Recognizing Regular Patterns in Mixed-Type Sequences using Symbolic Finite Automata

Jim E. Newton

EPITA Research Laboratory, Le Kremlin Bicêtre, FRANCE

May 26, 2025

Overview

Background

Goal

- To efficiently recognize *regular patterns* in mixed-type sequences.
- Primarily a research topic
  - . . . but extract useful libraries if possible.
- To support in various programming languages:
  - Scala as `Seq[Any]`
  - Python as `list`, `generator`
  - Clojure as `seq`
  - Common Lisp as `SEQUENCE`

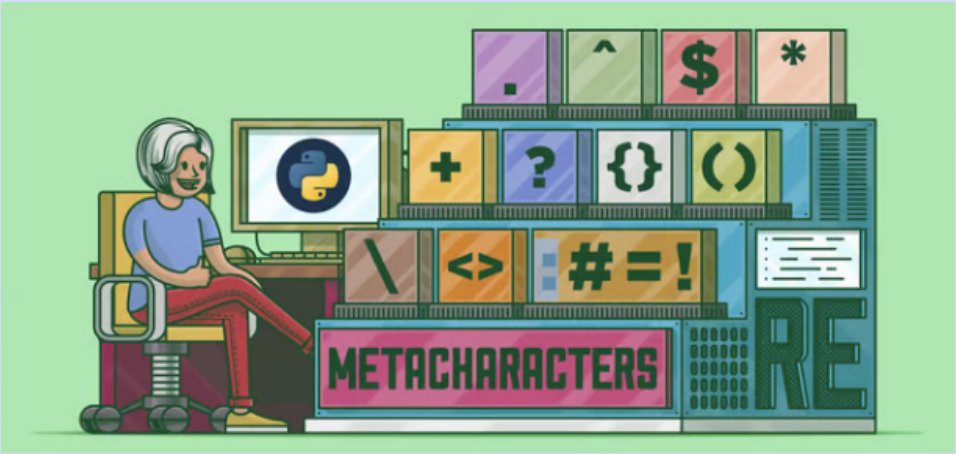
Publication History

- Common Lisp
  - *Type-Checking Heterogeneous CL Sequences* [Newton, Demaille, Verna] 2016 ELS
  - *Recognizing hetergeneous sequences by rational type expression*, [Newton, Verna] 2018 Workshop on Meta-Programming Techniques and Reflection.
  - PhD Thesis 2018 for theoretical, implementation, and performance details `https://www.lrde.epita.fr/wiki/Publications/newton.18.phd`
- Generalized beyond Common Lisp
  - *A Portable, Simple, Embeddable Type System* [Newton, Pommellet] 2021 ELS
  - *An Elegant and Fast Algorithm for Partitioning Types* [Newton] 2023 ELS
  - *Type-Checking Heterogeneous Sequences in a Simple Embeddable Type System* [Newton] 2025 PADL
- RTE Available at: `github.com/jimka2001/python-rte`.

Motivating Demo

Example

Regular Type Expressions (RTEs)



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*.

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

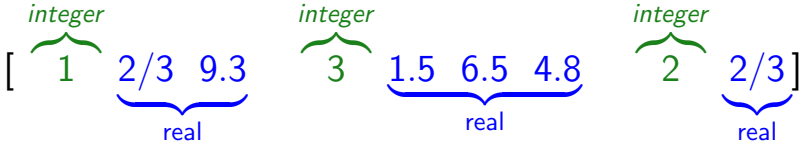
What's the pattern?

Regular Type Expressions (RTEs)

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

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

What's the pattern?



Regular Type Expressions (RTEs)

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

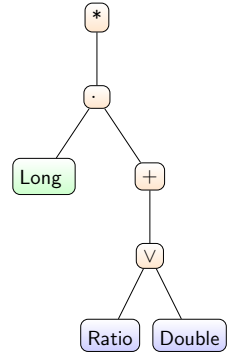
Regular Type Expressions (RTEs)

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

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

We propose Regular Type Expressions (RTEs)

- Regular type expression: (Long · (Ratio ∪ Double)$^+$)*
- *Challenge №1:* We need a surface syntax (and AST).
- *Challenge №2:* How to support *Long* ∩ $\overline{\textit{Number}}$ in a PL?



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]`

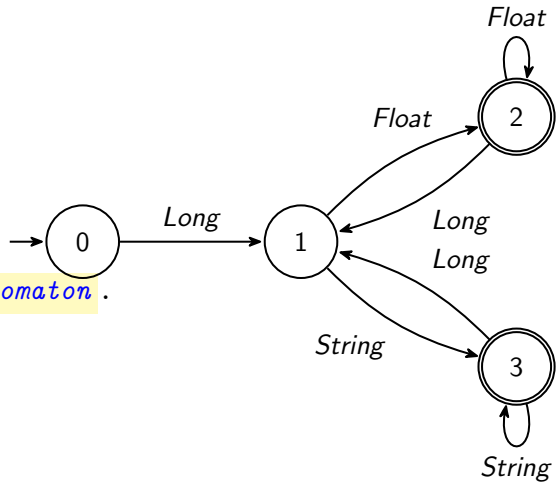
Example: How does a pattern predicate work?

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

We construct a (σ-DFA) *Symbolic Deterministic Finite Automaton*.

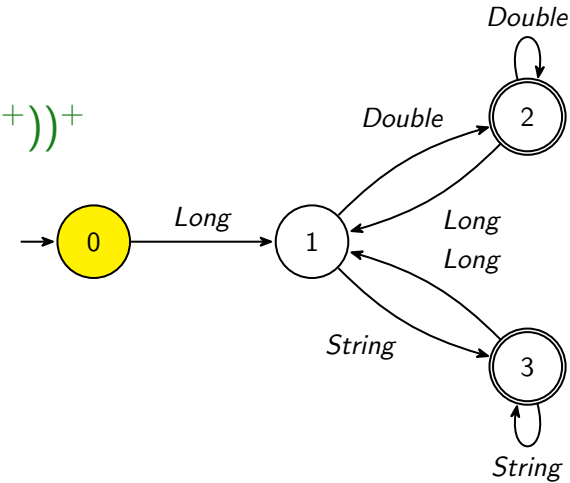
Challenge №3: How to construct a σ-DFA from an RTE? *Compile-time*



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]

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

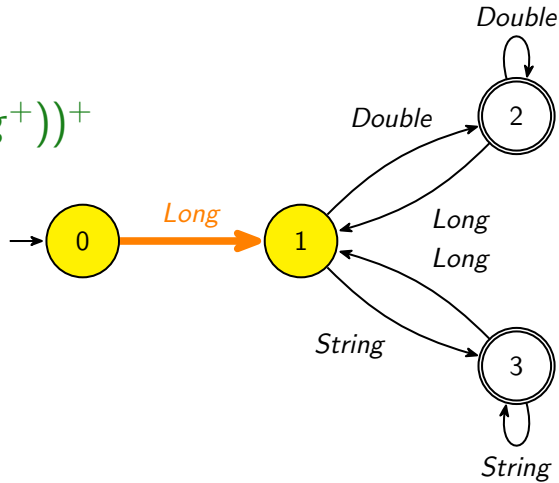


Does the sequence match the pattern? *Run-time*

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]

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

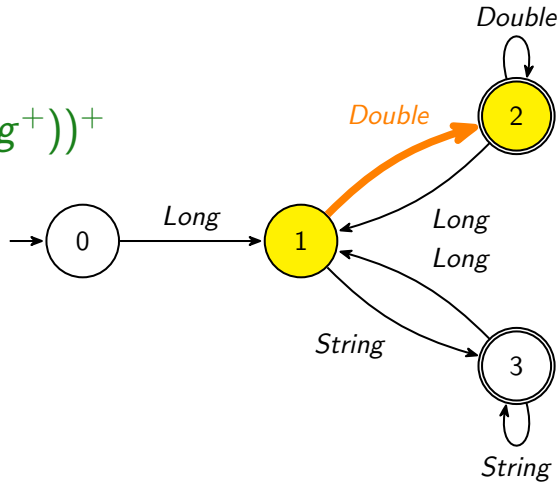


Does the sequence match the pattern? *Run-time*

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]

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

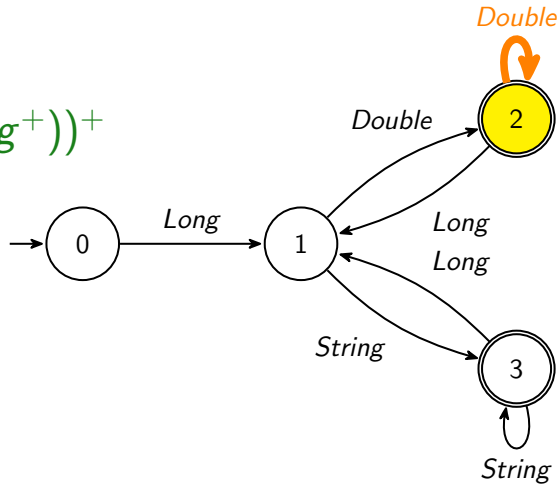


Does the sequence match the pattern? *Run-time*

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]

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

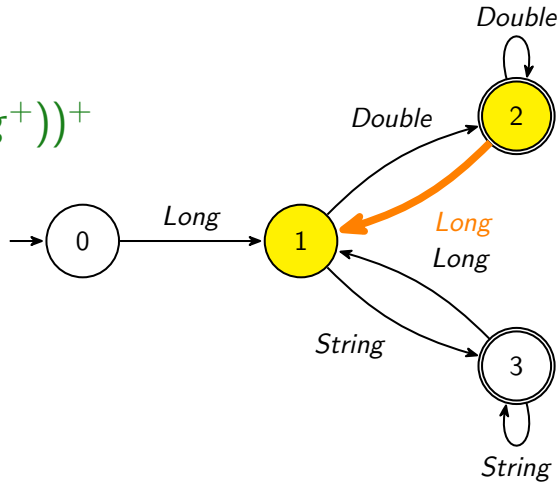


Does the sequence match the pattern? *Run-time*

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]

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

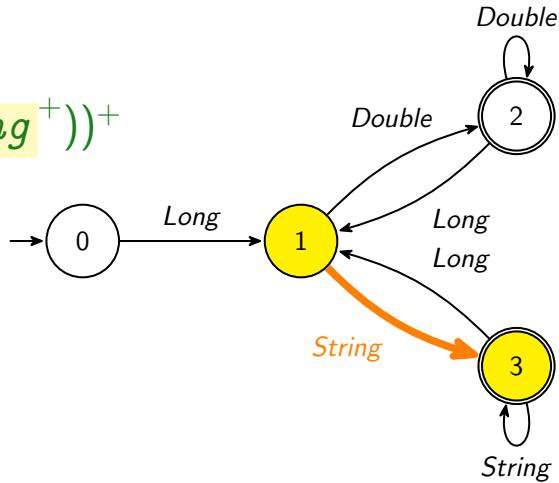


Does the sequence match the pattern? *Run-time*

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]

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

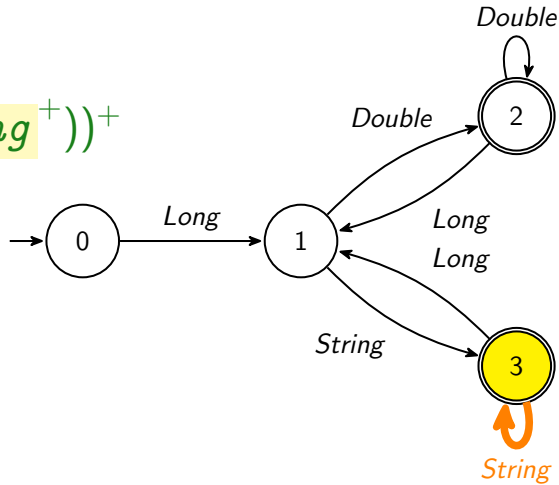


Does the sequence match the pattern? *Run-time*

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]

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

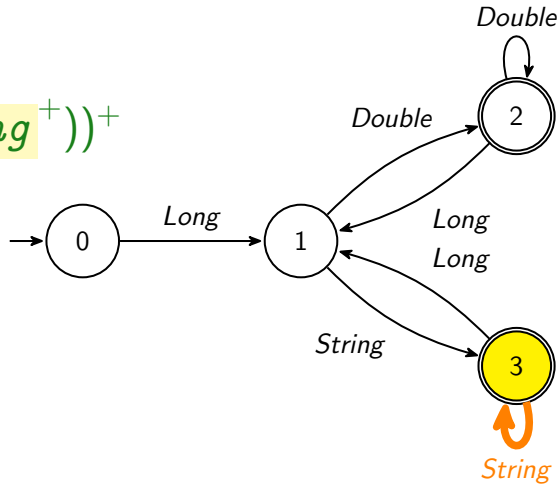


Does the sequence match the pattern? *Run-time*

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]

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

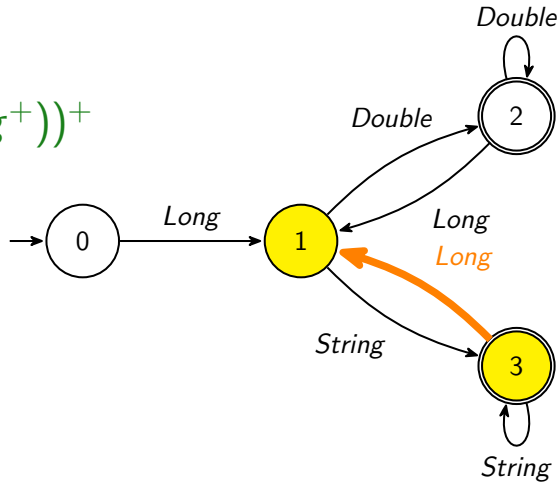


Does the sequence match the pattern? *Run-time*

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]

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

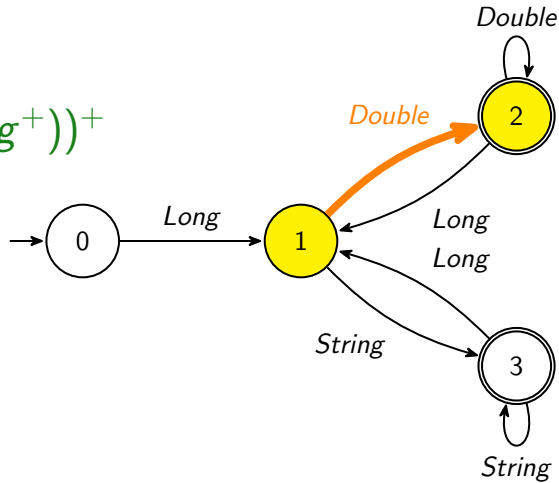


Does the sequence match the pattern? *Run-time*

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]

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

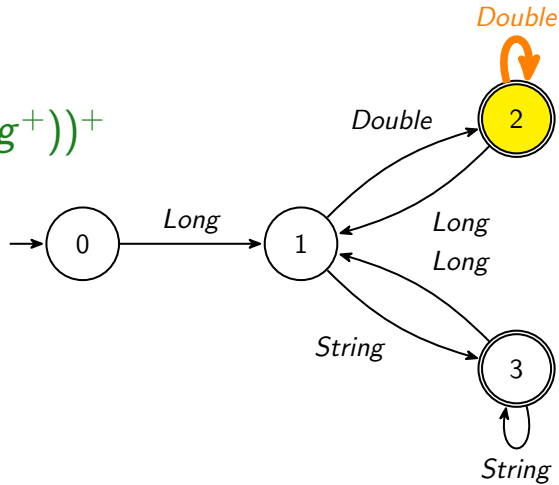


Does the sequence match the pattern? *Run-time*

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]

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

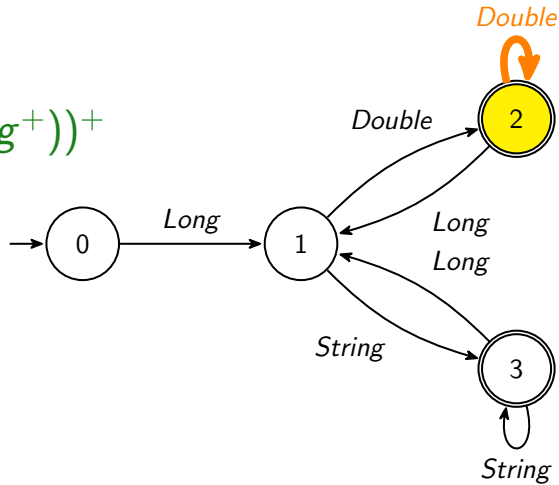


Does the sequence match the pattern? *Run-time*

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]

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

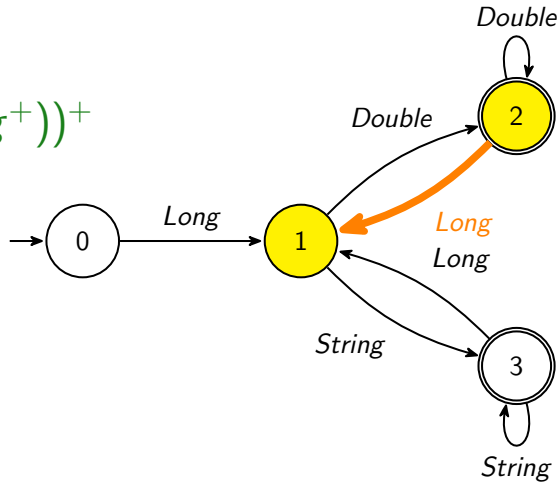


Does the sequence match the pattern? *Run-time*

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]

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

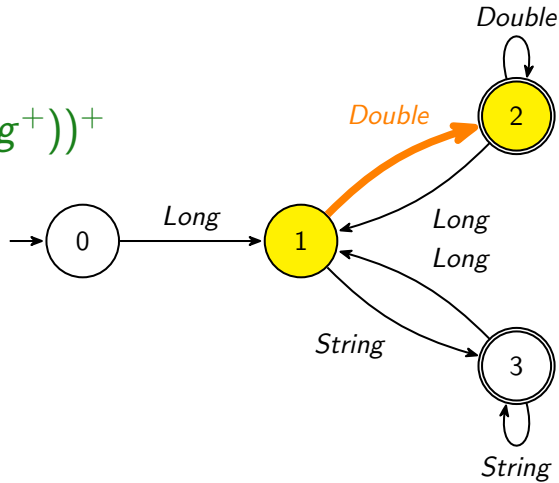


Does the sequence match the pattern? *Run-time*

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]

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

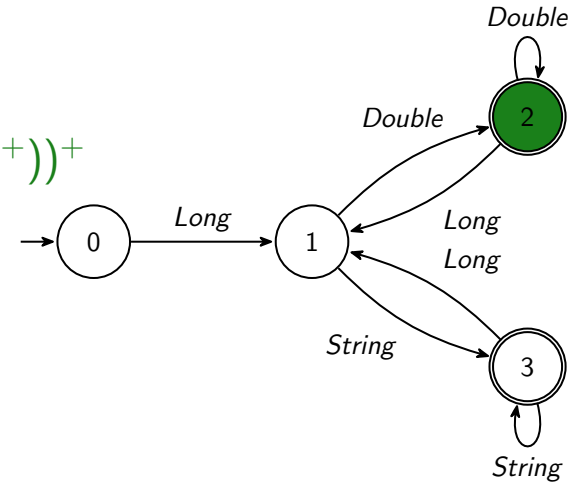


Does the sequence match the pattern? *Run-time*

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]

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

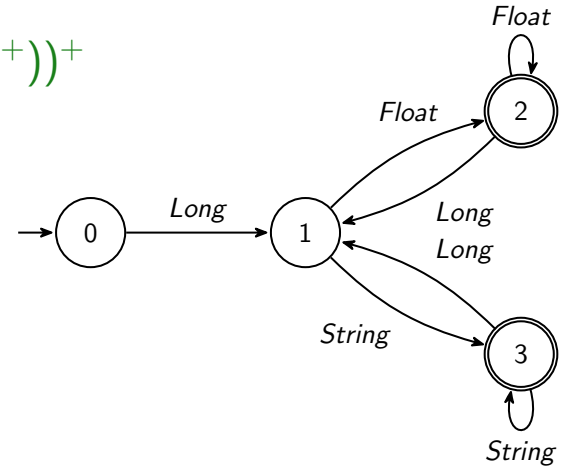


Yes, it's a match!

Example: How does a pattern predicate work?

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

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

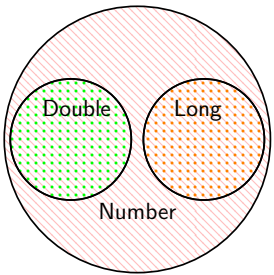


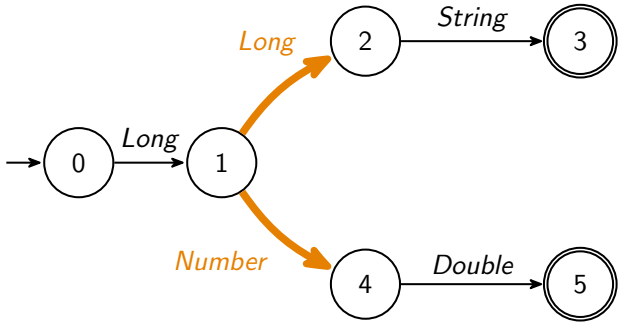
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$



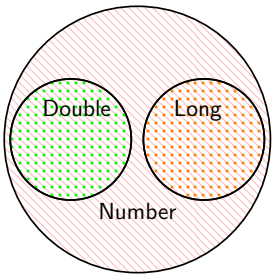


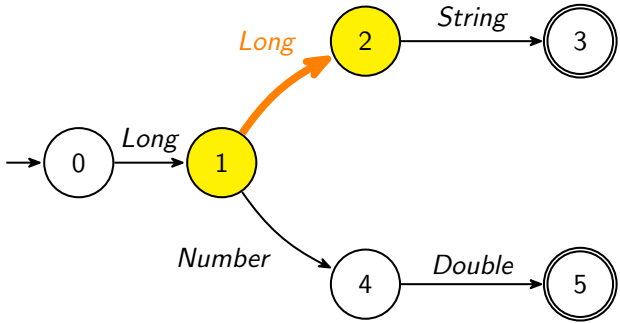
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$



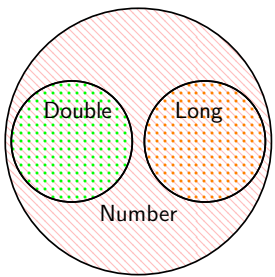


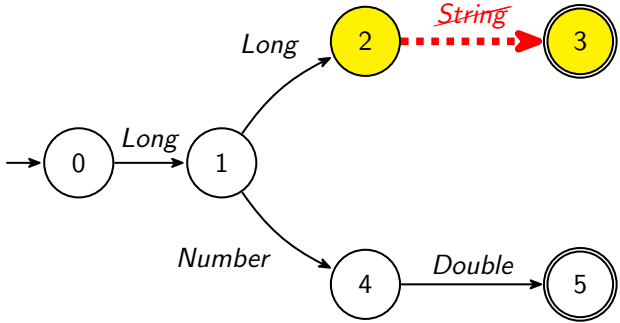
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$



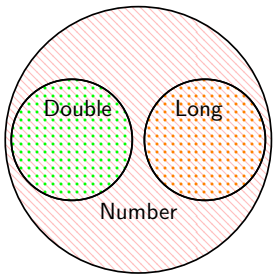


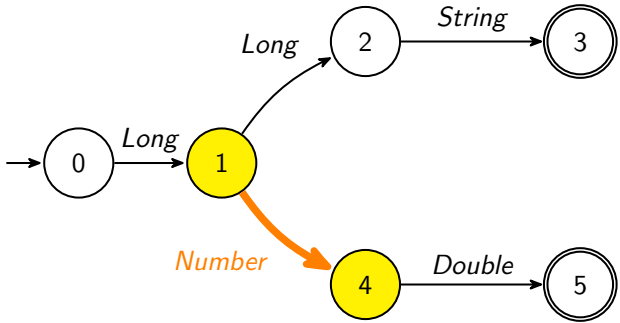
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$





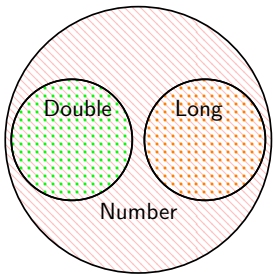
Backtracking $\Longrightarrow$ $O(2^n)$

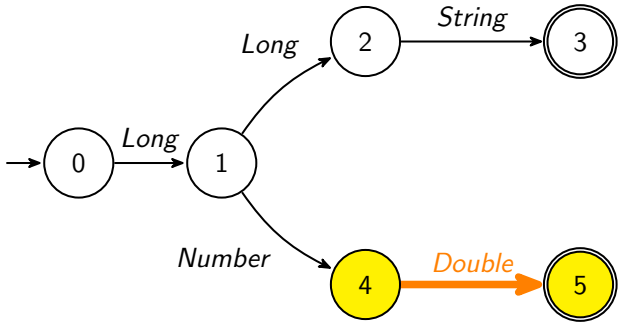
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$





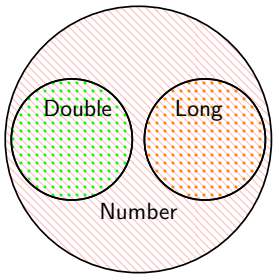
Backtracking $\Longrightarrow$ $O(2^n)$

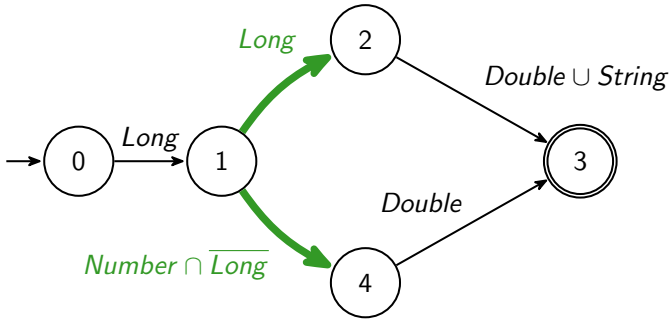
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$





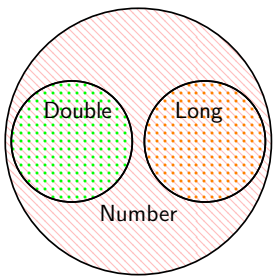
Deterministic choice between *Int* vs $Number \cap \overline{Long}$

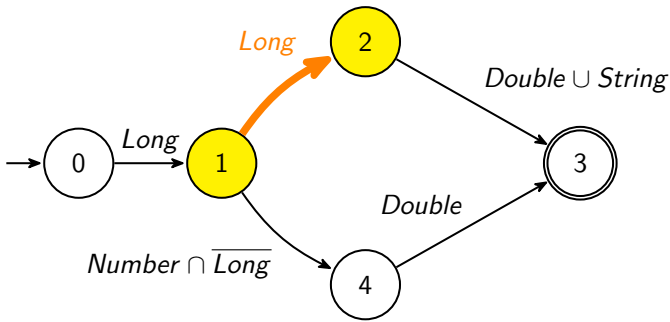
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$



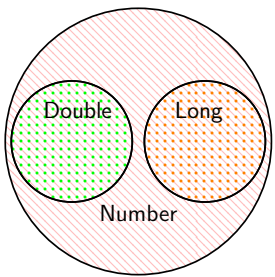


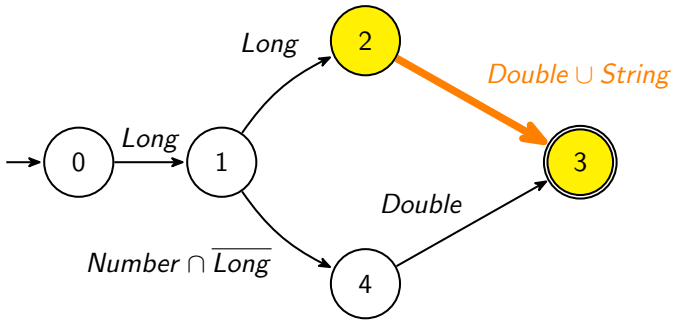
Deterministic (DFA) vs Non-deterministic (NFA)

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

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$

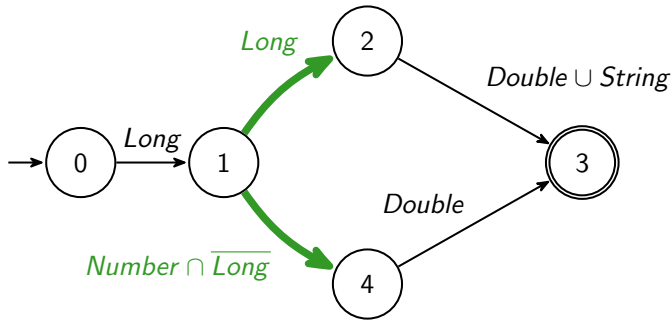




Deterministic (DFA) vs Non-deterministic (NFA)

$Long \subseteq Number$

$Long \cap Number \neq \emptyset$



Challenge №4: How to partition/decompose types?

$\{String, Int, Long\} \rightarrow \{String, Long, Number \cap \overline{Long}\}$

Challenges of the Project

- *Challenge № 1:* RTE Representation: Representing an RTE in a PL?
- *Challenge № 2:* Type Lattice: Union, intersection, complement types?
- *Challenge № 3:* DFA Construction: Constructing from RTE?
- *Challenge № 4:* Determinism: Type partitioning?

Challenge №1: RTE Representation

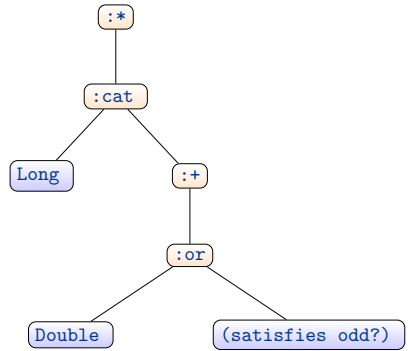
How to represent an RTE in Clojure, Scala, Python, etc...?

RTEs

PL Declarative, Composable Syntax

- Mathematical notation:
  $(Long \cdot (Double \cup odd)^{+})^{*}$
- Leaf nodes interface to native type system
- Clojure RTE notation:

```
1 (:* (:cat Long
2           (:+ (:or Double
3                    (satisfies odd?)))))
```



Challenge №2: Type Lattice

How to designate type $Double \cup (odd \cap \overline{13})$ in Clojure ?

Challenge №2: Type Lattice

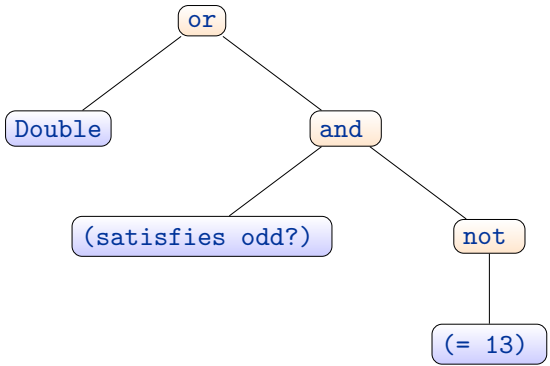
How to designate type $Double \cup (odd \cap \overline{13})$ in Clojure ?

```
(or Double (and (satisfies odd?) (not (= 13))))
```

Challenge №2: Type Lattice

How to designate type $Double \cup (odd \cap \overline{13})$ in Clojure ?

```
(or Double (and (satisfies odd?) (not (= 13))))
```



Type Membership Predicate

$x \in [\![\nu]\!] \rightarrow \{true, false\}$

Boolean membership question is *always answerable*.

```
1  ;; returns true
2  (typep -42 'Long)
3
4  ;; returns true
5  (typep 7 '(or String Long))
6
7  ;; returns false
8  (typep 32 '(satisfies odd?))
```

Subtype Predicate

$[\![\nu]\!] \subset [\![\mu]\!] \to \{\mathit{true}, \mathit{false}, \text{dont-know}\}$

- *Semi-Boolean* Subtype predicate *sometimes unanswerable*.

```
1 (subtype? 'Long 'Number) ;; returns true
2 (subtype? 'String 'Long) ;; returns false
3 (subtype? '(satisfies odd?) 'Number) ;; returns :dont-know
```

- Unanswerable because:
  - Impossible to compute, *e.g.* `satisfies`.
  - Code is incomplete.
  - Loading classes at run-time.
- Determine habitation/vacuity
  - $A \subset \emptyset$
- Determine disjointness
  - Disjoint: $A \subset \overline{B}$
  - Disjoint: $A \cap B \subset \emptyset$

Challenge №3: DFA Construction

Given an RTE, generate a finite automaton.

- Well-known classical technique: Brzozowski Derivative
- Must be adapted to work with RTEs

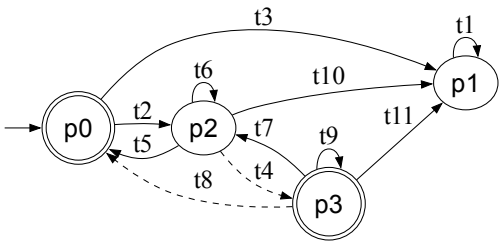
RTE Construction: $(int \cdot str^* \cdot even)^*$

$p_0 = (int \cdot str^* \cdot even)^*$

$p_1 = \emptyset$

$p_2 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^*$

$p_3 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^* + (int \cdot str^* \cdot even)^*$



$t_1 = \Sigma$

$t_2 = int$

$t_3 = \overline{int}$

$t_4 = str \cap even$

$t_5 = \overline{str} \cap even$

$t_6 = str \cap \overline{even}$

$t_7 = (int \cap \overline{even}) \cup (str \cap \overline{even})$

$t_8 = \overline{int} \cap \overline{str} \cap even$

$t_9 = (int \cap even) \cup (str \cap even)$

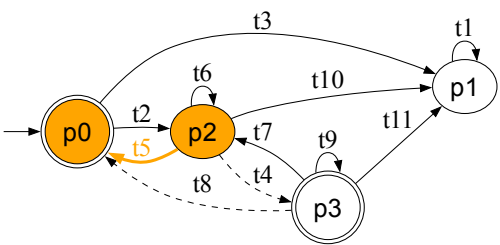
$t_{10} = \overline{str} \cap \overline{even}$

$t_{11} = \overline{int} \cap \overline{str} \cap \overline{even}$

RTE Construction: $(int \cdot str^* \cdot even)^*$

Q: How to compute a transition: $p_2 \xrightarrow{t_5} p_0$?

$$p_0 = (int \cdot str^* \cdot even)^*$$
$$p_1 = \emptyset$$
$$p_2 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^*$$
$$p_3 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^* + (int \cdot str^* \cdot even)^*$$



$$t_1 = \Sigma$$
$$t_2 = int$$
$$t_3 = \overline{int}$$
$$t_4 = str \cap even$$
$$t_5 = \overline{str} \cap even$$
$$t_6 = str \cap \overline{even}$$
$$t_7 = (int \cap \overline{even}) \cup (str \cap \overline{even})$$
$$t_8 = \overline{int} \cap \overline{str} \cap even$$
$$t_9 = (int \cap even) \cup (str \cap even)$$
$$t_{10} = \overline{str} \cap \overline{even}$$
$$t_{11} = \overline{int} \cap \overline{str} \cap \overline{even}$$

RTE Construction: $(int \cdot str^* \cdot even)^*$

Q: How to compute a transition: $p_2 \xrightarrow{t_5} p_0$?

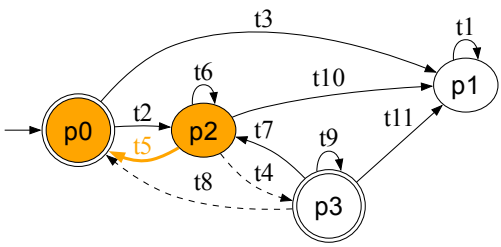
A: Construct $p_2 \xrightarrow{t_5} p_0$ that $p_0 = \partial_{t_5} p_2$

$p_0 = (int \cdot str^* \cdot even)^*$

$p_1 = \emptyset$

$p_2 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^*$

$p_3 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^* + (int \cdot str^* \cdot even)^*$



$t_1 = \Sigma$

$t_2 = int$

$t_3 = \overline{int}$

$t_4 = str \cap even$

$t_5 = \overline{str} \cap even$

$t_6 = str \cap \overline{even}$

$t_7 = (int \cap \overline{even}) \cup (str \cap \overline{even})$

$t_8 = \overline{int} \cap \overline{str} \cap even$

$t_9 = (int \cap even) \cup (str \cap even)$

$t_{10} = \overline{str} \cap \overline{even}$

$t_{11} = \overline{int} \cap \overline{str} \cap \overline{even}$

RTE Construction: $(int \cdot str^* \cdot even)^*$

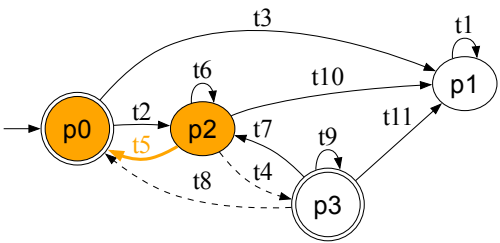
Challenge № 4: Determine $t_1, t_2, \ldots t_{13}$.

$p_0 = (int \cdot str^* \cdot even)^*$

$p_1 = \emptyset$

$p_2 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^*$

$p_3 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^* + (int \cdot str^* \cdot even)^*$



$t_1 = \Sigma$

$t_2 = int$

$t_3 = \overline{int}$

$t_4 = str \cap even$

$t_5 = \overline{str} \cap even$

$t_6 = str \cap \overline{even}$

$t_7 = (int \cap \overline{even}) \cup (str \cap \overline{even})$

$t_8 = \overline{int} \cap \overline{str} \cap even$

$t_9 = (int \cap even) \cup (str \cap even)$

$t_{10} = \overline{str} \cap \overline{even}$

$t_{11} = \overline{int} \cap \overline{str} \cap \overline{even}$

Brzozowski Derivative: $\partial_\nu\, r$

$$\begin{cases} r, s & \text{designate RTEs} \\ \nu, \mu & \text{designate types} \end{cases}. \qquad E.g., \begin{cases} r & = (int \cdot str^* \cdot even)^* \\ \nu & = int \end{cases}.$$

Recursive Rules	**Terminal Rules**
$\partial_\nu\,(r^*) = (\partial_\nu\, r) \cdot r^*$	
$\partial_\nu\,(r + s) = \partial_\nu\, r \;+\; \partial_\nu\, s$	
$\partial_\nu\,(r \,\&\, s) = \partial_\nu\, r \;\&\; \partial_\nu\, s$	
$\partial_\nu\,(r \cdot s) = \begin{cases} (\partial_\nu\, r) \cdot s & \text{if } () \notin [\![r]\!] \\ (\partial_\nu\, r) \cdot s + \partial_\nu\, s & \text{if } () \in [\![r]\!] \end{cases}$	
$\partial_\nu !r = !\partial_\nu\, r$	

Brzozowski Derivative: $\partial_\nu\, r$

$$\begin{cases} r, s & \text{designate RTEs} \\ \nu, \mu & \text{designate types} \end{cases}. \qquad E.g., \begin{cases} r & = (int \cdot str^* \cdot even)^* \\ \nu & = int \end{cases}.$$

Recursive Rules	Terminal Rules
$\partial_\nu (r^*) = (\partial_\nu r) \cdot r^*$	
$\partial_\nu (r + s) = \partial_\nu r \ + \ \partial_\nu s$	
$\partial_\nu (r \,\&\, s) = \partial_\nu r \ \&\ \partial_\nu s$	
$\partial_\nu (r \cdot s) = \begin{cases} (\partial_\nu r) \cdot s & \text{if } () \notin [\![r]\!] \\ (\partial_\nu r) \cdot s + \partial_\nu s & \text{if } () \in [\![r]\!] \end{cases}$	$\partial_\nu \lfloor \mu \rfloor = \varepsilon \quad \text{if } [\![\nu]\!] \subseteq [\![\mu]\!]$
$\partial_\nu !r = !\partial_\nu r$	$\partial_\nu \lfloor \mu \rfloor = \emptyset \quad \text{if } [\![\nu]\!] \cap [\![\mu]\!] = \emptyset$
	$\partial_\nu \lfloor \mu \rfloor$ otherwise, ERROR!

Brzozowski Derivative: $\partial_\nu\, r$

Recursive Rules	Terminal Rules
$\partial_\nu\,(r^*) = (\partial_\nu\, r)\cdot r^*$	$\partial_\nu\,\lfloor\mu\rfloor = \varepsilon \quad \text{if } [\![\nu]\!] \subseteq [\![\mu]\!]$
$\partial_\nu\,(r+s) = \partial_\nu\, r \;+\; \partial_\nu\, s$	$\partial_\nu\,\lfloor\mu\rfloor = \emptyset \quad \text{if } [\![\nu]\!] \cap [\![\mu]\!] = \emptyset$
$\partial_\nu\,(r\,\&\,s) = \partial_\nu\, r \;\&\; \partial_\nu\, s$	$\partial_\nu\,\lfloor\mu\rfloor \quad$ otherwise, ERROR!
$\partial_\nu\,(r\cdot s) = \begin{cases}(\partial_\nu\, r)\cdot s & \text{if } () \notin [\![r]\!] \\ (\partial_\nu\, r)\cdot s + \partial_\nu\, s & \text{if } () \in [\![r]\!]\end{cases}$	
$\partial_\nu !r = !\partial_\nu\, r$	

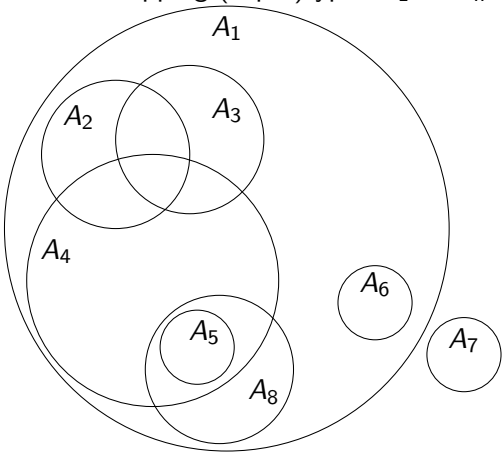
Houston we have a problem: Subtype relation *may be unknowable*.

Challenge №4: How to *sufficiently* partition types to assure $\partial_\nu\,\lfloor\mu\rfloor$ is computable?

Challenge №4: A Sufficient Type Partitioning

MDTD: Maximal Disjoint Type Decomposition

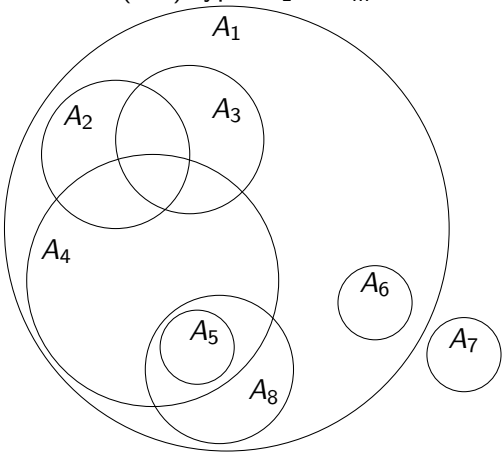
Given overlapping (super)types $A_1 \ldots A_n$ in an RTE

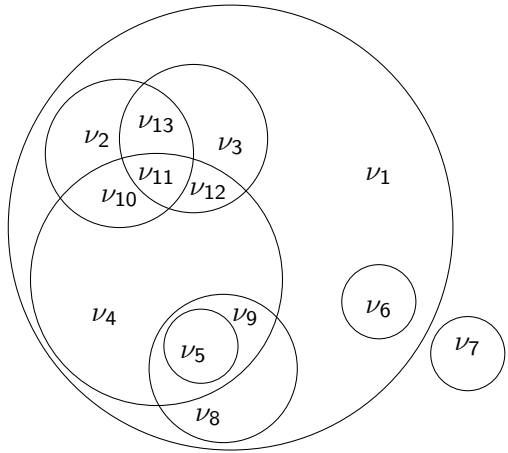


Challenge №4: A Sufficient Type Partitioning

MDTD: Maximal Disjoint Type Decomposition

Construct (sub) types $\nu_1 \ldots \nu_m$

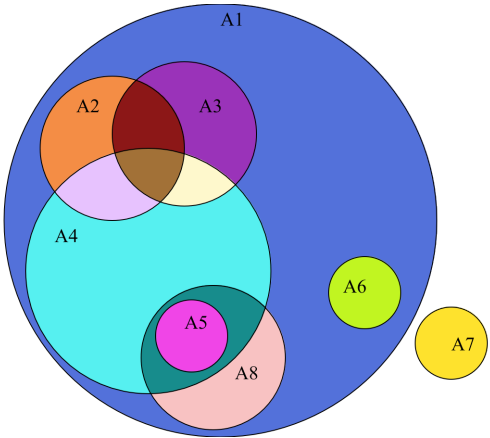


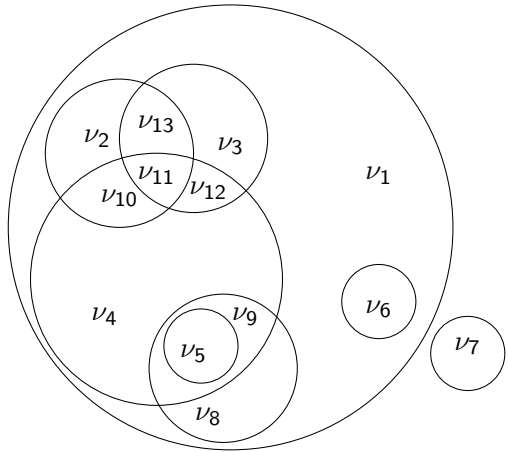


Challenge №4: A Sufficient Type Partitioning

MDTD: Maximal Disjoint Type Decomposition

Which are disjoint *by construction*, even if subtype relation is unknown.

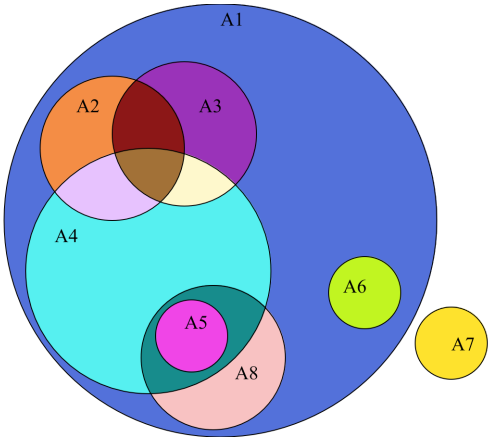


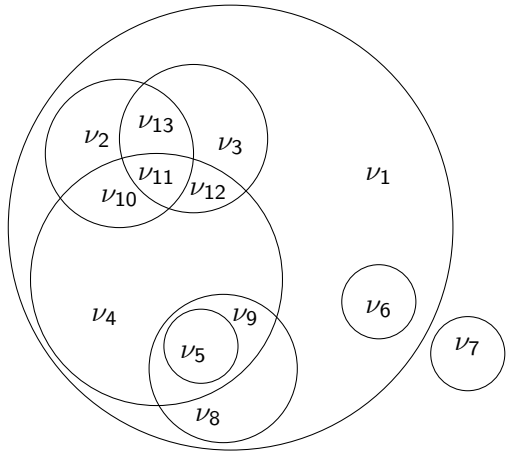


Challenge №4: A Sufficient Type Partitioning

MDTD: Maximal Disjoint Type Decomposition

Assures that $\partial_\nu r$ is computable.

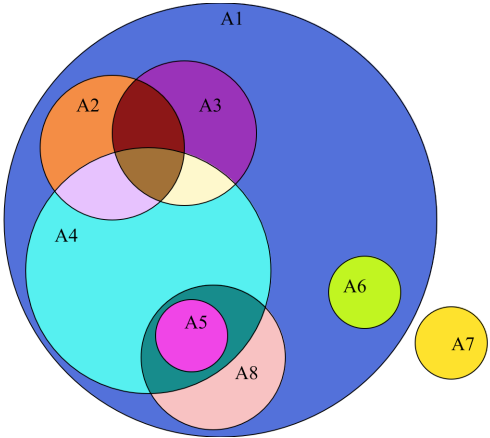


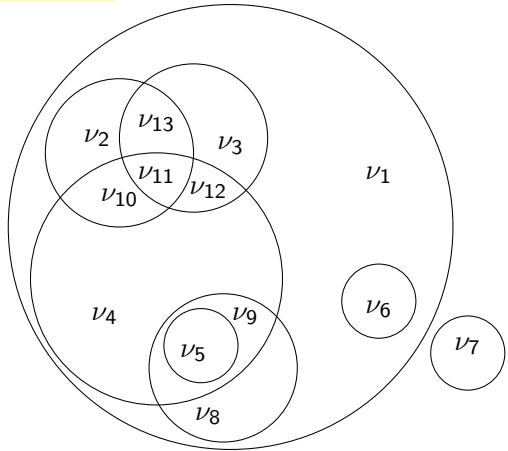


Challenge №4: A Sufficient Type Partitioning

MDTD: Maximal Disjoint Type Decomposition

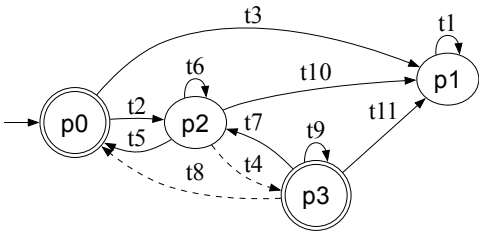
But transitions may be *indeterminate* and *unsatisfiable*.





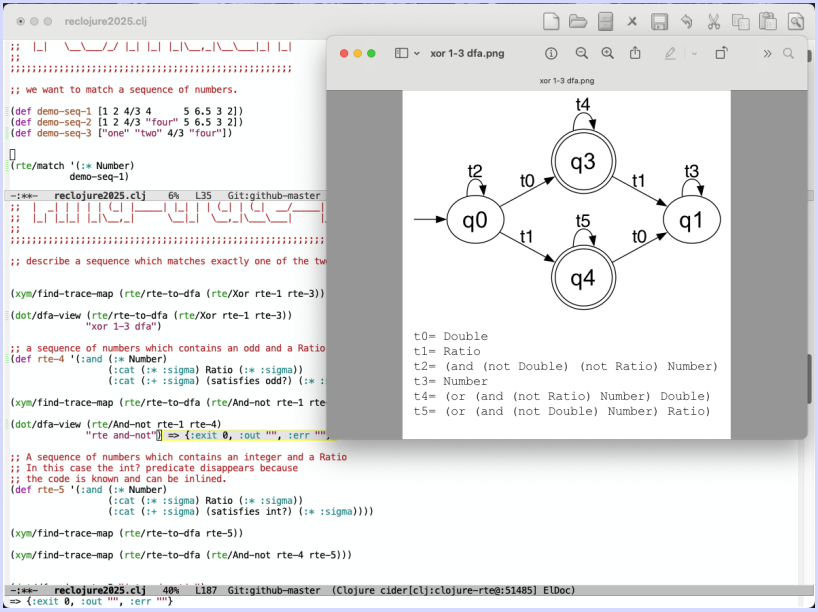
DFA with indeterminate transitions: $(int \cdot str^* \cdot even)^*$

$$p_0 = (int \cdot str^* \cdot even)^*$$
$$p_1 = \emptyset$$
$$p_2 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^*$$
$$p_3 = str^* \cdot even \cdot (int \cdot str^* \cdot even)^* + (int \cdot str^* \cdot even)^*$$



$$t_1 = \Sigma$$
$$t_2 = int$$
$$t_3 = \overline{int}$$
$$t_4 = str \cap even$$
$$t_5 = \overline{str} \cap even$$
$$t_6 = str \cap \overline{even}$$
$$t_7 = (int \cap \overline{even}) \cup (str \cap \overline{even})$$
$$t_8 = \overline{int} \cap \overline{str} \cap even$$
$$t_9 = (int \cap even) \cup (str \cap even)$$
$$t_{10} = \overline{str} \cap \overline{even}$$
$$t_{11} = \overline{int} \cap \overline{str} \cap \overline{even}$$

Demo


reclojure2025.clj
xor 1-3 dfa.png
xor 1-3 dfa.png
t4
q3
t2
t0
t1
t3
q0
q1
t5
t1
t0
q4
t0= Double
t1= Ratio
t2= (and (not Double) (not Ratio) Number)
t3= Number
t4= (or (and (not Ratio) Number) Double)
t5= (or (and (not Double) Number) Ratio)
;; we want to match a sequence of numbers.
(def demo-seq-1 [1 2 4/3 4 5 6.5 3 2])
(def demo-seq-2 [1 2 4/3 "four" 5 6.5 3 2])
(def demo-seq-3 ["one" "two" 4/3 "four"])
(rte/match '(:* Number)
demo-seq-1)
reclojure2025.clj 6% L35 Git:github-master
;; describe a sequence which matches exactly one of the tw
(xym/find-trace-map (rte/rte-to-dfa (rte/Xor rte-1 rte-3)))
(dot/dfa-view (rte/rte-to-dfa (rte/Xor rte-1 rte-3))
"xor 1-3 dfa")
;; a sequence of numbers which contains an odd and a Ratio
(def rte-4 '(:and (:* Number)
(:cat (:* :sigma) Ratio (:* :sigma))
(:cat (:+ :sigma) (satisfies odd?) (:* :
(xym/find-trace-map (rte/rte-to-dfa (rte/And-not rte-1 rte
(dot/dfa-view (rte/And-not rte-1 rte-4)
"rte and-not") => {:exit 0, :out "", :err ""}
;; A sequence of numbers which contains an integer and a Ratio
;; In this case the int? predicate disappears because
;; the code is known and can be inlined.
(def rte-5 '(:and (:* Number)
(:cat (:* :sigma) Ratio (:* :sigma))
(:cat (:+ :sigma) (satisfies int?) (:* :sigma))))
(xym/find-trace-map (rte/rte-to-dfa rte-5))
(xym/find-trace-map (rte/rte-to-dfa (rte/And-not rte-4 rte-5)))
reclojure2025.clj 40% L187 Git:github-master (Clojure cider[clj:clojure-rte@:51485] ElDoc)
=> {:exit 0, :out "", :err ""}

Conclusion

- Efficient pattern recognition for `Seq[Any]` in several programming languages
  - Extended Brzozowski derivative for Symbolic Finite Automata
  - Robust support for indeterminate subtype relation
  - Application: type based pattern matching
  - Detecting unreachable code
- Available here:

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

- *Critical Feedback Welcome!*
  - `jimka.issy@gmail.com`
  - Discord: jimka2001
  - Github: jimka2001

Determining whether two classes are disjoint

```
1  (isa? Integer Number) ;; true
2  (isa? Number Integer) ;; false
3
4  (:flags (refl/type-reflect Number))
5  ;; --> #{:public :abstract}
6
7  (:flags (refl/type-reflect Integer))
8  ;; --> #{:public :final}
9
10 (:flags (refl/type-reflect
       IPersistentVector))
11 ;; --> #{:interface :public :abstract}
```

- Explicit subtype relation `isa?`.

Determining whether two classes are disjoint

```
1  (isa? Integer Number) ;; true
2  (isa? Number Integer) ;; false
3
4  (:flags (refl/type-reflect Number))
5  ;; --> #{:public :abstract}
6
7  (:flags (refl/type-reflect Integer))
8  ;; --> #{:public :final}
9
10 (:flags (refl/type-reflect
       IPersistentVector))
11 ;; --> #{:interface :public :abstract}
```

- Explicit subtype relation `isa?`.
- Else if either is `:final`; they have no common *inhabited* subclass.

Determining whether two classes are disjoint

```
1  (isa? Integer Number) ;; true
2  (isa? Number Integer) ;; false
3
4  (:flags (refl/type-reflect Number))
5  ;; --> #{:public :abstract}
6
7  (:flags (refl/type-reflect Integer))
8  ;; --> #{:public :final}
9
10 (:flags (refl/type-reflect
       IPersistentVector))
11 ;; --> #{:interface :public :abstract}
```

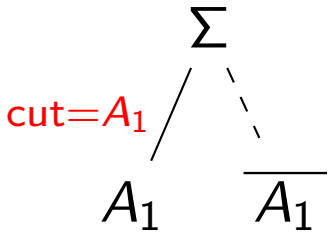
- Explicit subtype relation `isa?`.
- Else if either is `:final`; they have no common *inhabited* subclass.
- Else if either is `:interface`, there may be class which inherits from both?

Determining whether two classes are disjoint

```
1  (isa? Integer Number) ;; true
2  (isa? Number Integer) ;; false
3
4  (:flags (refl/type-reflect Number))
5  ;; --> #{:public :abstract}
6
7  (:flags (refl/type-reflect Integer))
8  ;; --> #{:public :final}
9
10 (:flags (refl/type-reflect
       IPersistentVector))
11 ;; --> #{:interface :public :abstract}
```

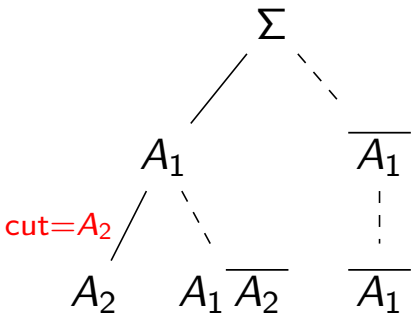
- Explicit subtype relation `isa?`.
- Else if either is `:final`; they have no common *inhabited* subclass.
- Else if either is `:interface`, there may be class which inherits from both?
- If *currently* no subclass exists, JVM might *later* run-time load `subclass.jar`

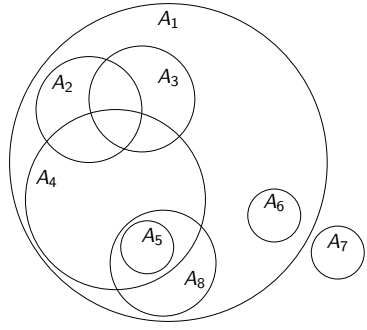
$MDTD(\{A_1, A_2, \ldots, A_5\}) \rightarrow \Pi$



Start with $\Sigma = \textit{Universe}$, and intersect with A_1 and $\overline{A_1}$.

$MDTD(\{A_1, A_2, \ldots, A_5\}) \rightarrow \Pi$





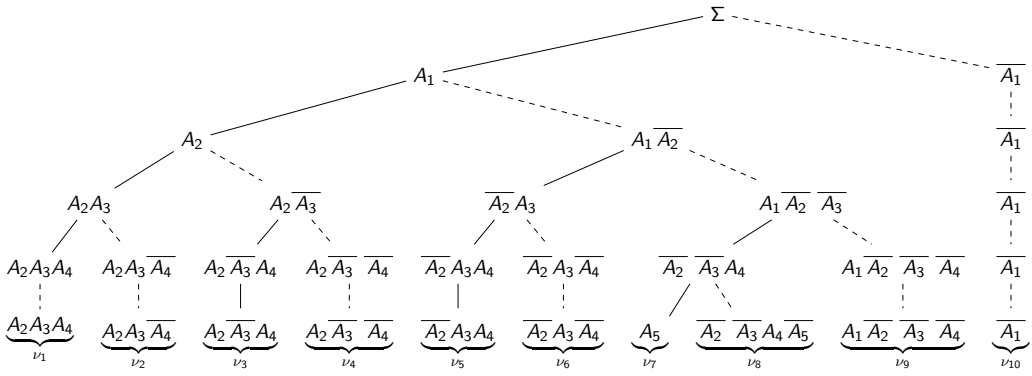
Intersect each leaf with A_2 and $\overline{A_2}$.

$$A_2 \subset A_1 \implies A_1 \cap A_2 = A_2 \qquad \textit{SIMPLIFY}$$

$$A_2 \subset A_1 \implies \overline{A_1} \cap A_2 = \emptyset \qquad \textit{PRUNE}$$

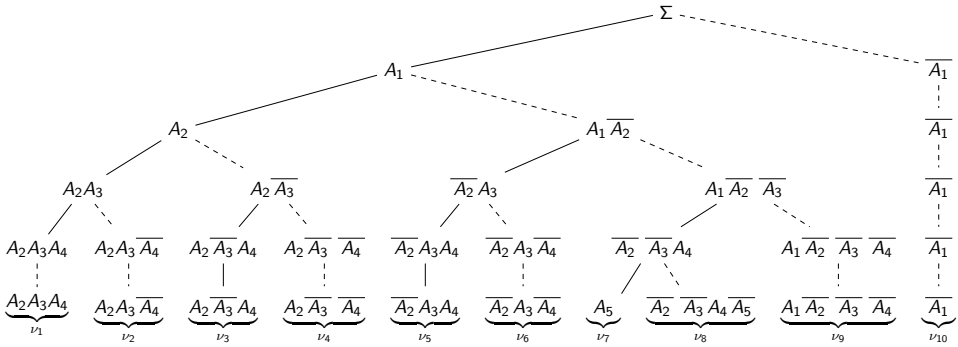
$$A_2 \subset A_1 \implies \overline{A_1} \cap \overline{A_2} = \overline{A_1} \qquad \textit{SIMPLIFY}$$

$MDTD(\{A_1, A_2, \ldots, A_5\}) \rightarrow \Pi$



$$MDTD = \Pi \xrightarrow{\leq O(2^n)} \{\nu_1, \nu_2, \ldots, \nu_{10}\} = \{A_2A_3A_4,\ A_2A_3\overline{A_4},\ \ldots,\ \overline{A_1}\}.$$

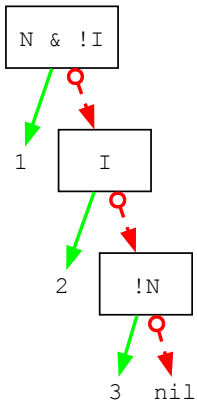
$MDTD(\{A_1, A_2, \ldots, A_5\}) \to \Pi$



- *By Construction:* either $\nu_i \subseteq A_j$ or $\nu_i \subseteq \overline{A_j}$.
- *By Construction:* $\{\nu_1, \nu_2, \ldots, \nu_{10}\}$ mutually disjoint.
- *By Construction:* $\bigcup_{i=1}^{5} \nu_i = \Sigma$.
- Exactly the properties we need to compute $\partial_\nu \lfloor A_i \rfloor$.

Decision Tree Structure

We programmatically manipulate `if then ... else ...` using a decision tree.

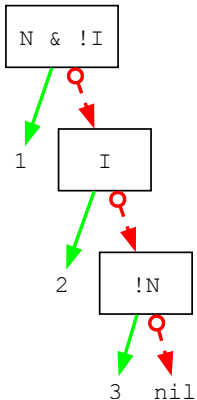


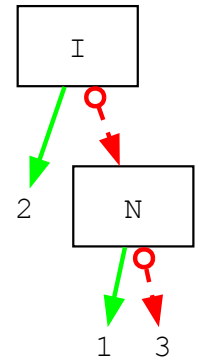

```
1  (typecase x
2    (and Number (not Long))
3    1
4
5    Long
6    2
7
8    (not Number)
9    3)
```

Decision Tree, Before and After

Viewing the decision tree before/after

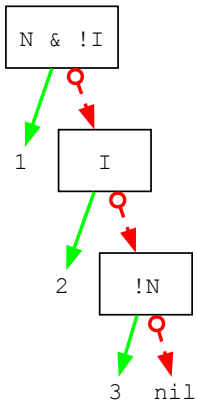
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$





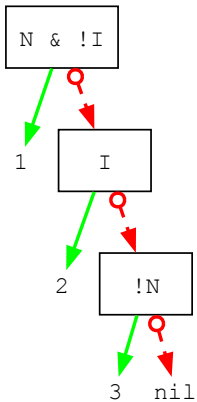
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

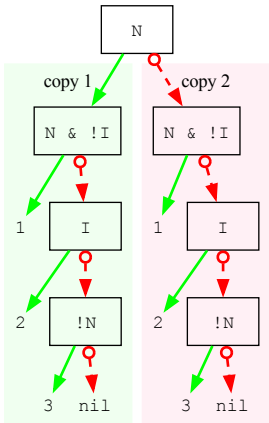
Duplicate tree, and introduce `if (typep x N) then ... else ...`



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

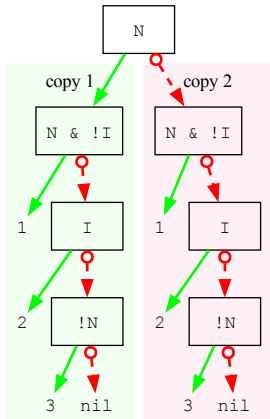
Duplicate tree, and introduce `if (typep x N) then ... else ...`





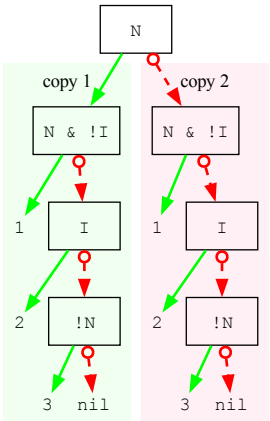
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

Duplicate tree, and introduce `if (typep x N) then ... else ...`



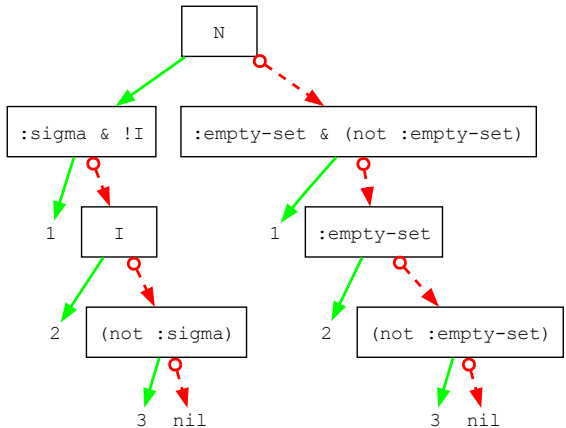
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

In `then`: Supertypes of `N` → `:sigma`. In `else`: Subtypes of `N` → `:empty-set`.



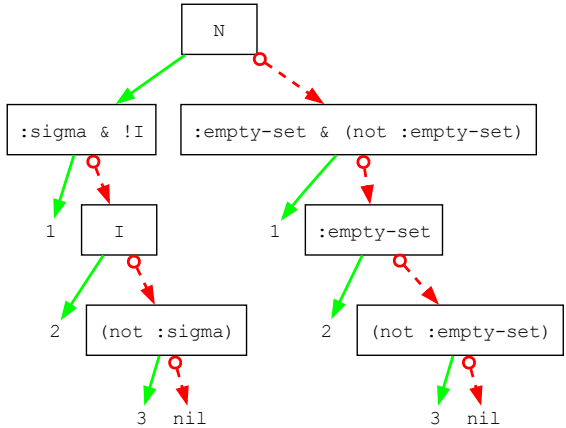
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

In `then`: Supertypes of `N` → `:sigma`. In `else`: Subtypes of `N` → `:empty-set`.



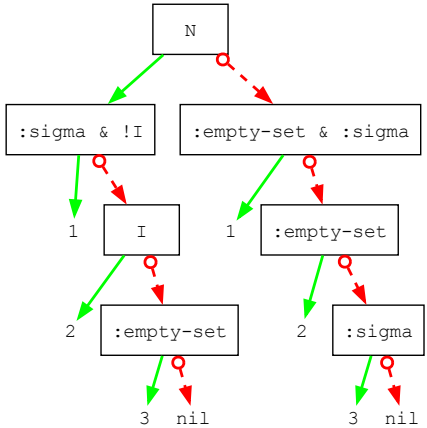
Rewrite: 1 → 2 → **3** → 4 → 5 → 6 → 7 → 8 → 9

`(not :sigma)` → `:empty-set` `(not :empty-set)` → `:sigma`



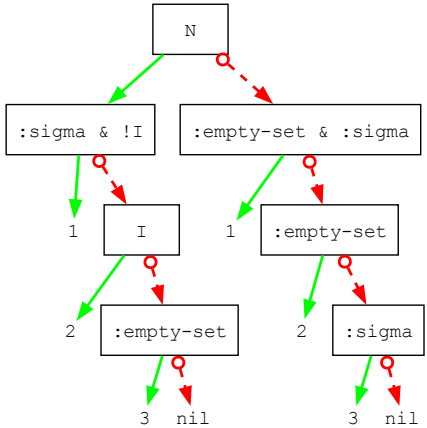
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

`(not :sigma)` → `:empty-set` `(not :empty-set)` → `:sigma`



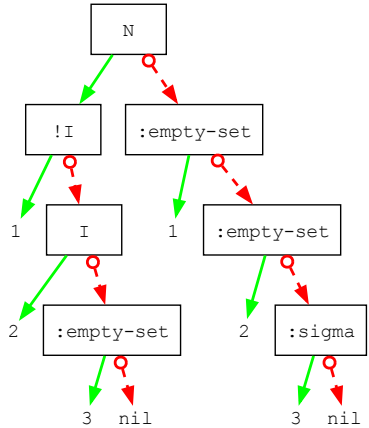
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

`(:sigma & x)` → `x` `(:empty-set & x)` → `:empty-set`



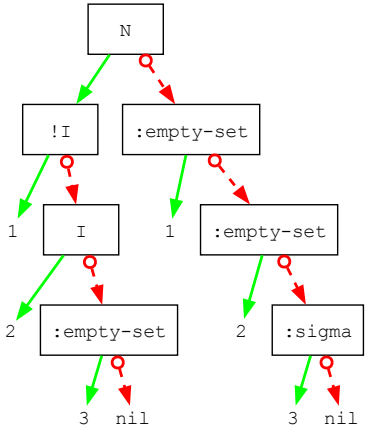
Rewrite: 1 → 2 → 3 → **4** → 5 → 6 → 7 → 8 → 9

`(:sigma & x)` → `x` `(:empty-set & x)` → `:empty-set`



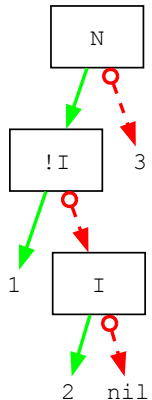
Rewrite: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$

Replace `:sigma` with `then` branch. Replace `:empty-set` with `else` branch.



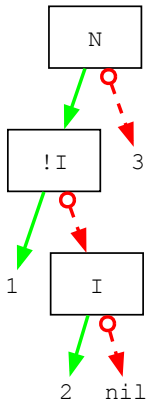
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

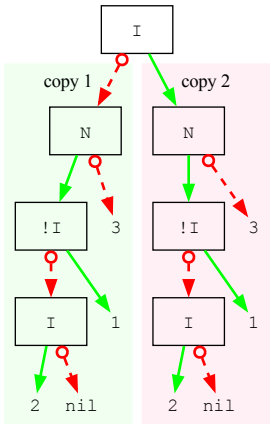
Replace `:sigma` with `then` branch. Replace `:empty-set` with `else` branch.



Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

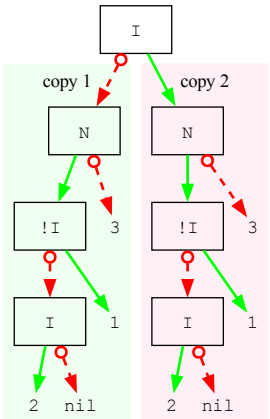
Duplicate tree, and introduce `if (typep x I) then ... else ...`





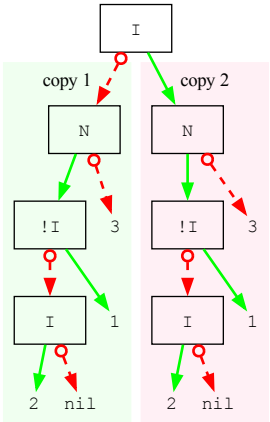
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

Duplicate tree, and introduce `if (typep x I) then ... else ...`



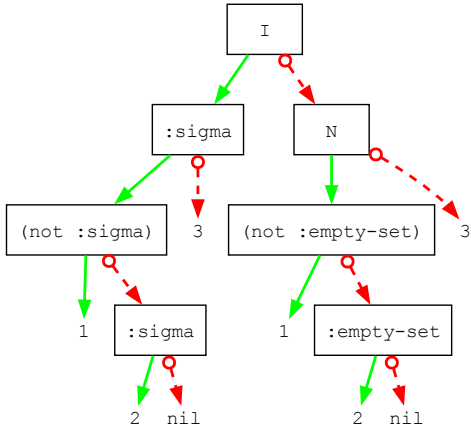
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

In `then`: Supertypes of `I` $\to$ `:sigma`. In `else`: Subtypes of `I` $\to$ `:empty-set`.



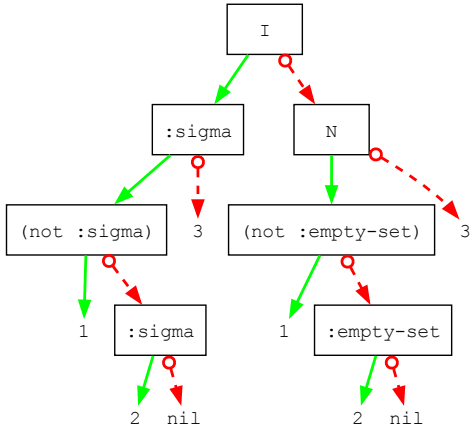
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

In `then`: Supertypes of `I` $\to$ `:sigma`. In `else`: Subtypes of `I` $\to$ `:empty-set`.



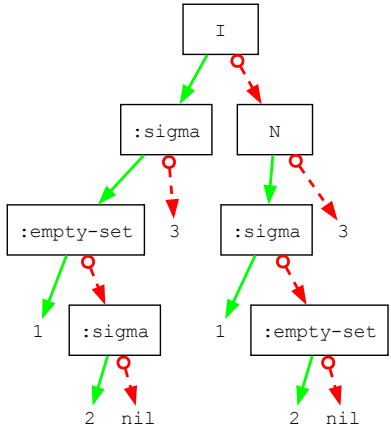
Rewrite: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

`(not :sigma)` $\to$ `:empty-set` `(not :empty-set)` $\to$ `:sigma`



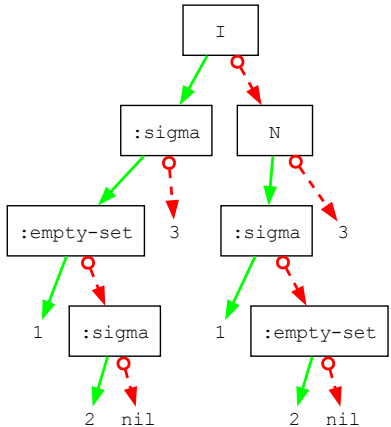
Rewrite: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

`(not :sigma)` → `:empty-set` `(not :empty-set)` → `:sigma`



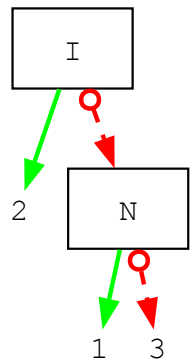
Rewrite: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$

Replace `:sigma` with `then` branch. Replace `:empty-set` with `else` branch.



Rewrite: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$

Replace `:sigma` with `then` branch. Replace `:empty-set` with `else` branch.



Rewrite: Summary

Code has been rewritten so that *any type check occurs no more than once*.

```
(typecase x
  (and Number (not Long))
  1

  Long
  2

  (not Number)
  3)
```

```
(if (typep x 'Long)
    2
    (if (typep x 'Number)
        1
        3))
```

And it is clear the code never returns `nil`.

Setup

Declaration Clojure Code

```
(def pat-1
  '(:* (:cat String
             (:* (? even)))))
(def pat-2
  '(:* (:cat String
             (:* (and Integer
                      (? even))))))
(def pat-3
  '(:* (:cat String
             (:* (and Integer
                      (? even)
                      (? large))))))
```

```
(defn large [n]
  (and (typep n 'Integer)
       (> n 127)))

(defn even [n]
  (and (typep n 'Integer)
       (= 0 (mod n 2))))
```

Setup

Declaration Clojure Code

```
(def pat-1
  '(:* (:cat String
             (:* (? even)))))
(def pat-2
  '(:* (:cat String
             (:* (and Integer
                      (? even))))))
(def pat-3
  '(:* (:cat String
             (:* (and Integer
                      (? even)
                      (? large))))))
```

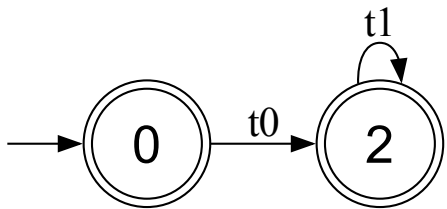
```
(def data ["Lorem" "ipsum"
           2 4 8
           "dolor" 10 20])

(rte/match pat-1 data) ;; --> true

(rte/match pat-2 data) ;; --> true

(rte/match pat-3 data) ;; --> false
```

DFAs

```
1  (let [pat-1 '(:* (:cat String (:* (? even))))]
2    (dfa-to-dot pat-1 :view true))
```

$$t_0 = \texttt{String}$$

$$t_1 = \texttt{(or String (? even))}$$

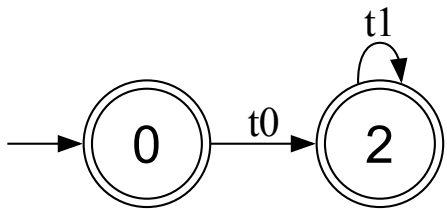


DFAs

```
1 (let [pat-2 '(:* (:cat String (:* (and Integer (? even)))))]
2   (dfa-to-dot pat-2 :view true))
```

$$t_0 = \texttt{String}$$

$$t_1 = \texttt{(or String (and Integer (? even)))}$$

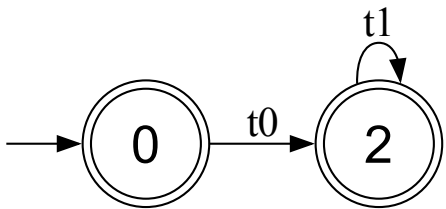


DFAs

```
1 (let [pat-3 '(:* (:cat String (:* (and Integer (? even) (? large)))))]
2   (dfa-to-dot pat-3 :view true))
```

$t_0 = \texttt{String}$

$t_1 = \texttt{(or String (and Integer (? even) (? large)))}$



Inclusion Checks

```
1 (def pat-1 '(:* (:cat String (:* (? even)))))
2 (def pat-2 '(:* (:cat String (:* (and Integer (? even))))))
3 (def pat-3 '(:* (:cat String (:* (and Integer (? even) (? large))))))
```

By inspection we see that

$$[\![pat3]\!] \subsetneq [\![pat2]\!] = [\![pat1]\!] .$$

Can we determine this programmatically?

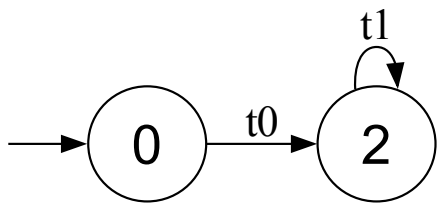
Inclusion Checks

$$[\![pat3]\!] \subsetneq \underbrace{[\![pat2]\!] = [\![pat1]\!]}_{[\![pat2]\!] \subset [\![pat1]\!] = \mathit{YES}}$$

$t_0 = \texttt{String}$

$t_1 = \texttt{(or String (and Integer (? even)))}$

```
1 (let [pat-1 '(:* (:cat String (? even)))
2       pat-2 '(:* (:cat String (and Long
3                                    (? even))))
4       diff-21 (gns/- pat-2 pat-1)]
5
6  ;; emptyset if pat2 subset of pat1
7  (dfa-to-dot diff-21 :view true))
```



Inclusion Checks

$$[\![pat3]\!] \subsetneq \underbrace{[\![pat2]\!] = [\![pat1]\!]}_{[\![pat2]\!] \supset [\![pat1]\!] = DONT-KNOW}$$

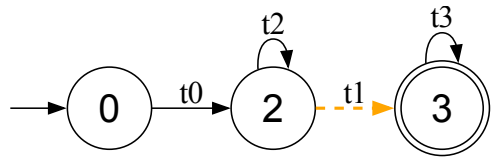
```
1 (let [pat-1 '(:* (:cat String (? even)))
2       pat-2 '(:* (:cat String (and Long (?
            even))))
3       diff-12 `(and ~pat-1 (not ~pat-2))]
4
5  ;; emptyset if pat1 subset of pat2
6  (dfa-to-dot diff-12 :view true))
```

$t_0 = \texttt{S}$

$t_1 = \texttt{!I \& !S \& Even}$ *Indeterminate*

$t_2 = \texttt{S | (I \& Even)}$

$t_3 = \texttt{S | Even}$



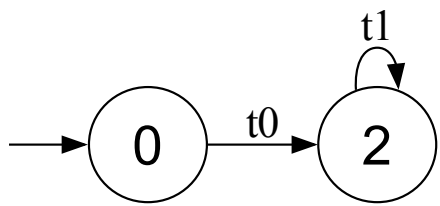
Inclusion Checks

$$\underbrace{[\![pat3]\!] \subsetneq [\![pat2]\!]}_{YES} = [\![pat1]\!]$$

```
1  (let [pat-1 '(:* String (:* (? even)))
2        pat-2 '(:* (:cat String (and Long (?
             even))))
3        pat-3 '(:* (:cat String (and Long
4                                    (? even)
5                                    (? large))))
6        diff-32 (template (and ~pat-3 (not
             ~pat-2)))]
7
8    ;; emptyset if pat3 subset of pat2
9    (dfa-to-dot diff-32 :view true))
```

$t_0 = \texttt{S}$

$t_1 = \texttt{S | (I \& Even, Large)}$



Inclusion Checks

$$[\![pat2]\!] \cap \overline{[\![pat3]\!]}$$

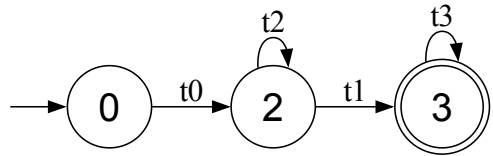
```
1 (let [pat-1 '(:* (:cat String (:* (? even))))
2       pat-2 '(:* (:cat String (:* (and Long (?
            even)))))
3       pat-3 '(:* (:cat String (:* (and Long (?
            even) (? large)))))
4       diff-23 (gns/- pat-2 pat-3)]
5
6   ;; Example of element in pat2 but not pat3
7   (dfa-to-dot diff-23 :view true))
```

$t_0 = \texttt{S}$

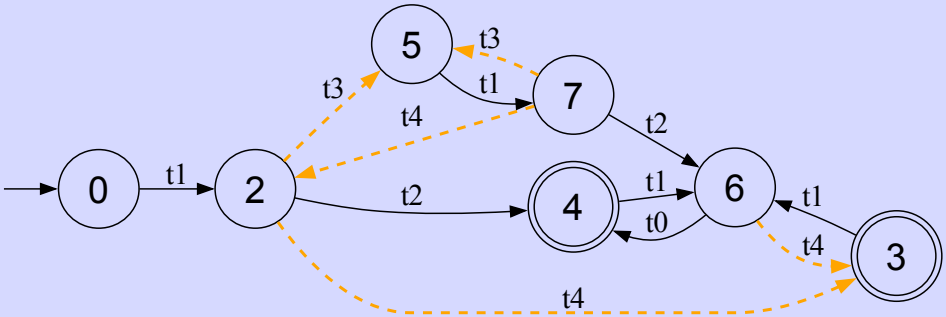
$t_1 = \texttt{I \& Even \& !Large}$

$t_2 = \texttt{S | (I \& Even \& Large)}$

$t_3 = \texttt{S | (I \& Even)}$



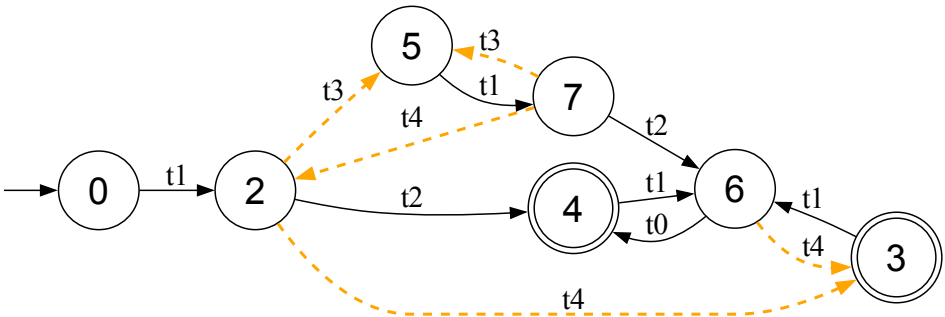
Challenge №6: How to detect habitation/vacuity in a DFA



Habitation Check

Is the language of the DFA inhabited?

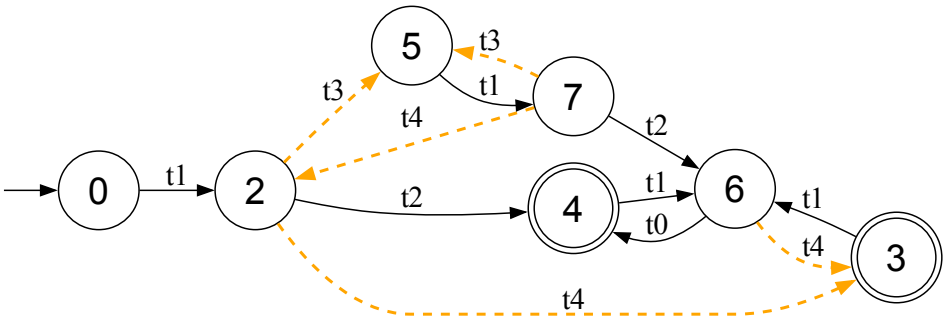
- YES – satisfiable path to accepting state



Habitation Check

Is the language of the DFA inhabited?

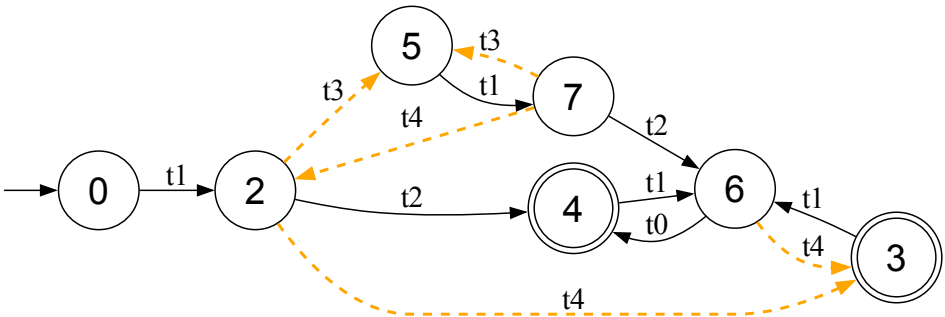
- YES – satisfiable path to accepting state
- NO – no path to accepting state



Habitation Check

Is the language of the DFA inhabited?

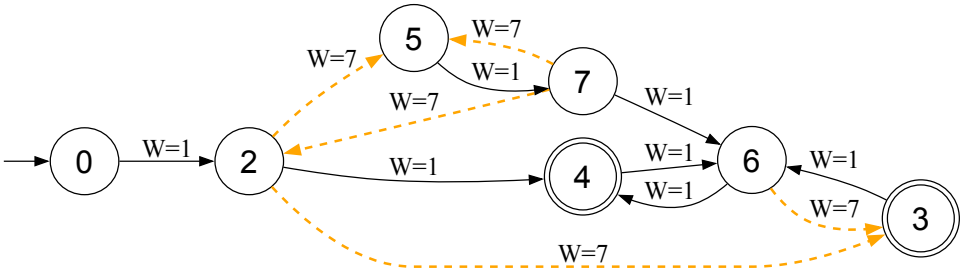
- YES – satisfiable path to accepting state
- NO – no path to accepting state
- DONT-KNOW – only indeterminate paths to accepting state



Habitation Check

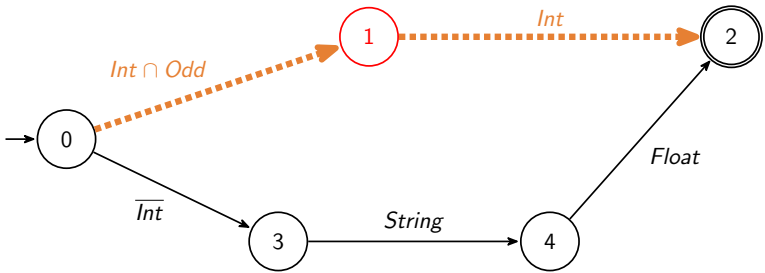
Is the language of the DFA inhabited?

- YES – satisfiable path to accepting state
- NO – no path to accepting state
- DONT-KNOW – only indeterminate paths to accepting state
- Find shortest path using Dijkstra weighted-graph search



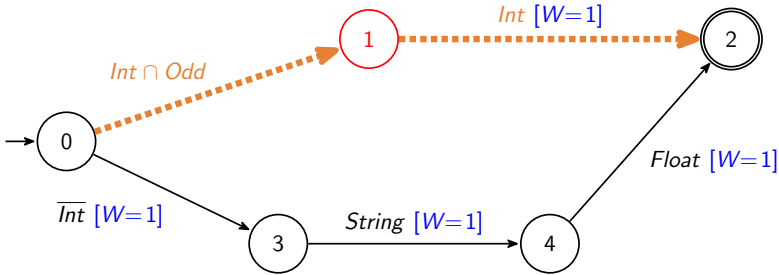
Habitation Check

How do Dijkstra weights work?



Habitation Check

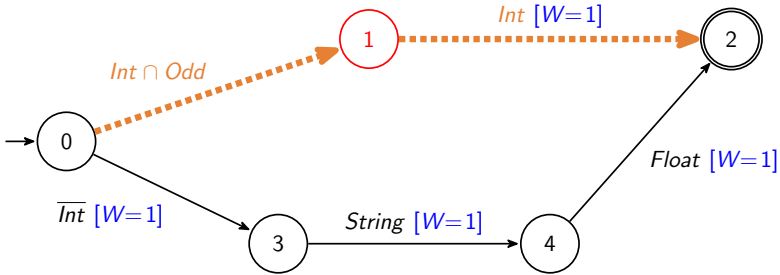
How do Dijkstra weights work?



- Satisfiable transitions have weight $W = 1$.

Habitation Check

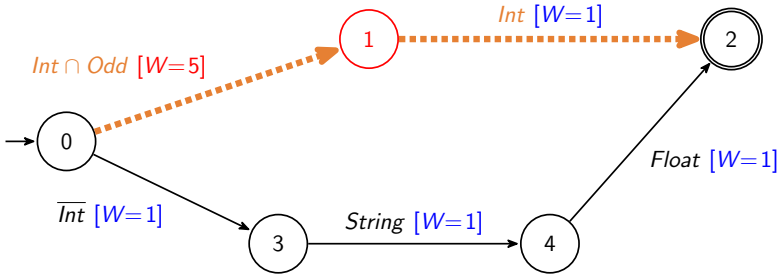
How do Dijkstra weights work?



- Satisfiable transitions have weight $W = 1$.
- Unsatisfiable transitions have weight $W = \infty$.

Habitation Check

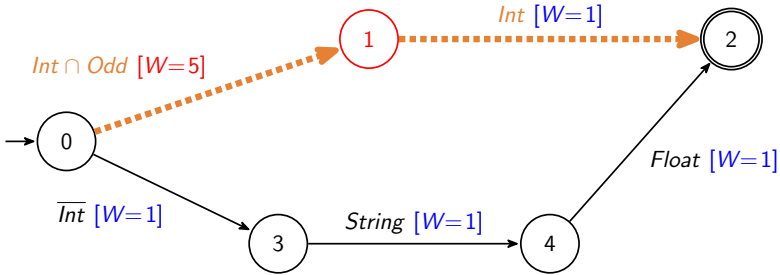
How do Dijkstra weights work?



- Satisfiable transitions have weight $W = 1$.
- Unsatisfiable transitions have weight $W = \infty$.
- Indeterminate transitions have weight $W = 5$ (number of states).

Habitation Check

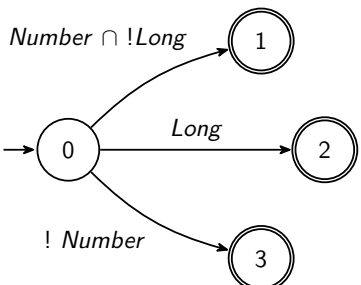
How do Dijkstra weights work?



- Satisfiable transitions have weight $W = 1$.
- Unsatisfiable transitions have weight $W = \infty$.
- Indeterminate transitions have weight $W = 5$ (number of states).
- Shortest path is $⓪ \xrightarrow{1} ③ \xrightarrow{1} ④ \xrightarrow{1} ②$; length is 3
  Longer path is $⓪ \xrightarrow{5} ① \xrightarrow{1} ②$; length is 6

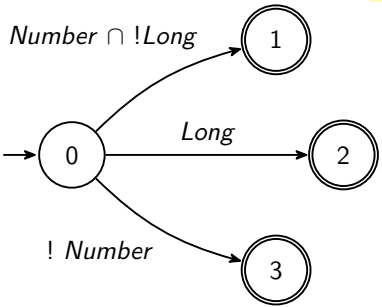
Challenge №5: Redundant Type Checks

Select correct transition, avoiding *redundant type checks*.



Sequential Type Check

A DFA state may have several *disjoint* transitions.

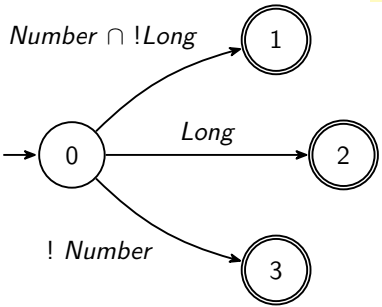



```
1  (typecase x
2    (and Number (not Long))
3    1
4
5    Long
6    2
7
8    (not Number)
9    3)
```

- Some types may be checked multiple times.

Sequential Type Check

A DFA state may have several *disjoint* transitions.




```
1  (if (typep x 'Long)
2      2
3      (if (typep x 'Number)
4          1
5          3))
```

- Some types may be checked multiple times.
- We can rewrite the code to *eliminate redundant checks*.