%. Worst performing is the comb distribution.



Restricted to RTEs with exactly 64 leaf nodes.

Balanced sampling results $\approx$ 22 to 23% of DFAs with 10 states or larger. Uniform sampling results in range 20% to 23.



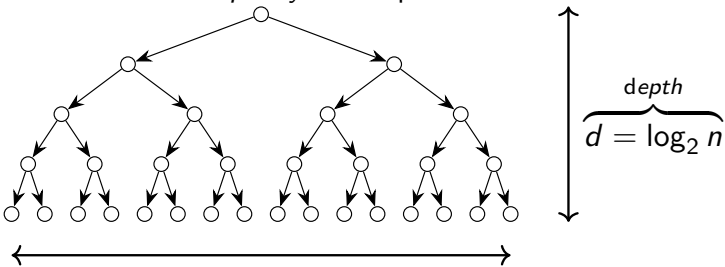
Restricted to larger RTEs with exactly 128 leaf nodes.

Balanced sampling results $\approx$ 20% to 32 of DFAs with 10 states or larger.

Imbalanced, results $< 5\%$

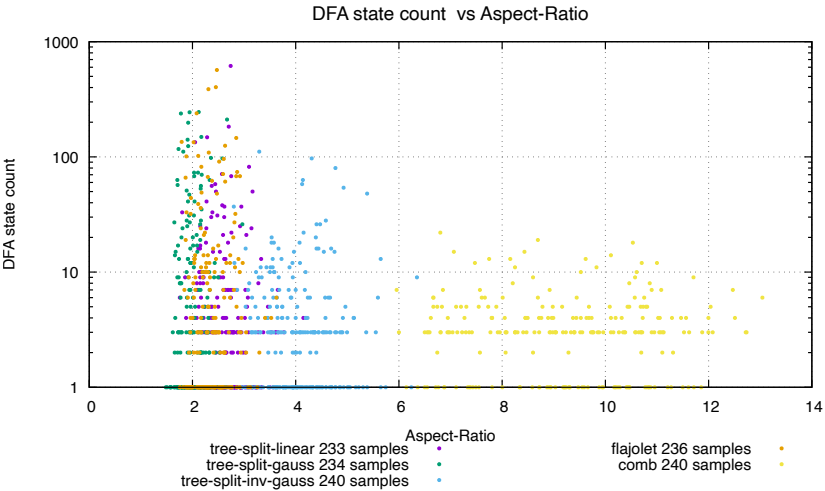
How to measure aspect-ratio of a binary tree

CLAIM: a *Landscape-style* AST performs better than *Portrait-style*.



$$\rho = \text{aspect ratio} = \frac{d}{\log_2 n} = \frac{\text{longest path}}{\log_2(\text{leaf count})}$$

$$\rho = 1 \implies \text{balanced}$$
$$\rho < 1 \implies \text{landscape}$$
$$\rho > 1 \implies \text{portrait}$$



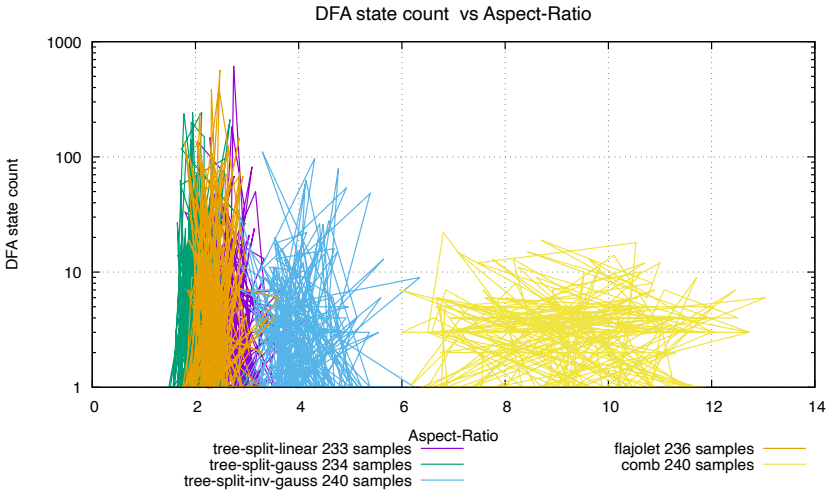
DFA state count
vs Aspect Ratio

$$\rho = \frac{longest\ path}{\log_2(leaf\ count)}$$

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



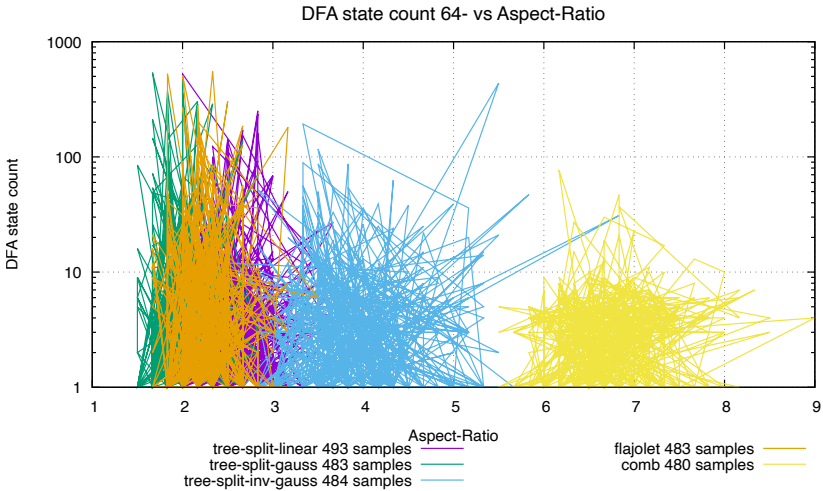
DFA state count
vs Aspect Ratio

$$\rho = \frac{longest\ path}{\log_2(leaf\ count)}$$

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



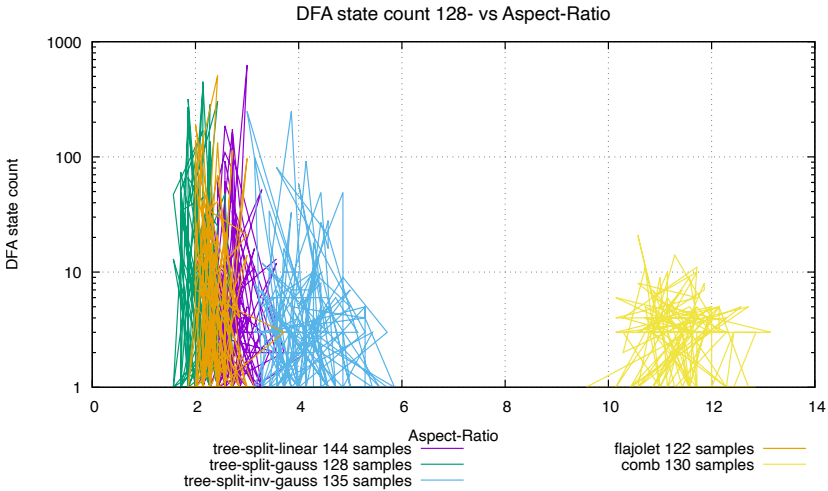
DFA state count
vs Aspect Ratio

$$\rho = \frac{longest\ path}{\log_2(leaf\ count)}$$

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



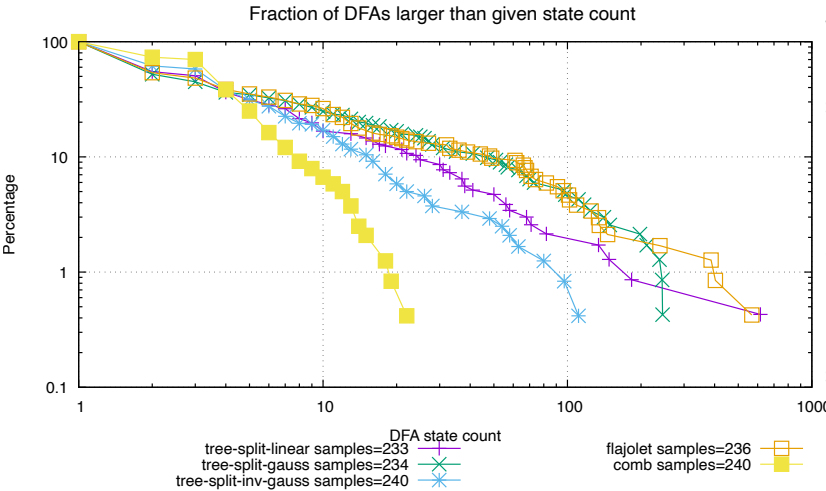
DFA state count
vs Aspect Ratio

$$\rho = \frac{longest\ path}{\log_2(leaf\ count)}$$

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

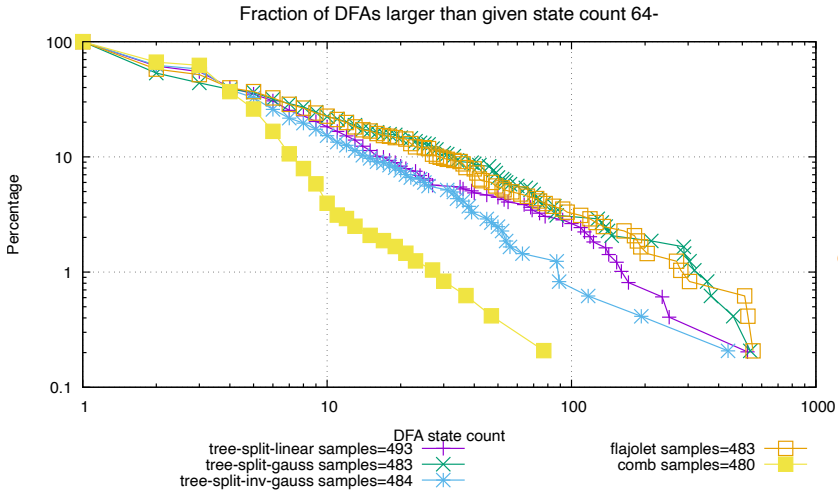
$\rho > 1 \implies$ portrait



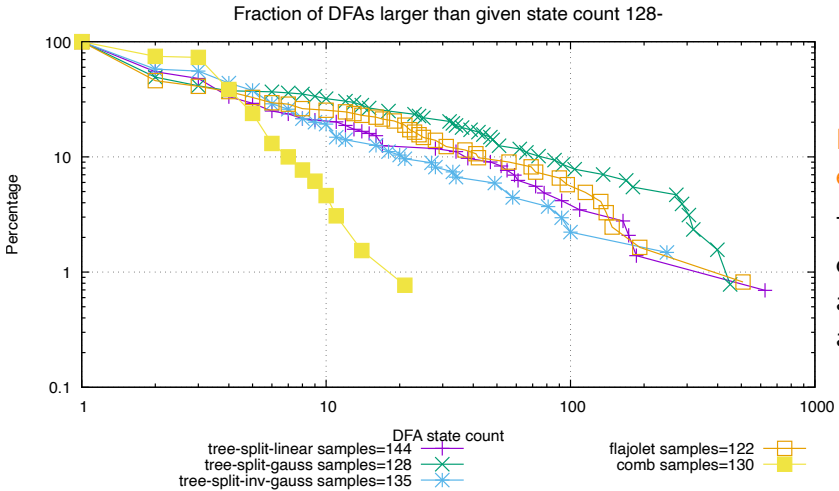
Which percent of all DFAs have larger than x-many states?

Flajolet and tree-split-mid (Gaussian) produces 10% of DFAs larger than 50 states. Tree-split-edge (inverted Gaussian) and tree-split-linear produces 11% larger than 10 states.

Imbalanced produces 1% larger than 11 states.

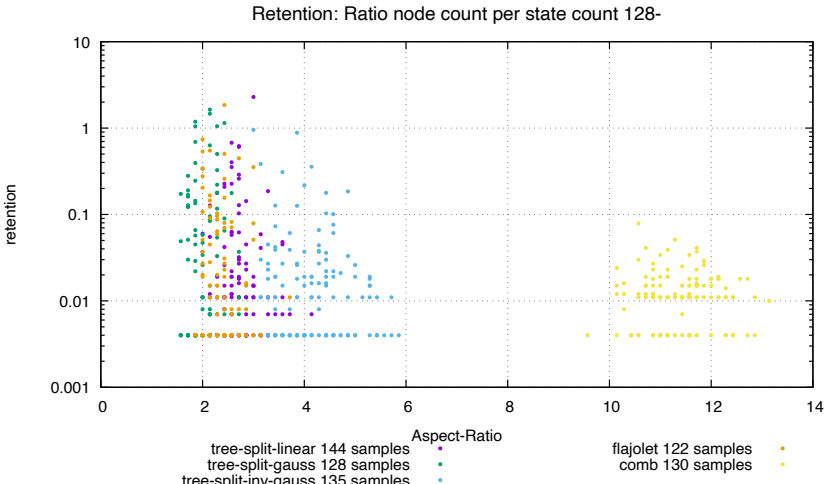


Restricted to RTEs with exactly 64 leaf nodes.



Restricted to RTEs with exactly 128 leaf nodes.

There seems to be some convergence of behavior for all the somewhat balanced algorithms for large RTEs.



Restricted to RTEs having exactly 128 leaf nodes.

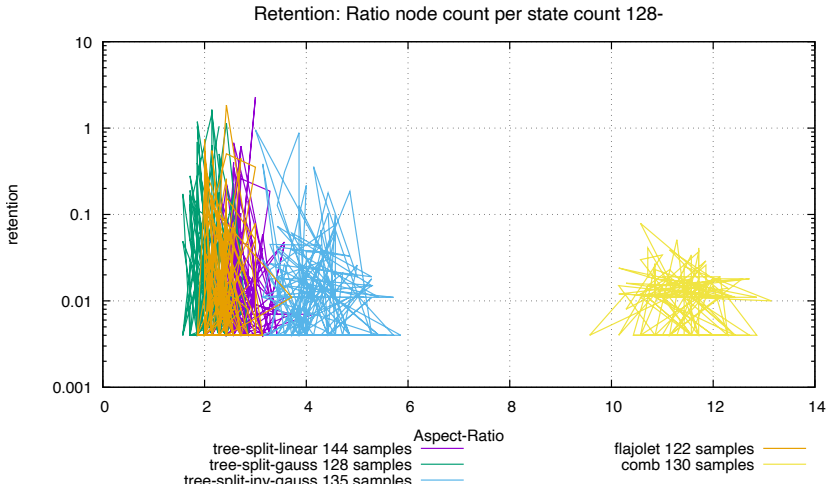
Retention measures resistance to lossage.

High lossage means large RTEs result in small DFAs.

High retention means more DFAs are large (don't *lose* as many states).

$\rho = \frac{state\ \ count}{node\ \ count}$

$\rho > 1$ is good.



Restricted to RTEs having exactly 128 leaf nodes.

Retention measures resistance to lossage.

High lossage means large RTEs result in small DFAs.

High retention means more DFAs are large (don't *lose* as many states).

$\rho = \frac{state\ \ count}{node\ \ count}$

$\rho > 1$ is good.

Summary

Perspectives

Our experiements do not show a direct correlation between aspect-ratio and lossage as we predicted.
There could be many reasons for this.

- need a better way to measure aspect ratio
- secondary effects of RE operations NOT/STAR have more drastic effects that our intuition indicates
- RE algebra has more annihilators that numerical arithmetic

Conclusion

- PBT often requires AST generation, so we hope our research can be applied far beyond RTE/DFA generation.
- Balanced AST generation improves uniformity in the *semantic* space.
- Uniform AST generation gives uniformity in the *syntactic* space.
- All code publically available: `https://github.com/jimka2001/scala-rte`
- Project also includes Common Lisp, Clojure, and Python implementations.
- Ongoing research to be presented at Clojure Conj 2025.`clojure-conj.org` (November 2024)

Questions and Answers