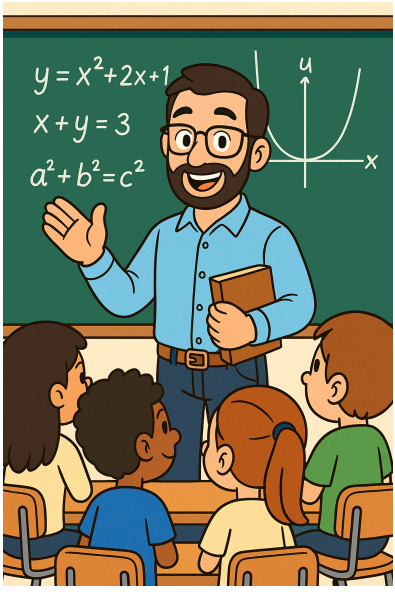
Balanced Sampling and Useful Random Tree Generation

Jim E. Newton
and Ghiles Ziat

EPITA Research Laboratory

September 22, 2026

Who am I?



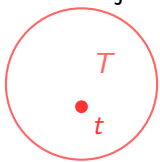
Research Professor at
EPITA School of Engineering and Computer Science
Paris, FRANCE

Motivation

- Testing (and in particular Property-Based Testing, PBT) requires generating randomized objects of a target type.

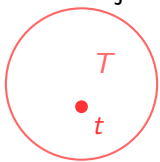
Motivation

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



Motivation

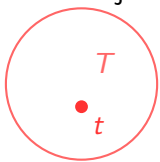
- Testing (and in particular Property-Based Testing, PBT) requires generating randomized objects of a target type.
- Select objects from an infinite set, T.



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

Motivation

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



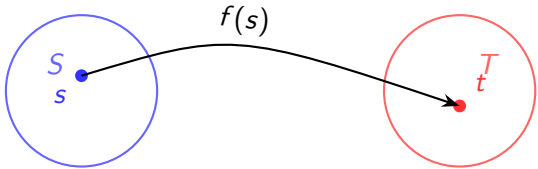
- Selection should be *surjective*.
  I.e. any conceivable object of the target type should be possible.
- Selection should be *uniform*. controversial
  I.e., any two objects are equally likely.

Motivation

- Selecting surjectively and uniformly in T may be difficult.

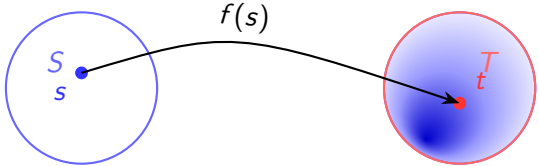
Motivation

- 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 $f(s) = t \in T$.



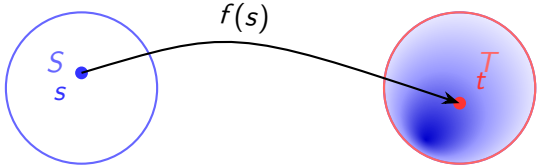
Motivation

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



Motivation

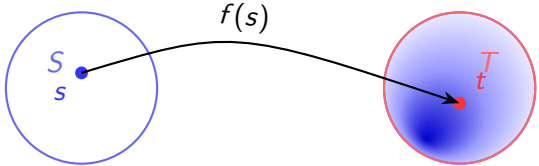
- Sadly uniform selection in S may not deliver uniformity in T.



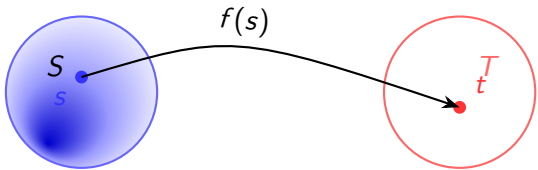
- OBSERVATION: When we select $s \in S$ then generate $f(s) \in T$ we often find that $f(s)$ is trivial.

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.

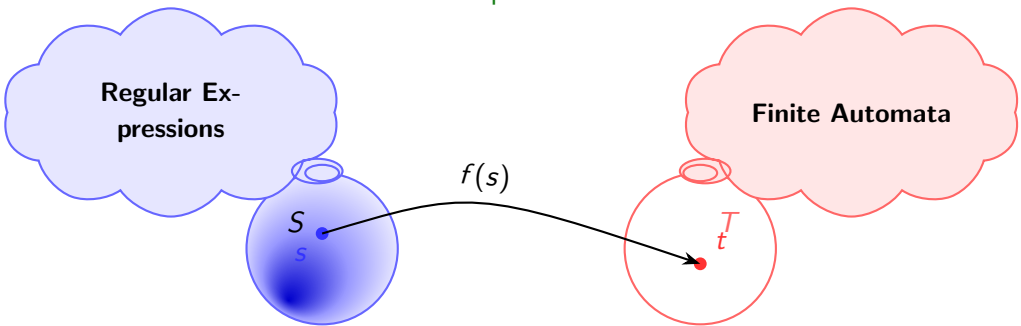


Motivation

CONCRETELY: uniformly sample Finite Automata by:

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

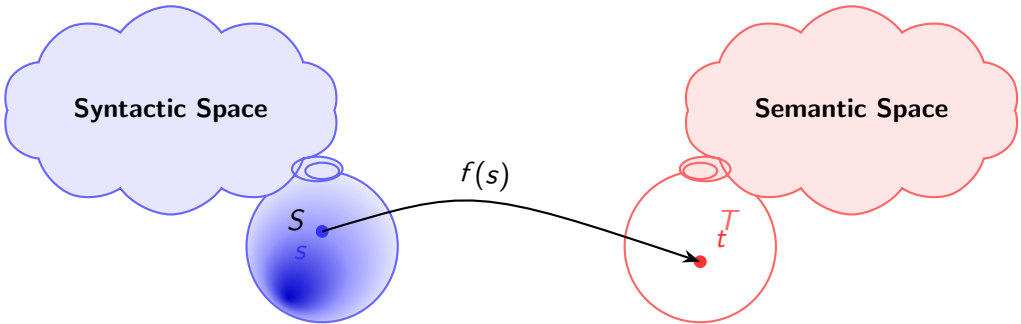
Uniform generation of Deterministic Finite Automata (DFA) is difficult.
Conversion from RE to DFA is a solved problem.



Generalization

Uniformly sample the *Semantic Space*:

- By biased sampling in the *Syntactic Space*



Overview

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

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

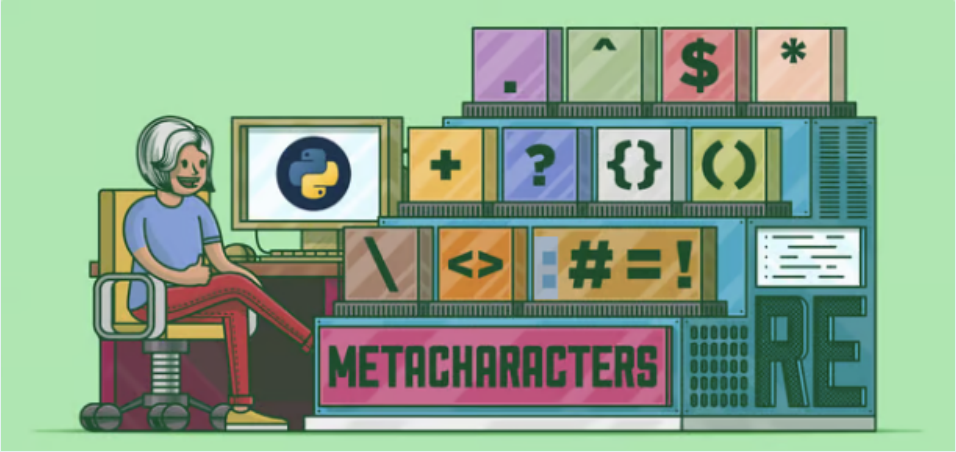
Generating random DFA

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*.

$$[\ 1, \quad 2/3, \quad 9.3, \quad 3, \quad 1.5, \quad 6.5, \quad 4.8, \quad 2, \quad 2/3]$$

Regular Type Expressions (RTEs)

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

$$[\ 1,\quad 2/3,\quad 9.3,\quad 3,\quad 1.5,\quad 6.5,\quad 4.8,\quad 2,\quad 2/3]$$

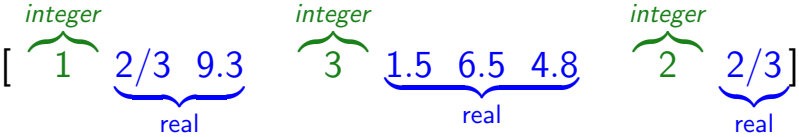
What's the pattern?

Regular Type Expressions (RTEs)

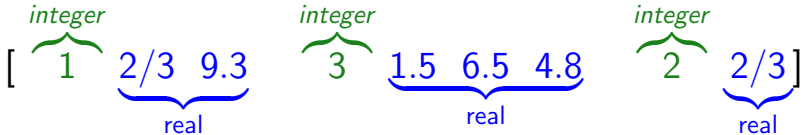
We'd like to recognize sequences with *regular patterns*.

[1, 2/3, 9.3, 3, 1.5, 6.5, 4.8, 2, 2/3]

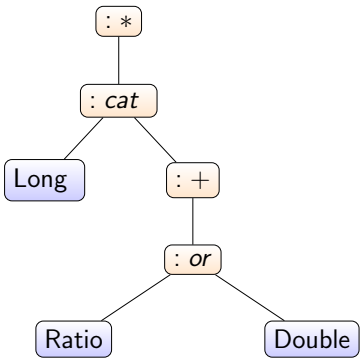
What's the pattern?



Regular Type Expressions (RTEs)



$$(\text{Long} \cdot (\text{Ratio} \cup \text{Double})^{+})^{*}$$



RTEs

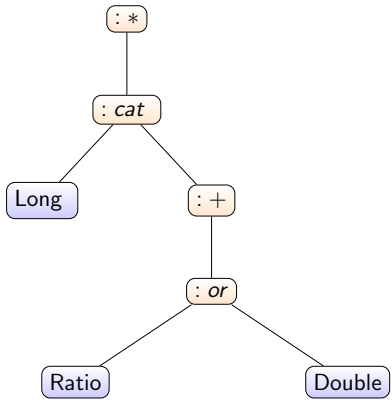
Surface Syntax and AST

What is an RTE?

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

```
(:* (:cat Long
          (:+ (:or Float
                   Ratio))))
```

  - Either: Leaf node designating a Clojure/Java type
  - Or: List of keyword followed by more RTEs
- Semantics: an RTE corresponds to a *set of sequences*: Union, Intersection



RTEs

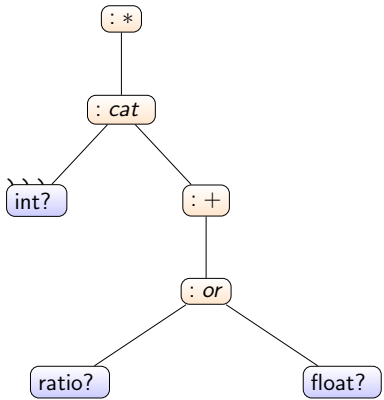
Surface Syntax and AST

What is an RTE?

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

```
(:* (:cat (satisfies int?)
          (:+ (:or (satisfies float?)
                   (satisfies ratio?)))))
```

  - Either: Leaf node designating a Clojure/Java type
  - Or: List of keyword followed by more RTEs
- Semantics: an RTE corresponds to a *set of sequences*: Union, Intersection



Inlining: thanks `clojure.repl.source-fn`

- Clojure notation: AST

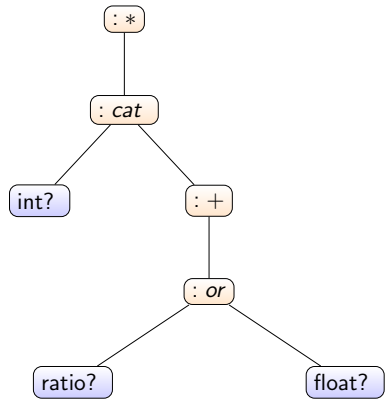
```
(:* (:cat (satisfies int?)
          (:+ (:or (satisfies float?)
                   (satisfies ratio?)))))
```

- Clojure Core

```
(defn int? [x]
  (or (instance? Long x)
      (instance? Integer x)
      (instance? Short x)
      (instance? Byte x)))
```

- Inlined

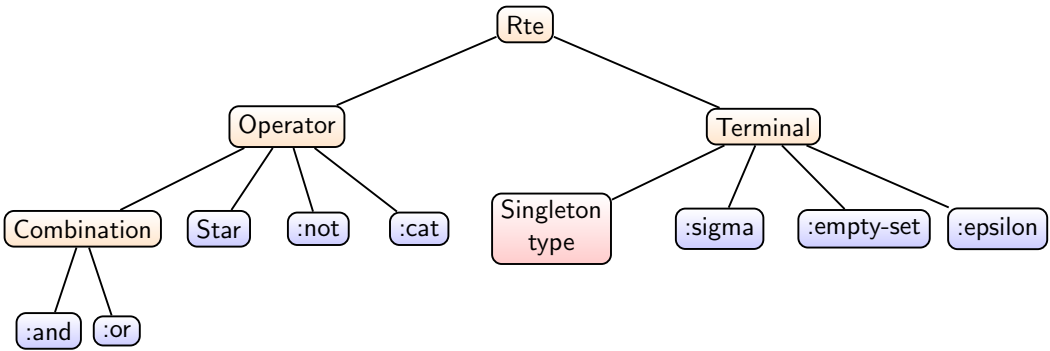
```
(:* (:cat (:or Long Integer Short Byte)
          (:+ (:or (:or Double Float)
                   clojure.lang.Ratio))))
```



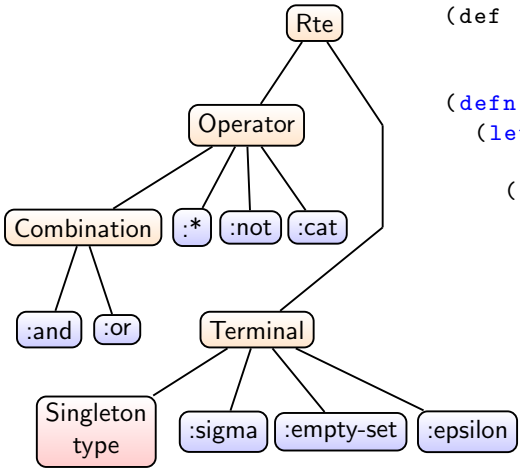
RTE Library

- 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 Abstract Data Type

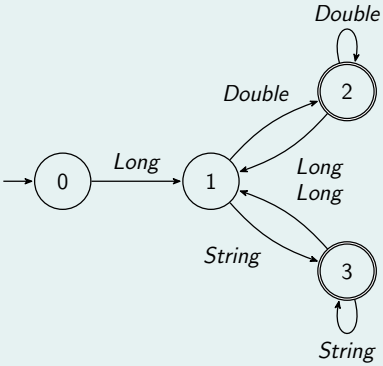


Generate a Random RTE/AST (depth-based)




```
(def rte-keys [:sigma :empty-set :epsilon
               :type :cat :* :and :or :not])

(defn gen-rte [depth]
  (let [key (rand-nth rte-keys)
        g (fn [] (gen-rte (dec depth)))]
    (case key
      (:type) (rand-nth known-types)

      (:sigma :empty-set :epsilon) key

      (:and :or :cat)
        (list key (g) (g))

      (:* :not)
        (list key (g)))))
```

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



Example: How does a pattern predicate work?

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

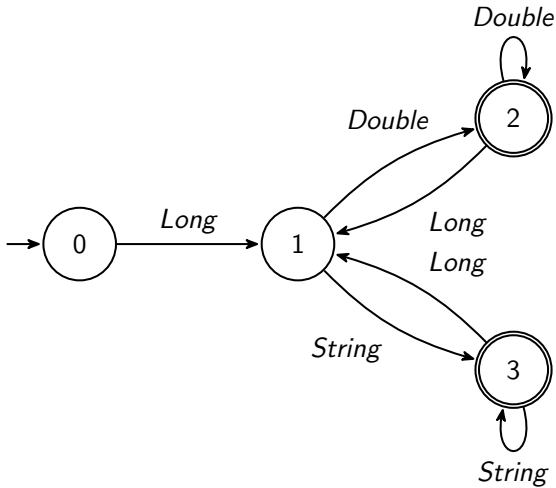
Example: How does a pattern predicate work?

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

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

(DFA)
Deterministic Finite Automaton.

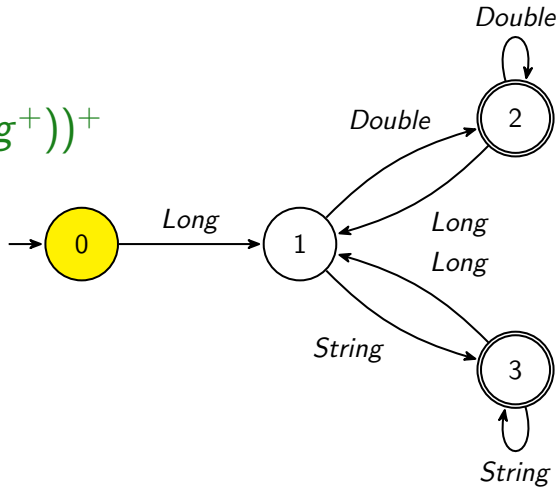
Construct a DFA from an RTE? *Compile-time*



Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \quad \vee String^{+}))^{+}$$

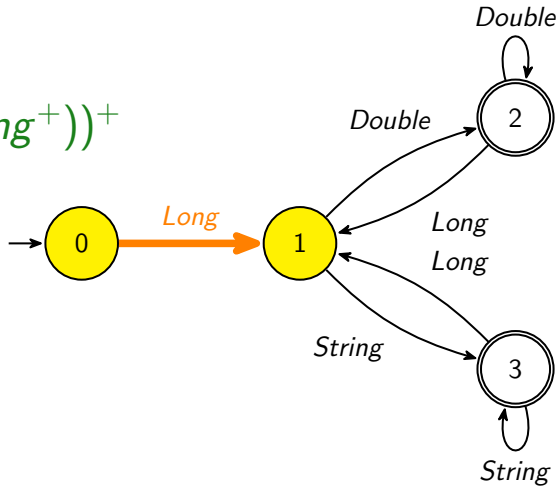


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

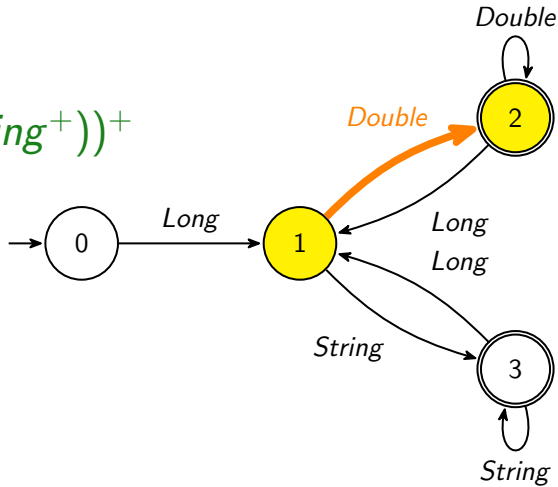


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

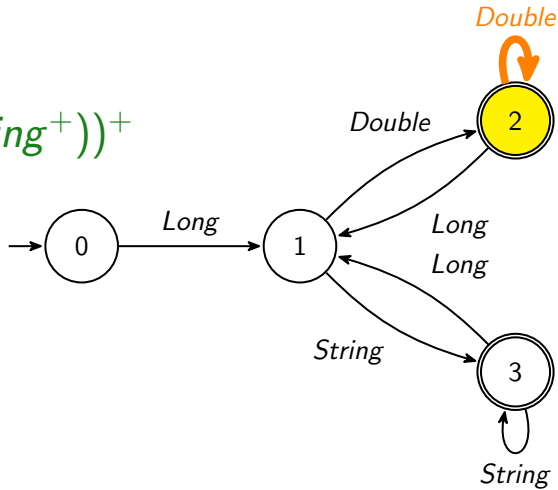


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

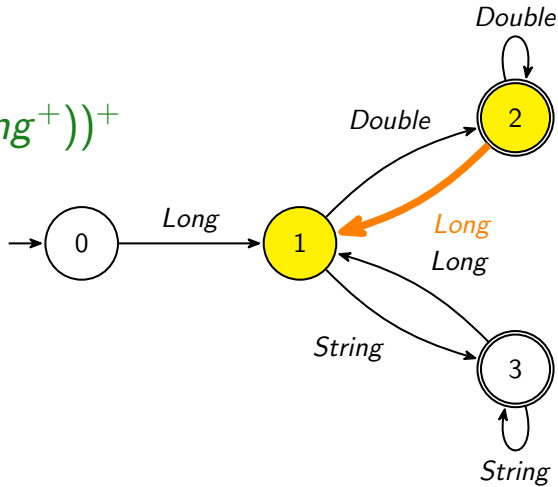


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

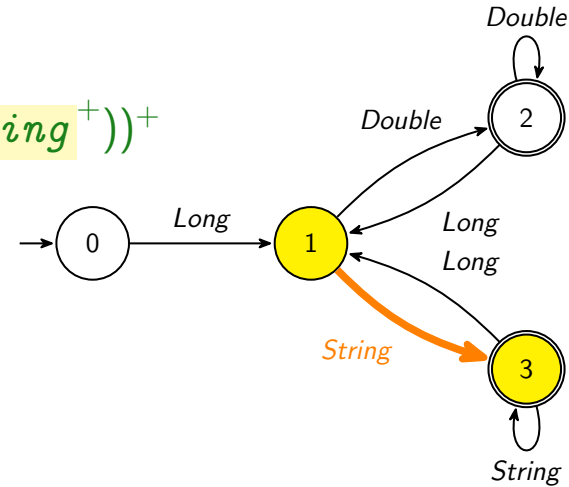


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

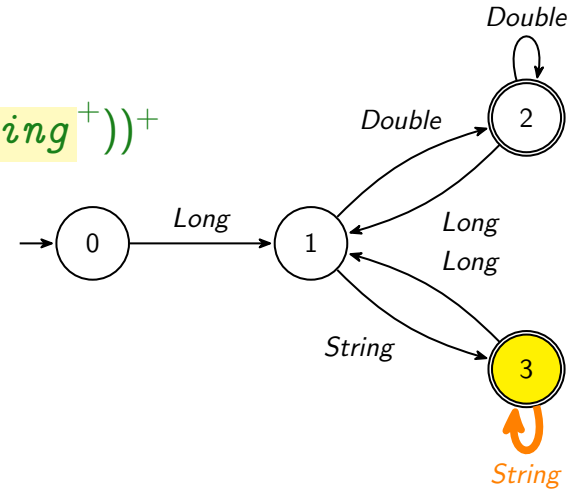


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

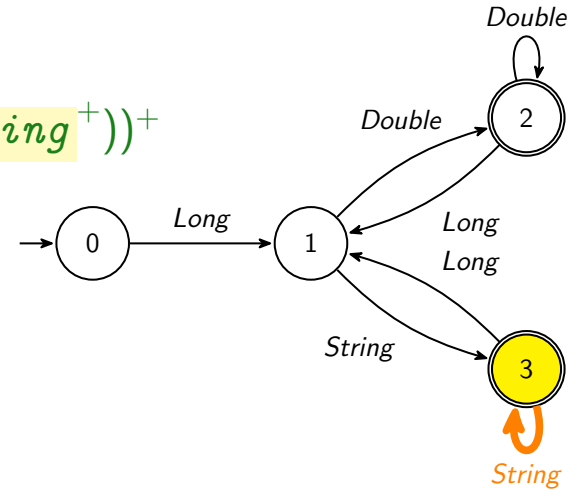


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

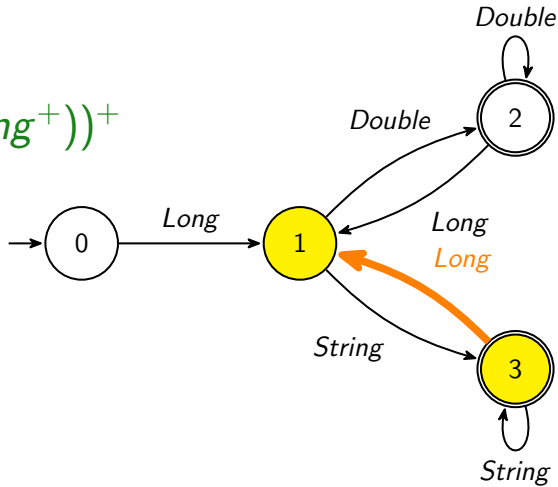


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

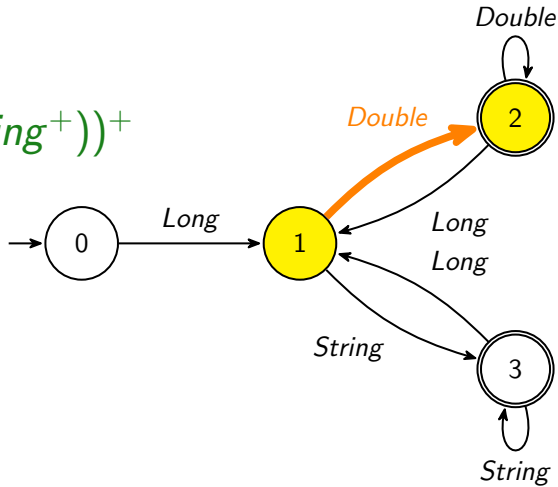


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

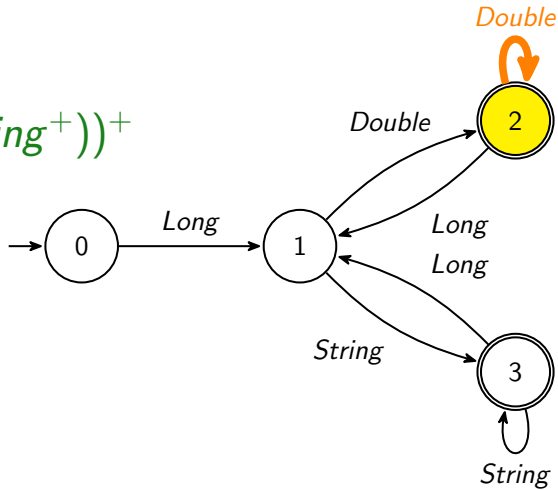


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

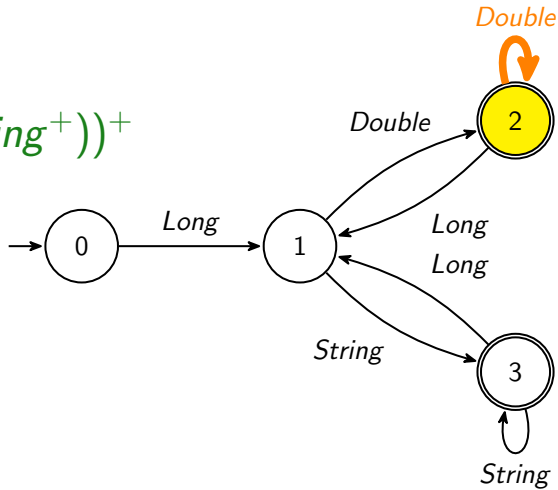


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

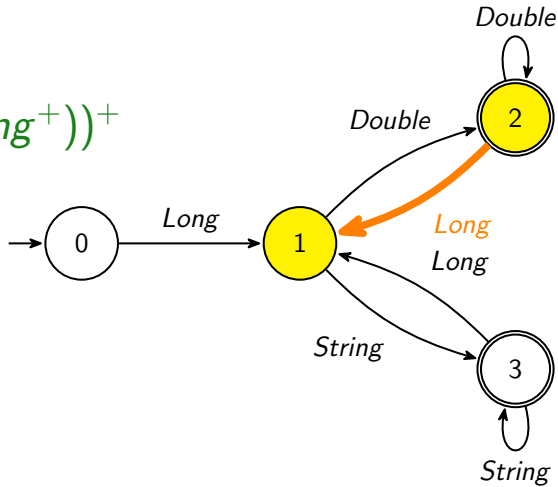


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

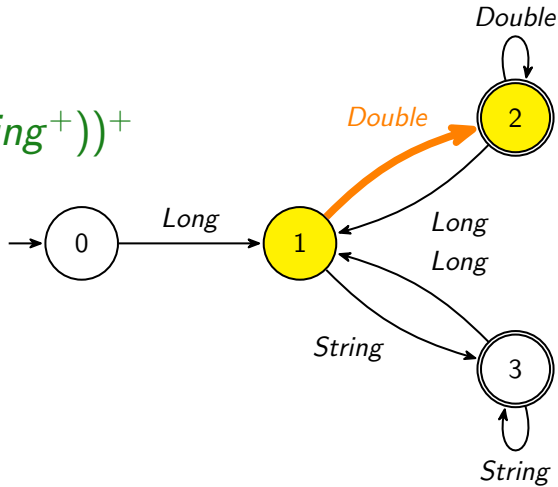


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

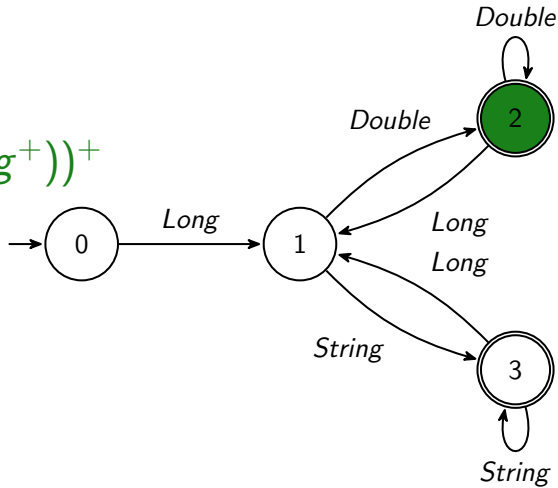


Does the sequence match the pattern? *Run-time*

Example: How does a pattern predicate work?

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

$$pattern = (Long \cdot (Double^{+} \vee String^{+}))^{+}$$

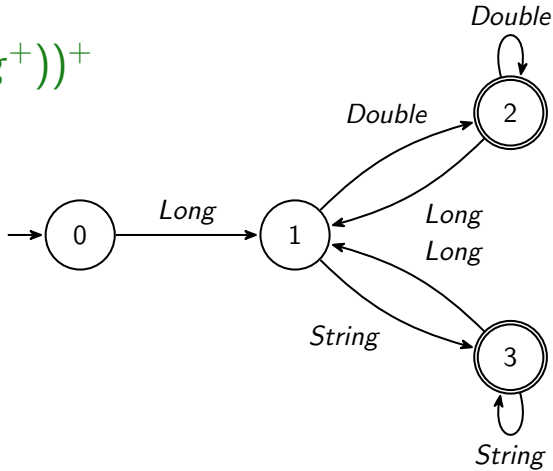


Yes, it's a match!

Example: How does a pattern predicate work?

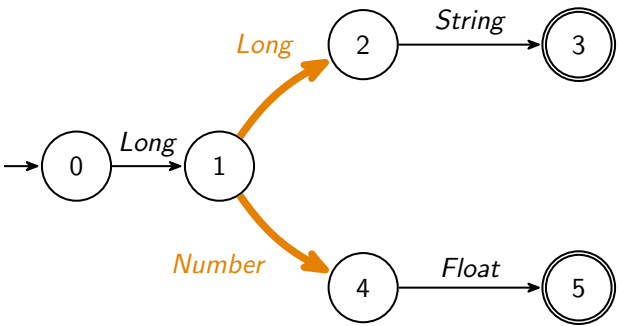
Decision procedure is $O(n)$, *independent of syntactical complexity* of the RTE.

$$pattern = (Long \cdot (Double^{+} \ \vee String^{+}))^{+}$$



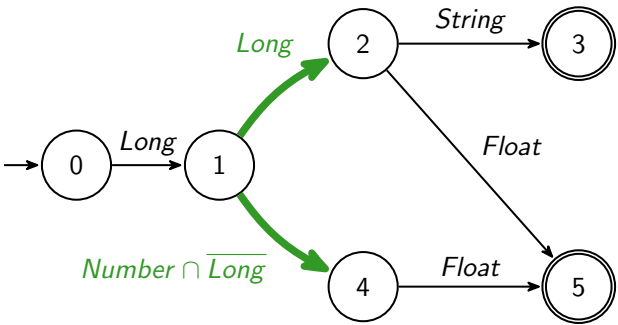
Deterministic (DFA) vs Non-deterministic (NFA)

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



Deterministic (DFA) vs Non-deterministic (NFA)

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



Example, destructuring-case macro

Macro `destructuring-case` evaluates clause matching the correct template.

```
(defn missile-demo [input-seq]
  (destructuring-case input-seq
    [[a b]]
    (rename-file a b)

    [[a b]]
    (delete-file a b)

    [[a b c & d]]
    (copy-file a b c d)))
```

Example, destructuring-case macro

Macro destructuring-case evaluates clause matching the correct template.
And discriminates *based on types* of imbedded values.

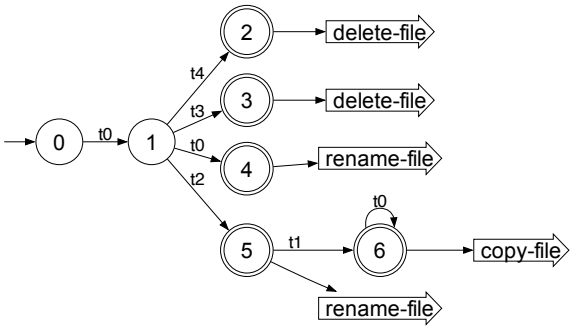
```
(defn missile-demo [input-seq]
  (destructuring-case input-seq
    [[a b]         {a Boolean b (or String Boolean)}]
    (rename-file a b)

    [[a b]         {a Boolean b (or String (satisfies int?))}]
    (delete-file a b)

    [[a b c & d]  {[a d] Boolean b String c (satisfies int?)}]
    (copy-file a b c d)))
```

Example, destructuring-case macro

```
(defn missile-demo [input-seq]
  (destructuring-case input-seq
    [[a b]         {a Boolean b (or String Boolean)}]
    (rename-file a b)

    [[a b]         {a Boolean b (or String (satisfies int?))}]
    (delete-file a b)

    [[a b c & d]   {[a d] Boolean b String c (satisfies int?)}]
    (copy-file a b c d)

    [[a b]         {a Boolean b (or String Long)}]
    (launch-the-missiles a b)))
```

We would like to know *at compile time* whether there is a possible value of `input-seq` which will *launch the missiles*?

Code converted to DFA



$t_0 = \mathit{Boolean}$

$t_1 = \mathit{Byte} \cup \mathit{Integer} \cup \mathit{Long} \cup \mathit{Short}$

$t_2 = \mathit{String}$

$t_3 = \mathit{Long}$

$t_4 = \mathit{Byte} \cup \mathit{Integer} \cup \mathit{Short}$

Macro-expansion warning:
`Unreachable code: (launch-the-missiles a b)`
because DFA has no path to `launch-the-missiles`!

DFA Library `xymbolyco`

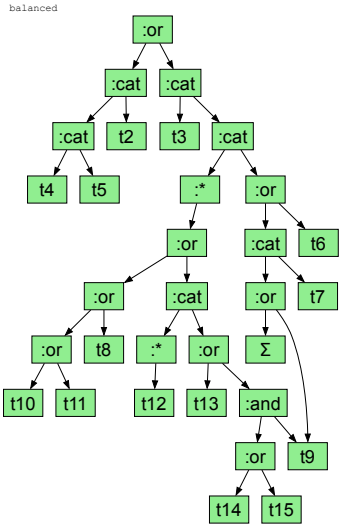
Symbolic Finite Automata Library

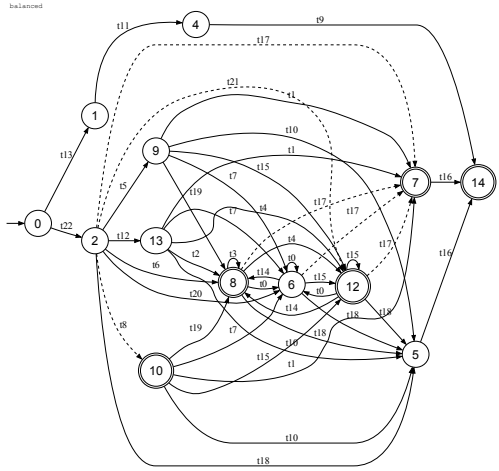
- RTE to DFA
- DFA to RTE
- Pattern matching
- Minimization
- Union, Intersection, Emptiness check
- Need to test `xymbolyco`
  - Random generation of DFA

Generating random DFA

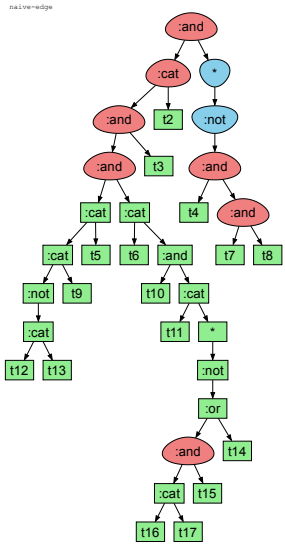
- We can construct DFA from RTE
- ... so generate random RTE (as seen earlier)
- ... and construct DFA

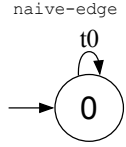
Successful: Generation of DFA from RTE





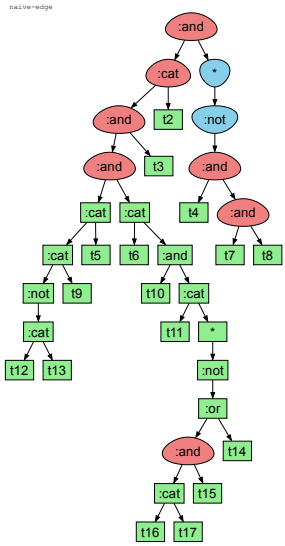
Dubious: Generation of DFA from RTE

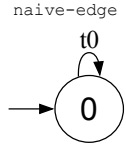




- Trivial DFA.

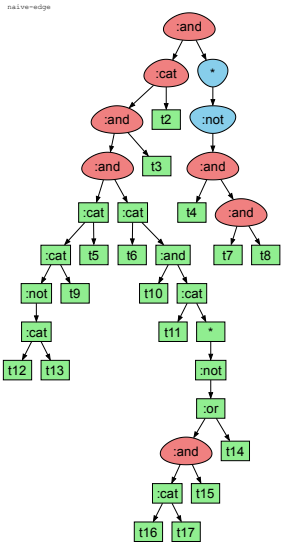
Dubious: Generation of DFA from RTE

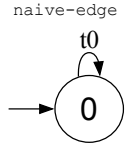




- Trivial DFA.
- Non-uniformity caused by so-called annihilators

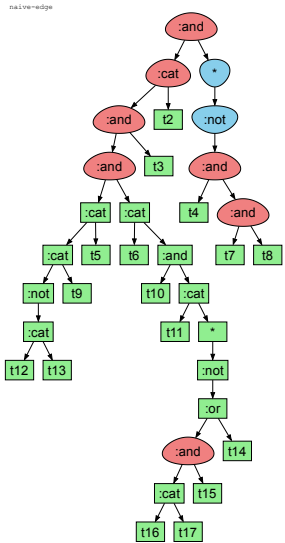
Dubious: Generation of DFA from RTE

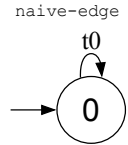




- Trivial DFA.
- Non-uniformity caused by so-called annihilators
- Whenever an input $X = \emptyset$ then $X \cap Y = \emptyset$.

Dubious: Generation of DFA from RTE



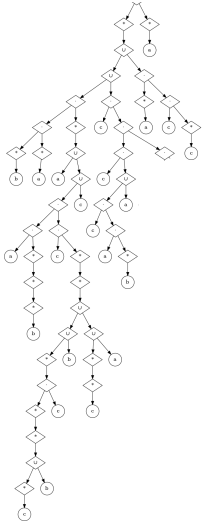


- Trivial DFA.
- Non-uniformity caused by so-called annihilators
- Whenever an input $X = \emptyset$ then $X \cap Y = \emptyset$.
- Thus entire tree generating Y is in vain.

Proposed fix: Balanced Tree Generation

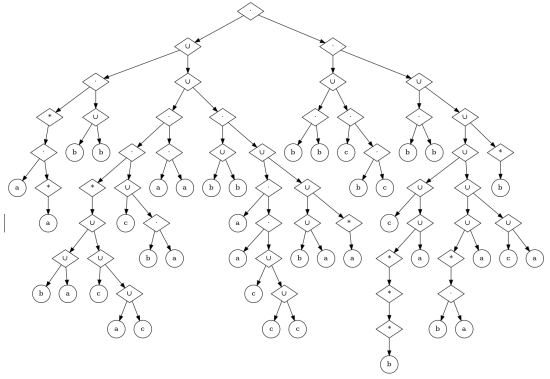
UNIFORM

Portrait — narrow and deep



BALANCED

Landscape — shallow and wide



Requirements

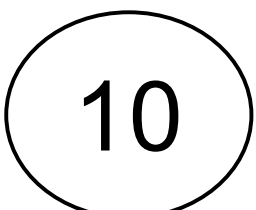
- If we want *large* DFAs, we necessarily should generate *large* RTE (AST).
- However, this is not sufficient. Many large RTEs generate small/trivial DFAs.
- We must generate Portrait not Landscape aspect ratios.

Algorithms for Balanced Generation

Tree-Split Algorithm

Divide and Conquer

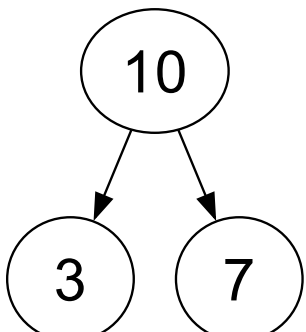
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

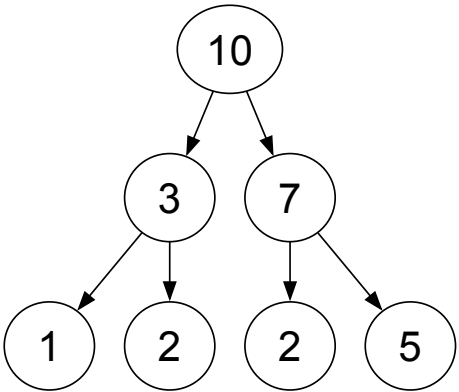
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

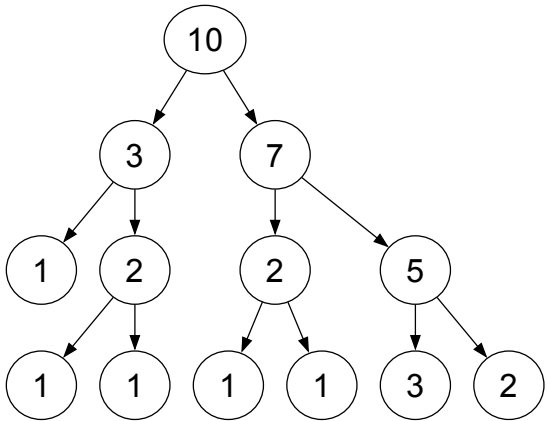
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

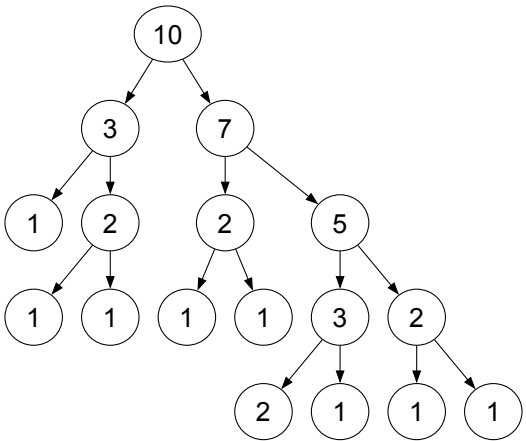
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

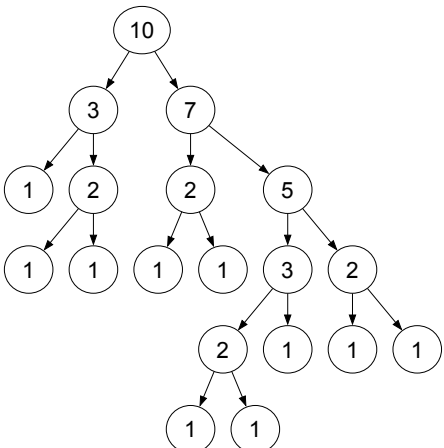
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



Tree-Split Algorithm

Divide and Conquer

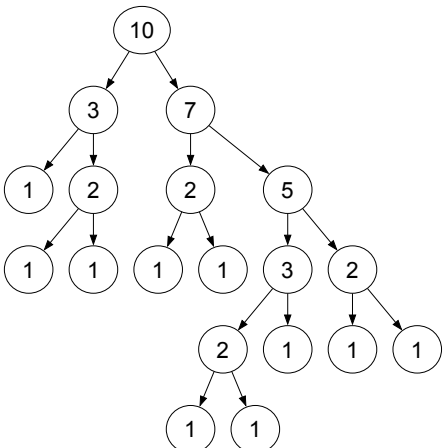
To generate tree having n leaves, find $p = pivot$ between 1 and n; build tree with p leaves on left and $n - p$ leaves on right. Repeat recursively.



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

RTE Tree-Split Generator

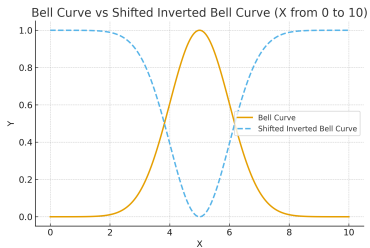
```
(defn tree-split-rte [leaves pivot]
  (if (= 1 leaves)
    (rand-nth inhabited-leaves)
    (let [left-size (pivot leaves)
          left  (fn [] (tree-split-rte left-size pivot))
          right (fn [] (tree-split-rte (- leaves left-size) pivot))
          mid   (fn [] (tree-split-rte leaves pivot))]
      (weighted-case 20 (list :cat (left) (right))
                     30 (list :and (left) (right))
                     30 (list :or (left) (right))
                      5 (list :* (mid))
                     15 (list :not (mid))))))
```

RTE Tree-Split 4-variants

```
;; lopsided comb
(defn comb-rte [leaves]
  (tree-split-rte leaves
     (fn [n]
       (case n
         (1 2 3)  1
          2))))

;; linear split
(defn tree-split-rte-linear [leaves]
  (tree-split-rte leaves
     (fn [n] (max 1 (rand-int n)))))
```

RTE Tree-Split 4-variants

```
;; Gaussian split
(defn tree-split-rte-gaussian [leaves]
  (tree-split-rte leaves
     (fn [n]
       (case n
         (1 2 3) 1
         (max 1 (+ (rand-int (quot n 2))
                   (rand-int (quot n 2))))))))

;; inverse Gaussian split
(defn tree-split-rte-inv-gaussian [leaves]
  (tree-split-rte leaves
     (fn [n]
       (let [r (max 1 (rand-int (quot n 2)))]
         (if (= 0 (rand-int 2))
             r
             (- n r))))))
```



Flajolet Algorithm

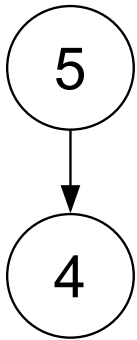
Generate a Skeleton

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

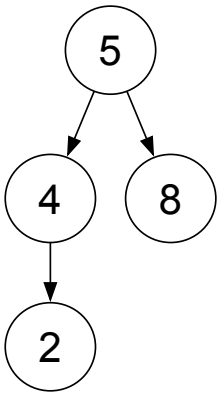
Generate a Skeleton

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

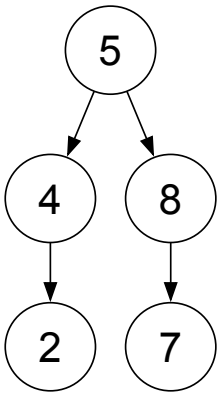
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

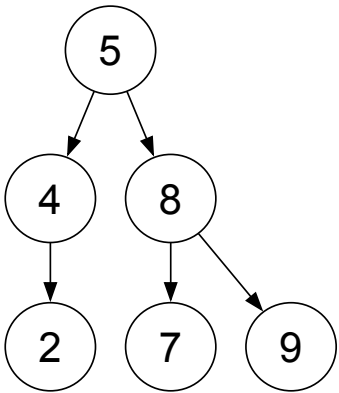
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

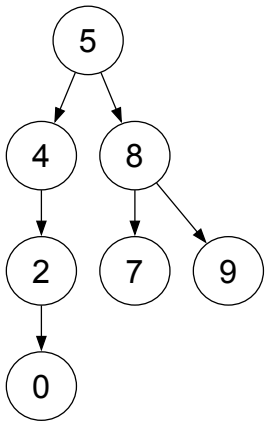
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

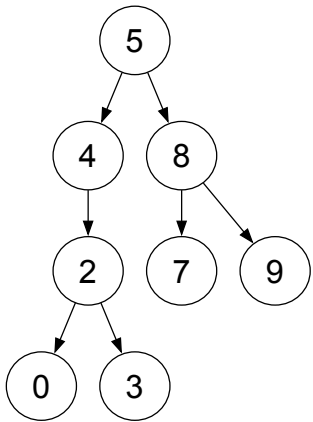
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

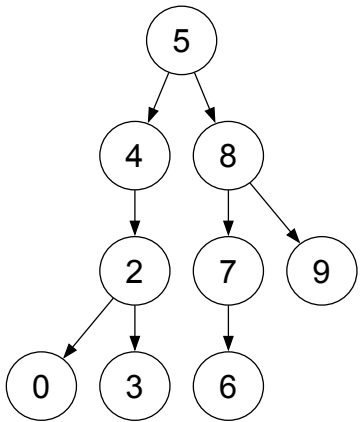
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

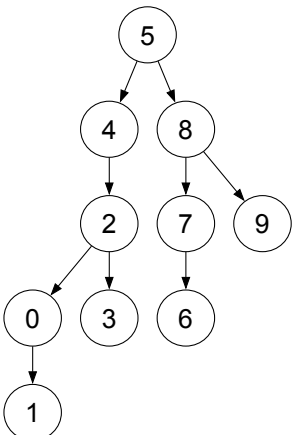
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



Flajolet Algorithm

Generate a Skeleton

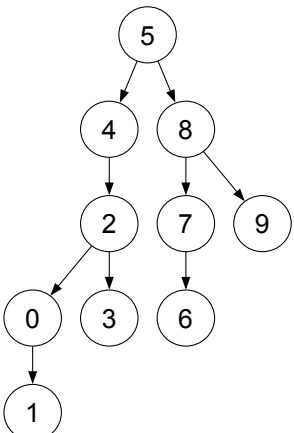
Inserting 0 . . . 9 in random order into Binary Search Tree: `[5 4 2 8 7 9 0 3 6 1]`



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

```
(defn flajolet-rte-by-size [leaves]
  (letfn [(insert-BST [tree d]
            (if (empty? tree)
              [d nil nil]
              (let [[n left right] tree]
                (if (< d n)
                  [n (insert-BST left d) right]
                  [n left (insert-BST right d)])))))
          ...]

    (let [skeleton (reduce insert-BST
                           nil
                           (shuffle (range (dec leaves))))]

      ...)))
```

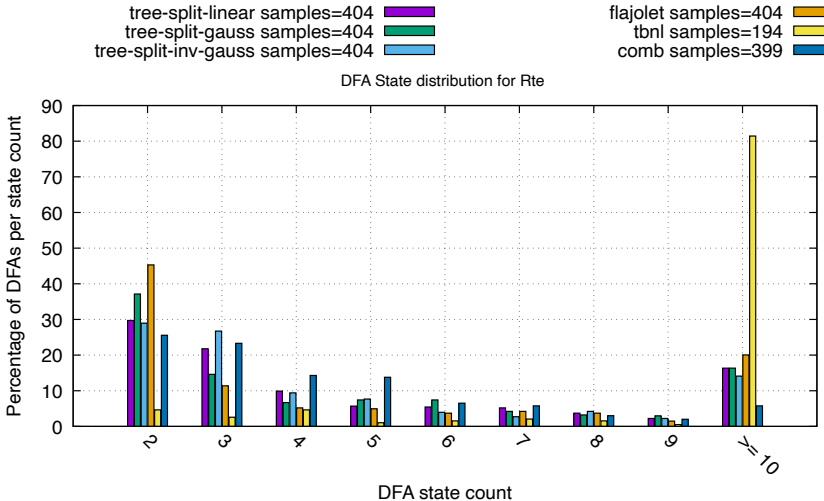
RTE Flajolet Generator

```
(defn flajolet-rte-by-size [leaves]
  (letfn [(insert-BST [tree d]
            ...)
          (to-rte [tree]
            (cond (> (random) 0.90) ;; 10% probability
                  (weighted-case 50 (list :not (to-rte tree))
                                 50 (list :* (to-rte tree)))

                  (empty? tree)
                  (rand-nth inhabited-leaves)

                  :else
                  (let [[_ left right] tree]
                    (weighted-case
                       34 (list :cat (to-rte left) (to-rte right))
                       33 (list :or  (to-rte left) (to-rte right))
                       33 (list :and (to-rte left) (to-rte right))))))]
    (let [skeleton ...]
      (to-rte skeleton))))
```

Quick Demo

Experimental Results

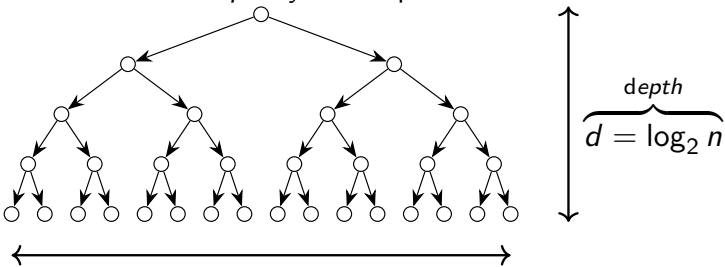
- Ran around 400 random RTE generations of various sizes
- Convert each to DFA
- Measured
  - RTE size (node count)
  - DFA size (state count)
  - Aspect ratio
- Plotted some of the results



Balanced sampling results $\approx$ 15 to 20% of DFAs with 10 states or larger. Flajolet performing best; comb performing worst.

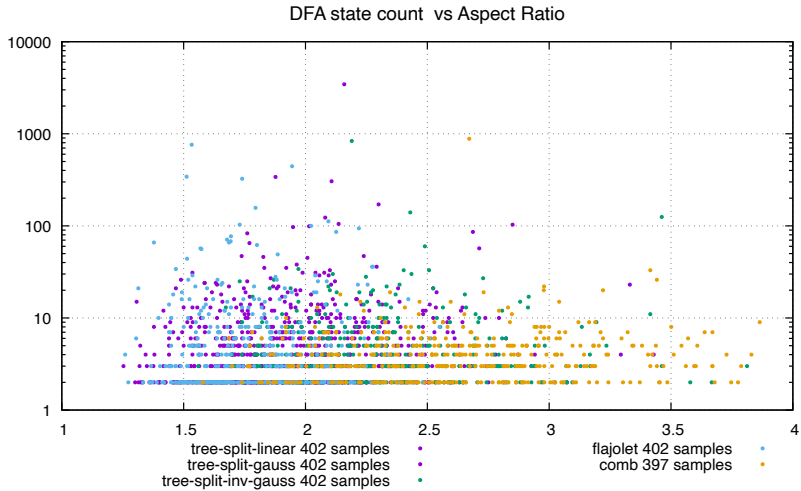
How to measure aspect-ratio of a binary tree

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



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

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



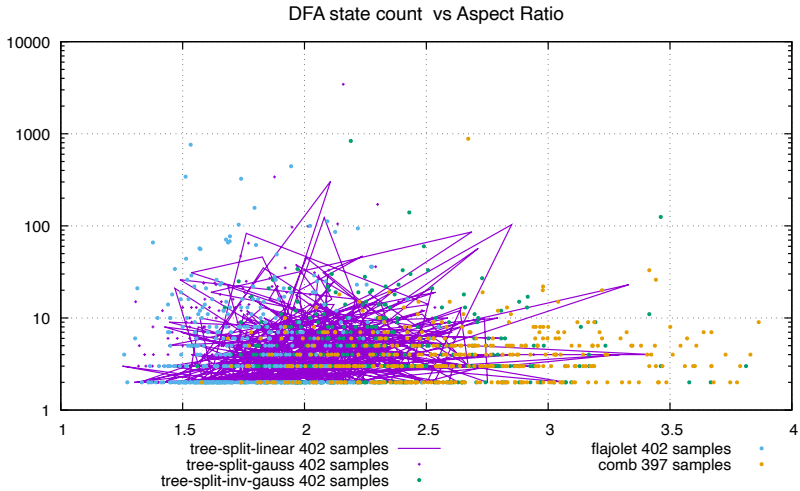
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



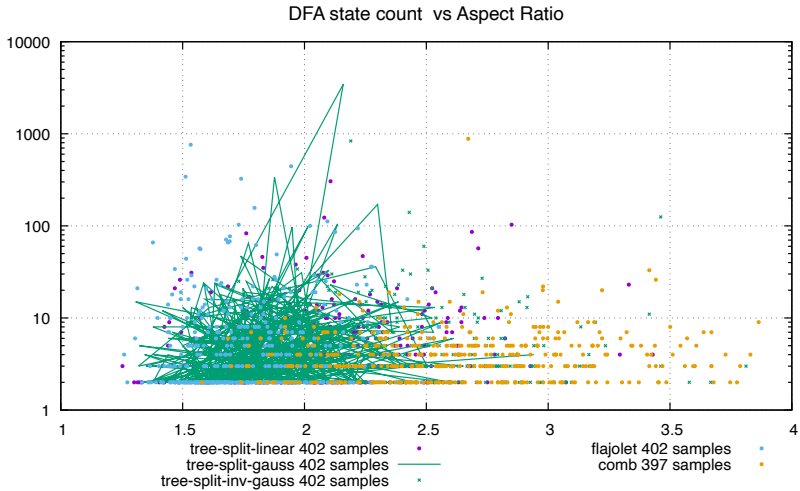
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



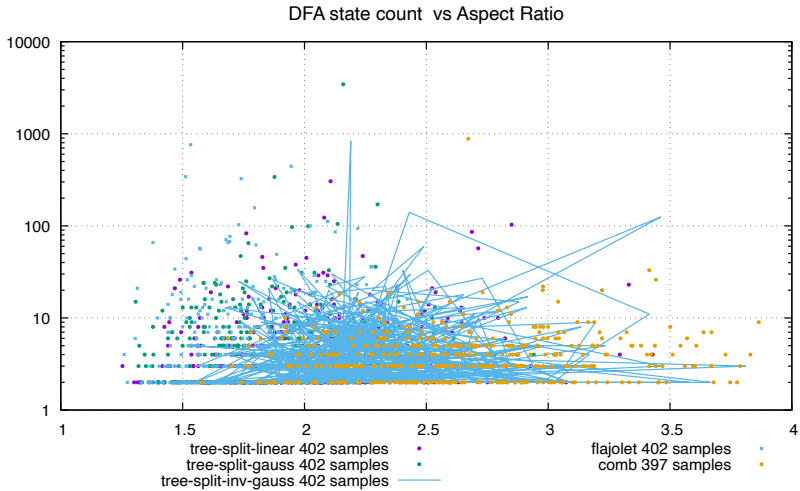
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



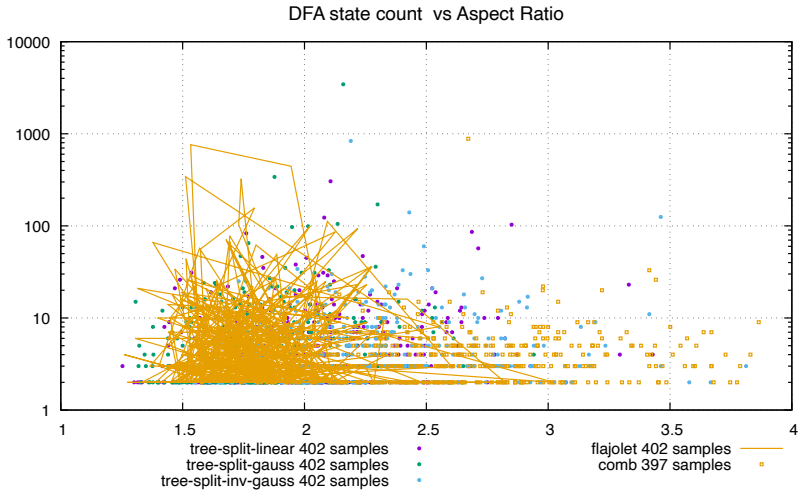
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



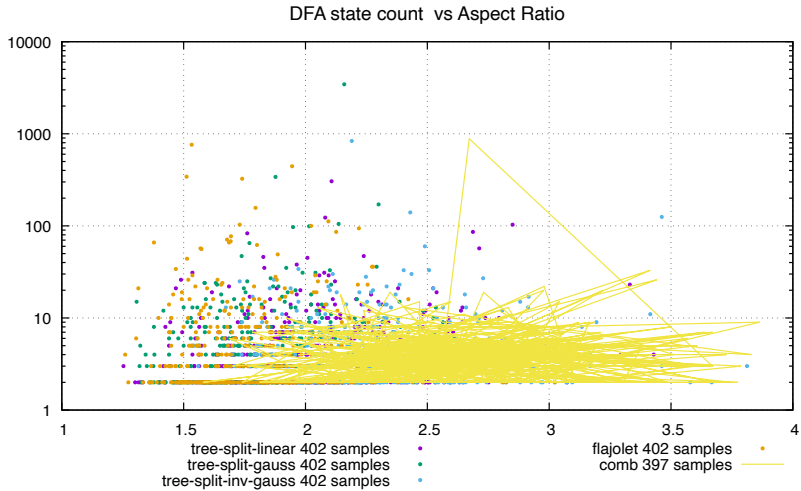
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



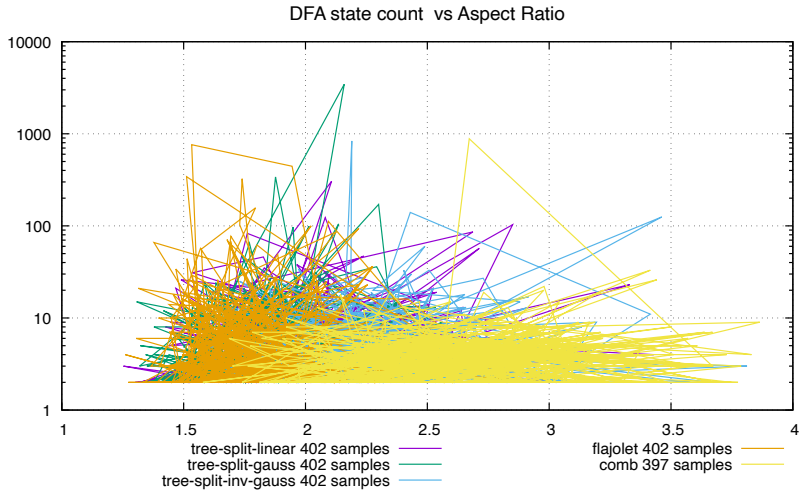
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



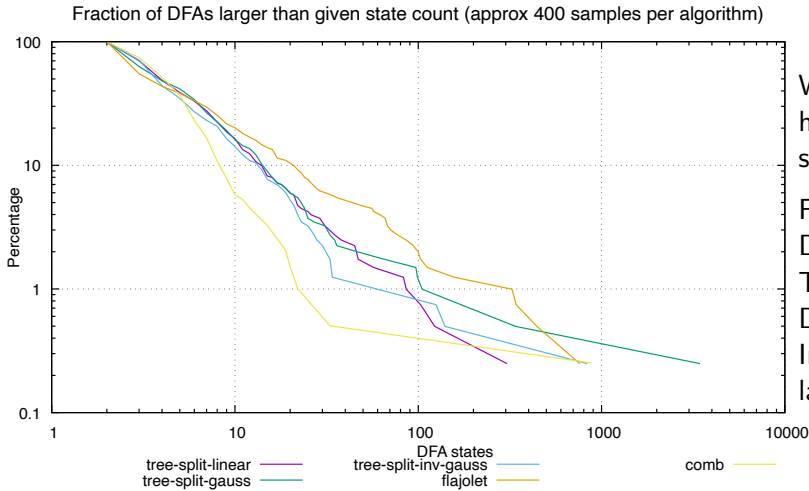
DFA state count
vs Aspect Ratio

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

$\rho = 1 \implies$ balanced

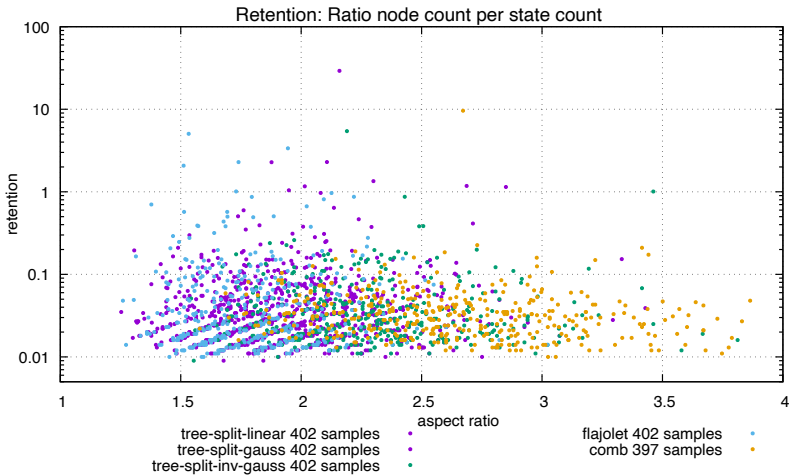
$\rho < 1 \implies$ landscape

$\rho > 1 \implies$ portrait



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

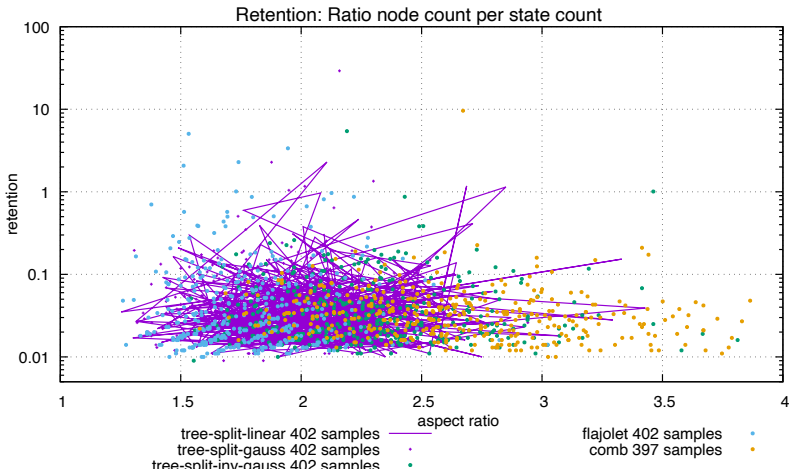
Flajolet produces 10% of DFAs larger than 25 states. Tree-split produces 10% of DFAs larger than 15 states. Imbalanced produces 10% larger than 8 states.



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).

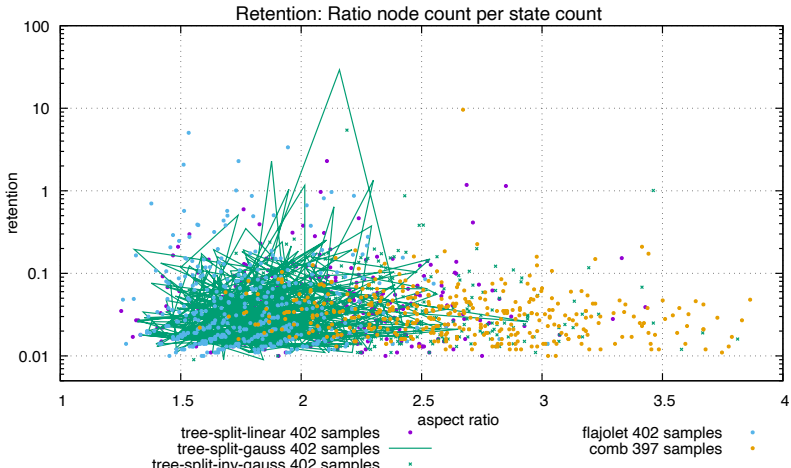
$\rho = \frac{state\ \ count}{node\ \ count}$

$\rho > 1$ is good.



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).

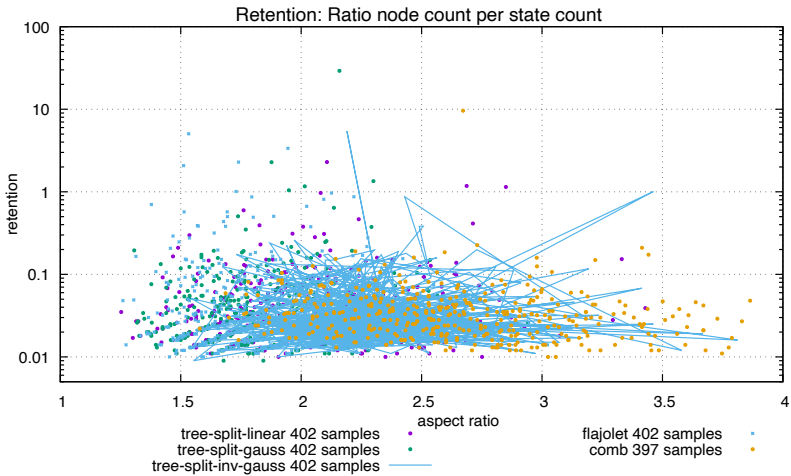
$\rho = \frac{state\ \ count}{node\ \ count}$
$\rho > 1$ is good.



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).

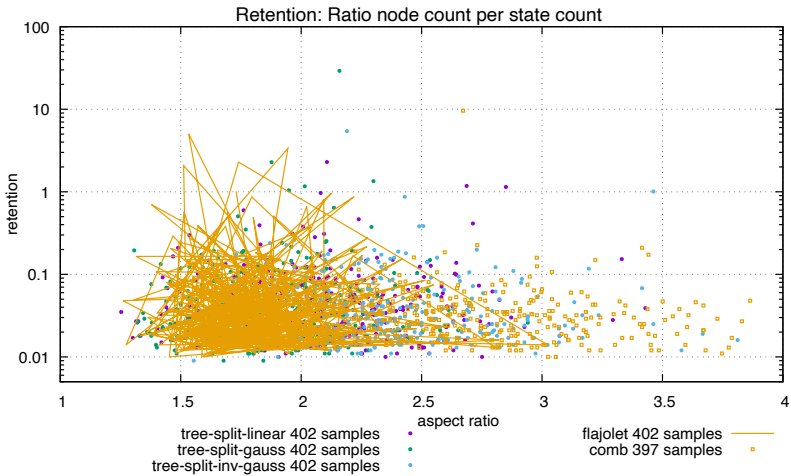
$\rho = \frac{state\ \ count}{node\ \ count}$

$\rho > 1$ is good.



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).

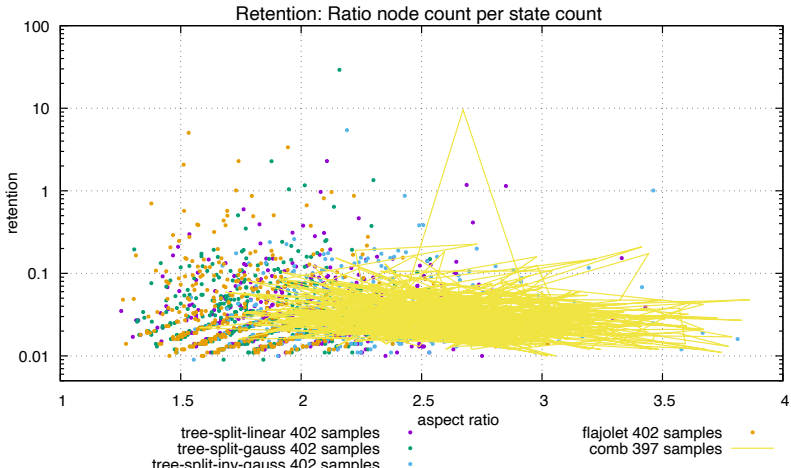
$\rho = \frac{state\ \ count}{node\ \ count}$
$\rho > 1$ is good.



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).

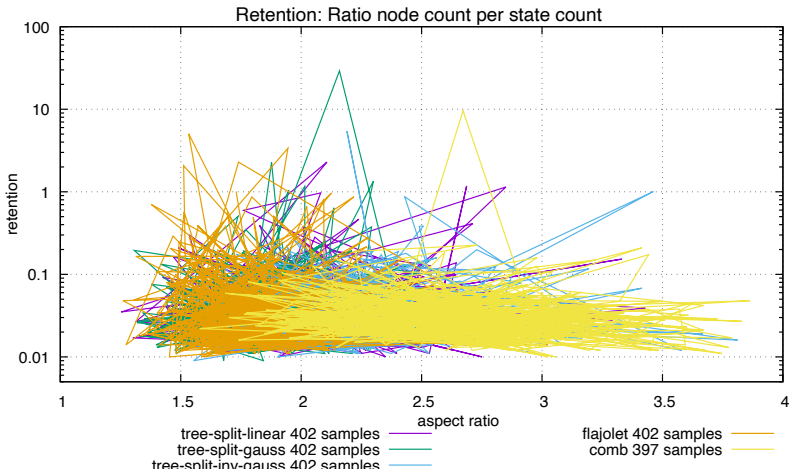
$\rho = \frac{\mathit{state\ \ count}}{\mathit{node\ \ count}}$

$\rho > 1$ is good.



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.



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 requires AST generation; we hope our research applies beyond RTE/DFA generation.
- *Balanced tree* generation *improves* coverage in the *semantic* space.
- But not a silver bullet.
- Overhead in terms of lines-of-code overhead is small.
- Project also includes Common Lisp, Clojure, and Python implementations.

Questions and Answers

Critical Feedback Welcome!

- jimka.issy@gmail.com
- Discord: jimka2001
- Github: jimka2001
- All code publically available:

Clojure	`https://github.com/jimka2001/clojure-rte`
Scala	`https://github.com/jimka2001/scala-rte`
Python	`https://github.com/jimka2001/python-rte`
Common Lisp	`https://github.com/jimka2001/cl-rte`