2. Regular Languages

Emil Sekerinski, McMaster University, January 2022

Regular Expressions

Regular expressions are closely related to regular languages and to finite state automata.

The *regular expressions* over symbols T consist of

- any symbol $a \in T \cup \{\varepsilon\}$,
- $E_1 \mid E_2$, where E_1, E_2 are regular expressions (*choice*),
- $E_1\ E_2$, where E_1, E_2 are regular expressions (*concatenation*),
- $E*$, where E is a regular expression (*repetition*).

Parenthesis (E) are used for grouping. By convention, repetition binds tighter than concatenation, which binds tighter than choice. That is, $ab*|c$ is understood as $(a(b*))|c$. The operator for repetition is called the Kleene star.

The language $L(E)$ *described* by regular expression E over set T of symbols is defined recursively over the structure of E. Assuming $a \in T \cup \{\varepsilon\}$ and E, E_1, E_2 are regular expressions,

regular expression	language
a	$\{a\}$
$E_1 \mid E_2$	$L(E_1) \cup L(E_2)$
$E_1\ E_2$	$L(E_1)\ L(E_2)$
$E*$	$\bigcup\ n \ge 0 \bullet L^n(E)$

Example.

$$\begin{aligned}
&L(ab*|c)\\
&= L(ab*) \cup L(c)\\
&= (L(a)\ L(b*)) \cup L(c)\\
&= (\{a\}\ (\bigcup\ n \ge 0 \bullet L^n(b))) \cup \{c\}\\
&= (\{a\}\ (\bigcup\ n \ge 0 \bullet \{b\}^n)) \cup \{c\}\\
&= (\bigcup\ n \ge 0 \bullet \{a\}\{b\}^n) \cup \{c\}
\end{aligned}$$

where for sets A, B of sequences over T:

$$\begin{aligned}
A\ B &= \{a\ b \mid a \in A \land b \in B\}\\
A^0 &= \{\varepsilon\}\\
A^n &= A\ A^{n-1},\ n > 0
\end{aligned}$$

As a consequence $A^1 = A\ A^0 = A\ \{\varepsilon\} = A$ and therefore:

$$L(E*) = \{\varepsilon\} \cup L(E) \cup L^2(E) \cup \ldots$$

The notation $[E] = \varepsilon \mid E$ is used to avoid writing ε. Consequently:

$$L([E]) = \{\varepsilon\} \cup L(E)$$

Sometimes $\{E\} = E*$ is used, as in EBNF.

Example.

$$\begin{aligned}
&L([a \mid b])\\
&= \{\varepsilon\} \cup L(a \mid b)\\
&= \{\varepsilon\} \cup L(a) \cup L(b)\\
&= \{\varepsilon\} \cup \{a\} \cup \{b\}\\
&= \{\varepsilon, a, b\}
\end{aligned}$$

Question. What language does $[a\ a*]$ describe? How can the expression be simplified? Give a formal proof!

Answer.

$$\begin{aligned}
&\ L([a\ a*])\\
=&\ \{\varepsilon\} \cup L(a\ a*)\\
=&\ \{\varepsilon\} \cup (L(a)\ L(a*))\\
=&\ \{\varepsilon\} \cup (\{a\}\ (\bigcup\ n \ge 0 \bullet L^n(a)))\\
=&\ \{\varepsilon\} \cup (\{a\}\ (\bigcup\ n \ge 0 \bullet \{a\}^n))\\
=&\ \{\varepsilon\} \cup (\bigcup\ n \ge 0 \bullet \{a\}\{a\}^n)\\
=&\ \{a\}^0 \cup (\bigcup\ n > 0 \bullet \{a\}^n)\\
=&\ \bigcup\ n \ge 0 \bullet \{a\}^n\\
=&\ \bigcup\ n \ge 0 \bullet L(a)^n\\
=&\ L(a*)
\end{aligned}$$

That is, $[a\ a*]$ describes the same language as $a*$.

The similarity in the notation of regular expression and productions in EBNF is justified by following theorem, given without proof:

Example. Taking $ab*|c$,

Theorem. For regular expression `E` over `T` and grammar `G` with the single non-recursive production `S → E`, the language described by `E` and the language generated by `G` are the same, `L(E) = L(G)`.

`L(ab*|c) = L(G)`

where `G = (T, N, S, P}`
with `T = {a, b, c}`, `N = {S}`,
and `P = {S → a{b}|c}`

For regular expressions, equivalence is of fundamental importance. Regular expressions `E`, `E'` are *equal*, `E = E'`, if `L(E) = L(E')`; that is, a regular expression is identified with the set of sentences it describes. The basic rules are, where `E`, `F`, `G` are regular expressions:

- *Associativity:* `(E | F) | G = E | (F | G)` and `(E F) G = E (F G)`
- *Commutativity:* `E | F = F | E`
- *Distributivity:* `E (F | G) = E F | E G` and `(E | F) G = E G | F G`
- *Idempotence:* `E | E = E`
- *Identity:* `E ε = ε E = E`

Some of the basic rules of `*` are:

- `E* = [E E*]`
- `E E* = E* E`
- `E** = E*`
- `E* E* = E*`
- `(E | F)* = E* (F E*)*`
- `(E F)* E = E (F E)*`

Question. How can you formally prove those?

Answer. Here is the proof of `E | F = F | E`:

```
  L(E | F)
=  L(E) ∪ L(F)
=  L(F) ∪ L(E)
=  L(F | E)
```

That is, the proof of the commutativity of choice relies on the commutativity of union. Here is the proof of `E (F | G) = E F | E G`:

```
  L(E (F | G))
=  L(E) L(F | G)
=  L(E) (L(F) ∪ L(G))
=  L(E) {b | b ∈ L(F) ∨ b ∈ L(G)}
=  {a b | a ∈ L(E) ∧ (b ∈ L(F) ∨ b ∈ L(G))}
=  {a b | (a ∈ L(E) ∧ b ∈ L(F)) ∨ (a ∈ L(E) ∧ b ∈ L(G))}
=  {a b | a ∈ L(E) ∧ b ∈ L(F)} ∪ {a b | a ∈ L(E) ∧ b ∈ L(G)}
=  L(E F) ∪ L(E G)
=  L(E F | E G)
```

That is, the proof of distributivity of concatenation over choice relies on the distributivity of conjunction over disjunction.

Aside. If one adds `∅` with `L(∅) = {}` as the empty language, the regular expressions form a Kleene algebra with following properties of `∅`:

- *Identity:* `E | ∅ = ∅ | E = E`
- *Annihilation:* `E ∅ = ∅ E = ∅`

Having `∅` in regular expressions for searching text is not intuitive and is not used. However, Kleene algebras have applications to programming, with `∅`, `ε` corresponding to `stop`, `skip` and with concatenation, choice, and repetition (star) corresponding to sequential composition, nondeterministic choice, and repetition. In that case, the underlying model is that of relations, with sequential composition being relational composition and nondeterministic choice being union. *Kleene algebras with tests* distinguish between statements and guards (tests) and allow while-programs to be expressed, see Kozen's tutorial Part 1 and Part 2.

Regular Grammar to Regular Expression

For every regular grammar `G` we can construct a regular expression `E` such that `L(G) = L(E)` by transforming the productions of the grammar. The assumption is that the grammar is in BNF, with one production for everyone nonterminal, including one for the start symbol `S`. The grammar is transformed by a number of steps with intemediate grammars in EBNF. An EBNF production of the form

```
A → E A | F
```

where `A` does not occur in `E`, `F`, is equivalent to (*Arden's Rule*)

```
A → E* F
```

which can be used to replace `A` by `E* F` in all other productions. This is repeated until a single production for `S` is left, whose right-hand side is the equivalent regular expression.

Example. Given regular grammar with productions

```
S → a | b X     X → b X | c Y     Y → c
```

first the last production can be eliminated by replacing `Y` with `c` in all other productions:

```
S → a | b X     X → b X | c c
```

An equivalent production for `X` is `X → b* c c` which allows `X` to be replaced by `b* c c`:

```
S → a | b b* c c
```

Thus an equivalent regular expression is `a | b⁺ c c`.

Question. What is an equivalent regular expression for the grammar with productions:

```
S → a S | b X
X → a X | b Y | a
Y → a Y | a
```

Answer.

```
S → a S | b X     X → a X | b Y | a     Y → a Y | a
```

An equivalent production for `Y` is `Y → a* a` which allows `Y` to be replaced by `a⁺`:

```
S → a S | b X     X → a X | b a⁺ | a
```

The production for `X` can be written as `X → a X | (b a⁺ | a)`, which matches the form that is needed for it be rewritten as `X → a* (b a⁺ | a)`, which in turn allows `X` to be eliminated from the other productions:

```
S → a S | b (a* (b a⁺ | a))
```

Now `S` can be equivalently defined by `S → a* b (a* (b a⁺ | a))`. The regular expression equivalent to the grammar is therefore:

```
a* b (a* (b a⁺ | a))
```

Finite State Automata

A finite state automaton `A = (T, Q, R, q₀, F)` is specified by

- a finite set `T` of *symbols*,
- a finite set `Q` of *states*,
- a finite set `R` of *transitions*,
- an *initial state* `q₀ ∈ Q`,
- a set of *final states* `F ⊆ Q`,

where each transition is a triple with a state `q ∈ Q`, a symbol `t ∈ T`, and a state `q' ∈ Q`, written:

```
q t → q'
```

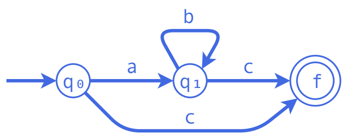
Example. `A₀ = (T, Q, R, q₀, F)` with `T = {a, b, c}`, `Q = {q₀, q₁, f}`, `F = {f}` and transitions `R`:

```
q₀ a → q₁
q₁ b → q₁
q₁ c → f
q₀ c → f
```

A finite state automaton is given a sequence `τ ∈ T*` as input and starts in its initial state. A transition `q t → q'` allows it to move from `q` to `q'` while consuming `t` from the input.

In some treatments of finite state automata `ε`-transitions are allowed, i.e. transitions of the form $q \to q'$, which can be taken without consuming a symbol from the input.

Finite state automata can be graphically represented by *finite state diagrams*:



- States are enclosed in a circle.
- Transitions are arrows between states labeled with a symbol.
- One arrow points to the initial state.
- Each final state is enclosed in a double circle.

Sequence $\tau \in T^*$ is *accepted* in state q leading to state q', written $q\ \tau \Rightarrow^* q'$, if q' can be reached in zero or more steps, each step accepting the next element from the input,

- $q \Rightarrow^* q$
- $q\ t\sigma \Rightarrow^* q'$ if $q\ t \to r$ and $r\ \sigma \Rightarrow^* q'$ for some state r.

Example. $q_0\ abc \Rightarrow^* f$ in A_0:

$q_0\ abc \Rightarrow^* f$ as $q_0\ a \to q_1$ and $q_1\ bc \Rightarrow^* f$
$q_1\ bc \Rightarrow^* f$ as $q_1\ b \to q_1$ and $q_1\ c \Rightarrow^* f$
$q_1\ c \Rightarrow^* f$ as $q_1\ c \to f$ and $f \Rightarrow^* f$

The language $L(A)$ accepted by finite state automaton $A = (T, Q, R, q_0, F)$ is the set of all sequences of symbols which can be recognized when starting from the initial state:

$$L(A) = \{\tau \in T^* \mid q_0\ \tau \Rightarrow^* q,\ q \in F\}$$

Automata A and A' are *equivalent* if they accept the same language, $L(A) = L(A')$. By extension, grammars, regular expressions, and automata are equivalent if they generate, describe, and accept the same language.

Question. What is the language accepted by A_0? What is an equivalent regular expression? What is an equivalent regular grammar?

Answer.

- Accepted language: $\{ab^nc \mid n \geq 0\} \cup \{c\}$
- Regular expression: `ab*c | c`
- Productions of regular grammar: `A → a B | c`, `B → b B | c`

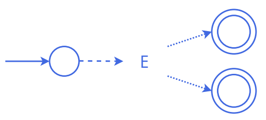
Regular Expression to Finite State Automaton

For every regular expression, we can construct an equivalent finite state automaton recursively over the structure of the expression. As regular expression `[E]` is `E | ε`, here, we now do not consider `[E]`, but allow `ε`. The *abstract syntax* of regular expressions, i.e. a grammar not defining operator precedence, is

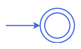
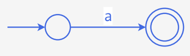
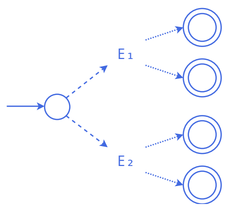
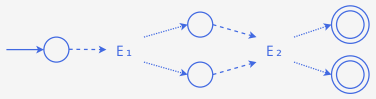
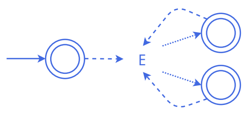
```
expr ::= 'ε' | sym | expr '|' expr | expr expr | expr '*'
```

where `sym` is any symbol.

An automaton recognizing regular expression `E` is visualized with one dashed arrow representing all transitions leaving the initial state and a one dotted arrow to each of the final states representing all transitions targeting that final state:



The table to the right illustrates the construction:

RE	automaton
ε	
a	
$E_1 \mid E_2$	
$E_1\ E_2$	
E^*	

- The automaton for `ε` consists only of an initial state, which is also the final state.
- The automaton for `a` consists of a transition on `a` from the initial to the final state.
- The automaton for $E_1 \mid E_2$ consists of all transitions of the automata for E_1 and E_2 with the initial states merged.
- The automaton for $E_1\ E_2$ consists of all transitions of the automata for E_1 and E_2 but with the transitions leaving the initial state of E_2 replicated to leave all final states of E_1. If the initial state of E_2 was final, then the final states of E_1 become final.
- The automaton for E^* consists of all transitions of the automaton for E but with the transitions leaving the initial state of E replicated to leave all final states of E. The initial state of E becomes also a final state.

Question. What are the steps to construct an automaton for `(ab)*|c` ?

With the execution of finite state automata in mind, we define a Python class `FiniteStateAutomaton` and a procedure for creating a `FiniteStateAutomaton` object from a textual representation of following form, where state names are strings without space:

```
initial
final, final, ...
source symbol → target
source symbol → target
...
```

The vocabulary becomes all the symbols that appear in the transitions. The set of states becomes all the inital states, final states, and states mentioned in transitions. Empty lines are allowed. Symbols must be individual characters and states can be arbitrary strings. Later on, sets of sets of states will be considered. Python does not allow a `set` object with `set` elements to be constructed: elements have to have a hash method defined to allow fast checking of membership, but that is only defined for immutable data types like `str` and `frozenset`, not for `set`. Therefore `frozenset` is used throughout. To make the programs use the more readable notation for `set`, the class `set` is redefined to be `frozenset`:

In [ ]:
```python
class set(frozenset):
    def __repr__(self):
        return '{' + ', '.join(str(e) for e in self) + '}'

class FiniteStateAutomaton:
    def __init__(self, T, Q, R, q0, F):
        self.T, self.Q, self.R, self.q0, self.F = T, Q, R, q0, F
    def __repr__(self):
        return str(self.q0) + '\n' + ' '.join(str(f) for f in self.F) + '\n' + \
            '\n'.join(str(q) + ' ' + a + ' → ' + str(r) for (q, a, r) in self.R)

def parseFSA(fsa: str) -> FiniteStateAutomaton:
    fsa = [line for line in fsa.split('\n') if line.strip() != '']
    q0 = fsa[0] # first line: initial
    F = set(fsa[1].split()) # second line: final, final, ...
    R = set()
    for line in fsa[2:]: # all subsequent lines: "source symbol → target"
        l, r = line.split('→')
        R |= {(l.split()[0], l.split()[1], r.split()[0])}
    T = {r[1] for r in R}
    Q = {q0} | F | {r[0] for r in R} | {r[2] for r in R}
    return FiniteStateAutomaton(T, Q, R, q0, F)
```

For example, for `A₀` as earlier:

In [ ]:
```python
A0 = parseFSA("""
q₀
f
q₀ a → q₁
q₁ b → q₁
q₁ c → f
q₀ c → f
"""); A0
```

In Python we use classes to construct regular expressions: constructors `Choice(e1, e2)`, `Conc(e1, e2)`, `Star(e)` create objects that represent `e1 | e2`, `e1 e2`, and `e*`, respectively:

In [ ]:
```python
class Choice:
    def __init__(self, e1, e2): self.e1, self.e2 = e1, e2
    def __repr__(self): return '(' + str(self.e1) + '|' + str(self.e2) + ')'

class Conc:
    def __init__(self, e1, e2): self.e1, self.e2 = e1, e2
    def __repr__(self): return '(' + str(self.e1) + str(self.e2) + ')'

class Star:
    def __init__(self, e): self.e = e
    def __repr__(self): return '(' + str(self.e) + ')*'
```

Symbols are represented by characters, with `''` representing `ε`. For example, `(ab)*|c` is represented as:

In [ ]:
```python
E1 = Choice(Star(Conc('a', 'b')), 'c'); E1
```

The Python implementation below closely follows the algorithm. All states are given unique numbers by maintaining a global counter, `QC`, with the next state name. In the case of `Choice`, the initial states are unified by taking that of one of the automata and renaming the initial state in the other automaton.

In [ ]:
```python
def REToFSA(re) -> FiniteStateAutomaton:
    global QC
    if re == '': q = QC; QC += 1; return FiniteStateAutomaton(set(), {q}, set(), q, {q})
    elif type(re) == str:
        q = QC; QC += 1; r = QC; QC += 1
        return FiniteStateAutomaton({re}, {q, r}, {(q, re, r)}, q, {r})
    elif type(re) == Choice:
        A1, A2 = REToFSA(re.e1), REToFSA(re.e2)
        R2 = {(A1.q0 if q == A2.q0 else q, a, r) for (q, a, r) in A2.R} # A2.q0 renamed to A1.q0 in A2.R
        F2 = {A1.q0 if q == A2.q0 else q for q in A2.F} # A2.q0 renamed to A1.q0 in A2.F
        return FiniteStateAutomaton(A1.T | A2.T, A1.Q | A2.Q, A1.R | R2, A1.q0, A1.F | F2)
    elif type(re) == Conc:
        A1, A2 = REToFSA(re.e1), REToFSA(re.e2)
        R = A1.R | {(f, a, r) for (q, a, r) in A2.R if q == A2.q0 for f in A1.F} | \
            {(q, a, r) for (q, a, r) in A2.R if q != A2.q0}
        F = (A2.F - {A2.q0}) | (A1.F if A2.q0 in A2.F else set())
        return FiniteStateAutomaton(A1.T | A2.T, A1.Q | A2.Q, R, A1.q0, F)
    elif type(re) == Star:
        A = REToFSA(re.e)
        R = A.R | {(f, a, r) for (q, a, r) in A.R if q == A.q0 for f in A.F}
        return FiniteStateAutomaton(A.T, A.Q, R, A.q0, {A.q0} | A.F)
    else: raise Exception('not a regular expression')

def convertRegExToFSA(re) -> FiniteStateAutomaton:
    global QC; QC = 0
    return REToFSA(re)
```

For example, revisiting `E₁ = (ab)*|c`:

In [ ]:
```python
convertRegExToFSA(E1)
```

Question. What is a finite state automaton for `((a|b)c)*`? What is one for `ab|ac`? Analyze the results of applying `convertRegExToFSA`!

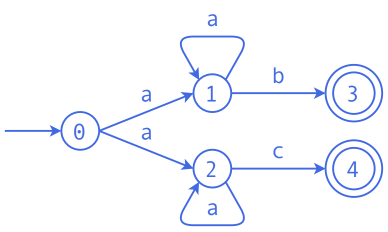
In [ ]:
```python
E2 = Star(Conc(Choice('a', 'b'), 'c')); A2 = convertRegExToFSA(E2); A2
```

In [ ]:
```python
E3 = Choice(Conc('a', 'b'), Conc('a', 'c')); A3 = convertRegExToFSA(E3); A3
```

Note that there are two possible transitions in state `0` on `a`.

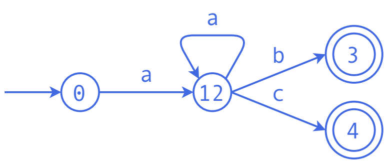
Nondeterministic to Deterministic Finite State Automaton

A finite state automaton `A` is *deterministic* if in every state for any input there is at most one transition which can be taken, otherwise it is *nondeterministic.* In the example to the right, if the automaton is in state `0` and next input symbol is `a`, either the transition to `1` or to `2` can be taken. If the whole input is `ac` and the transition to `1` was taken, the automaton would need to backtrack to state `0` and take the transition to `2` instead. If the input is `aⁿc`, the automaton would have to backtrack `n` transitions. As in general nondeterminism can arise with every state, backtracking may require in the order of `2ⁿ` paths to be explored for an input of length `n`, thus is undesirable.



Question. What is an equivalent deterministic finite state automaton? How is it constructed?

Answer. A deterministic finite state automaton can be constructed by unifying the states `1` and `2` of the original automaton.

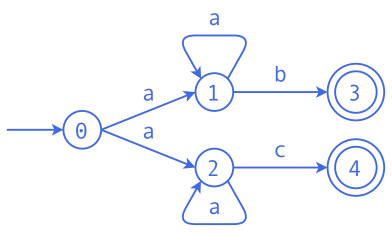


For finite state automaton `A = (T, Q, R, q₀, F)`, we can construct an equivalent, deterministic finite state automaton `A' = (T, Q', R', q'₀, F')` over the same symbols `T` by the *subset construction* algorithm, called so as the states `Q'` become sets of states of `Q`. The algorithm iteratively visits all states of `Q` and adds states to `Q'` by merging states of `Q` and adds transitions to `R'` by transforming those of `R`:

```
procedure deterministicFSA(T, Q, R, q₀, F) → (T, Q', R', q'₀, F')
    q'₀ := {q₀}
    Q', R', visited := {q'₀}, {}, {}
    while Q' ≠ visited do
        q' :∈ Q' − visited ; visited := visited ∪ {q'}
        for t ∈ T do
            r' := {r | q t → r ∈ R, q ∈ q'}
            if r' ≠ {} then Q', R' := Q' ∪ {r'}, R' ∪ {q' t → r'}
    F' := {q' | f ∈ q', f ∈ F}
```

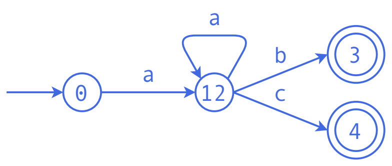
The nondeterministic assignment `x :∈ S` assigns to `x` an arbitrary element of set `S`.

Applied to the example, q'_0 becomes `{0}`. Variables `Q'`, `R'`, `visited` successively become:



Q'	R'	visited
`{{0}}`	`{}`	`{}`
`{{0}, {1, 2}}`	`{{0} a → {1, 2}}`	`{{0}}`
`{{0}, {1, 2}, {3}, {4}}`	`{{0} a → {1, 2}, {1, 2} a → {1, 2}, {1, 2} b → {3}, {1, 2} c → {4}}`	`{{0}, {1, 2}}`
`{{0}, {1, 2}, {3}, {4}}`	`{{0} a → {1, 2}, {1, 2} a → {1, 2}, {1, 2} b → {3}, {1, 2} c → {4}}`	`{{0}, {1, 2}, {3}}`
`{{0}, {1, 2}, {3}, {4}}`	`{{0} a → {1, 2}, {1, 2} a → {1, 2}, {1, 2} b → {3}, {1, 2} c → {4}}`	`{{0}, {1, 2}, {3}, {4}}`

In the second last step, either `{3}` or `{4}` can be visited first. Finally, `F'` becomes `{{3}, {4}}`.



Question. What are the steps in making the finite state automaton accepting `ab|ac` deterministic?

Answer. The finite state automaton is has `T = {a, b, c}`, `Q = {0, 1, 2, 3, 4}`, q_0 `= 0`, `F = {2, 4}` and has rules `R` consisting of `0 a → 1`, `0 a → 3`, `1 b → 2`, `3 c → 4` according to A_3 above. Then q'_0 becomes `{0}` and `Q'`, `R'`, `visited` successively become:

Q'	R'	visited
`{{0}}`	`{}`	`{}`
`{{0}, {1, 3}}`	`{{0} a → {1, 3}}`	`{{0}}`
`{{0}, {1, 3}, {2}, {4}}`	`{{0} a → {1, 3}, {1, 3} b → {2}, {1, 3} c → {4}}`	`{{0}, {1, 3}}`
`{{0}, {1, 3}, {2}, {4}}`	`{{0} a → {1, 3}, {1, 3} b → {2}, {1, 3} c → {4}}`	`{{0}, {1, 3}, {4}}`
`{{0}, {1, 3}, {2}, {4}}`	`{{0} a → {1, 3}, {1, 3} b → {2}, {1, 3} c → {4}}`	`{{0}, {1, 3}, {2}, {4}}`

In the second last step, either `{2}` or `{4}` would be visited first. Finally, `F'` becomes `{{2}, {4}}`.

In general, if the number of states of `A` is `n`, then `A'` will have at most 2^n states. The resulting deterministic finite state automaton does not require backtracking. That is, the subset construction trades exponential run-time (in the length of the input) for exponential memory (in the size of the original automaton).

The implementation of the subset construction in Python follows closely the algorithm. As Python does not allow sets of sets, `frozenset` is used for the new states (elements of sets have to have a hash method defined to allow fast checking of membership; the hash method is only defined for immutable data types like `str` and `frozenset` but sets are mutable). To make the states more readable, at the end all `frozenset` are converted to strings:

In [ ]:
```python
def deterministicFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    q′0 = set({fsa.q0})
    Q′, R′, visited = {q′0}, set(), set()
    # print(Q′, R′, visited)
    while visited != Q′:
        q′ = (Q′ - visited).pop(); visited |= {q′}
        for t in fsa.T:
            r′ = {r for (q, u, r) in fsa.R if u == t and q in q′}
            if r′ != set(): Q′ |= {set(r′)}; R′ |= {(q′, t, set(r′))}
        # print(Q′, R′, visited)
    F′ = {q′ for q′ in Q′ for f in fsa.F if f in q′}
    return FiniteStateAutomaton(fsa.T, Q′, R′, q′0, F′)
```

In [ ]:
```python
A0 = parseFSA("""
0
2
0 a → 1
1 a → 0
1 a → 2
"""); A0
```

In [ ]:
```python
A0det = deterministicFSA(A0); A0det
```

For example, continuing with E_3 `= ab|ac`:

In [ ]:
```python
E3 = Choice(Conc('a', 'b'), Conc('a', 'c'))
```

```
A3 = convertRegExToFSA(E3); A3
```

In [ ]:

```
A3det = deterministicFSA(A3); A3det
```

As another example, let us revisit `aa*b|aa*c`

In [ ]:

```
E4 = Choice(Conc(Conc('a', Star('a')), 'b'), Conc(Conc('a', Star('a')), 'c'))
A4 = convertRegExToFSA(E4); A4
```

In [ ]:

```
A4det = deterministicFSA(A4); A4det
```

Since `A` is deterministic, for every `q` and `u` there exists at most one `r` such that `(q, u, r) ∈ R`. Thus we can define a partial *next-state function* `δ` that, for given state and input in its domain, returns the unique next state:

```
    δ(q, u) = r  ≡  (q, u, r) ∈ R          (q, u) ∈ dom δ  ≡  (∃ r • (q, u, r) ∈ R)
```

A deterministic finite state automaton can be executed without backtracking. A simple way is by a *universal interpreter* that takes the finite state automaton and looks up successively the symbols of the input in `δ` to determine the next state from the current state:

In [ ]:

```
def accepts(fsa: FiniteStateAutomaton, τ: str) -> bool:
    δ = {(q, a): r for (q, a, r) in fsa.R}
    q = fsa.q0
    for t in τ:
        if (q, t) in δ: q = δ[q, t]
        else: return False
    return q in fsa.F
```

In [ ]:

```
accepts(A4det, 'aac')
```

With readability and efficiency in mind, the states of a (deterministic or nondeterministic) finite state automaton can *renamed* by numbering them `0`, `1`, etc. The Python implementation represents the names as strings. A mapping `m` is constructed that maps the original names to the new names:

In [ ]:

```
def renameFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    m, c = {}, 0
    for q in fsa.Q:
        m[q] = c; c = c + 1
    Q′ = {i for i in range(c)}
    R′ = {(m[q], u, m[r]) for (q, u, r) in fsa.R}
    q′0 = m[fsa.q0]
    F′ = {m[q] for q in fsa.F}
    return FiniteStateAutomaton(fsa.T, Q′, R′, q′0, F′)
```

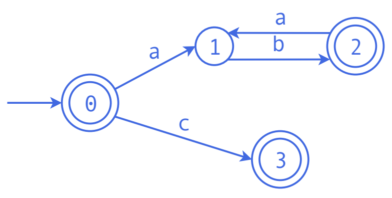
In [ ]:

```
A4simp = renameFSA(A4det); A4simp
```

Finite State Automaton to Regular Grammar

For a (deterministic or nondeterministic) finite state automaton `A`, we can construct an equivalent regular grammar `G` over the same symbols, that is `L(A) = L(G)`.

For example, for the automaton to the right a regular grammar is `G = (T, N, P, S)` with `T = {a, b, c}`, `N = {0, 1, 2, 3}`, `S = 0`, and `P` consisting of:

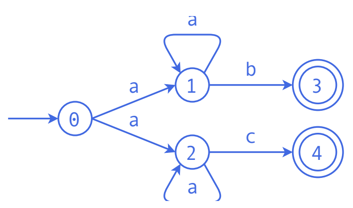


```
0 → a 1
    0 → c 3
    0 → ε
    1 → b 2
    2 → a 1
    2 → ε
    3 → ε
```

In general, a grammar `G = (T, N, P, S)` that is equivalent to `A = (T, Q, R, q₀, F)` is:

- `N = Q`
- `P = {q → t q' | q t → q' ∈ R} ∪ {q → ε | q ∈ F)`
- `S = q₀`

Question. What is an equivalent grammar for the automaton to the right?



Answer. An equivalent regular grammar is `G = (T, N, P, S)` with `T = {a, b}`, `N = {0, 1, 2, 3, 4}`, `S = 0`, and `P` consisting of:

```
0 → a 1
0 → a 2
1 → a 1
1 → b 3
2 → a 2
2 → c 4
3 → ε
4 → ε
```

To summarize,

- from every regular grammar an equivalent regular expression can be constructed,
- from every regular expression an equivalent finite state automaton can be constructed, and
- from every finite state automaton, an equivalent regular grammar can be constructed.

That is, all three define the same set of languages, the *regular languages*. They are of particular interest since *deterministic acceptors with finite states* can be constructed for them. These acceptors run in time linear to the length of the input.

For larger classes of languages, like the context-free languages, acceptors with finite states cannot be constructed in general.

Minimizing a Finite State Automaton

For a deterministic finite state automaton `A`, an equivalent *minimal* and deterministic finite state automaton `A′` over the same symbols can be constructed by "merging" equivalent states of `A`. The resulting automaton has the least possible number of states and is unique up to the naming of the states.

The algorithm partitions states into equivalent states, starting with the coarsest partitioning and refining it as needed. Initially, it is assumed that only the final and non-final states are distinct, i.e. there are only two equivalence classes. The algorithm uses the next-state function `δ`: two states, say `q`, `r` become distinct if on some input `u ∈ T`,

- `δ(q, u)` is defined and `δ(r, u)` not, or vice versa, or
- in case both `δ(q, u)` and `δ(r, u)` are defined, their values are states that were previously identified as distinct.

This is repeated until no two states can be further distinguished.

Consider A_4 to the right. A table is used to show which states are distinct. Initially, `2` and `4`, the final states, are distinct from all other states, indicated by ⓪. Going over all pairs of states, the first round distinguishes `0` from `1`, as there are transitions on `b` and `c` in `1` but not in `0`, and distinguishes `0` from `3`, for the same reason; this is indicated by ①. The next round does not distinguish any more states, so the process stops; `1` and `3` are equivalent and so are `2` and `4`:

Example.
A_4 `= (T, Q, R,` q_0`, F)` with `T = {a, b, c}`, `Q = {0, 1, 2, 3, 4}`, q_0 `= 0`, `F = {2, 4}`, and transitions `R`:

```
0 a → 1
 1 a → 3
 1 b → 4
 1 c → 2
 3 a → 3
 3 b → 4
 3 c → 2
```

	0	**1**	**2**	**3**	**4**
0		①	⓪	①	⓪
1	①		⓪		⓪
2	⓪	⓪		⓪	
3	①		⓪		⓪
4	⓪	⓪		⓪	

The new states `Q'` are `{0}`, `{1, 3}`, and `{2, 4}`. For each transitions of `R` there must be a new transition in `R′` that subsumes the source and target states of `R`; these would be:

```
{0} a → {3, 1}
{3, 1} a → {3, 1}
{3, 1} b → {2, 4}
{3, 1} c → {2, 4}
```

The new initial state q'_0 is that state that contains the old initial state q_0, here `{0}`. The new final states `F′` are those that contain a final state of `F`, here `{2, 4}`.

In the *minimization algorithm* the table is represented as a set of pairs, `dist`, which is modified only by adding pairs. New states `Q′` are sets of old states that according to `dist` are not distinct:

```
procedure minimizeFSA(T, Q, R, $q_0$, F) → (T, Q', R', $q'_0$, F')
    dist := {(q, r) | q, r ∈ Q, (q ∈ F) ≠ (r ∈ F)}
    done := false
    while ¬done do
        done := true
        for q, r ∈ Q do
```

```
                if q ≠ r ∧ (q, r) ∉ dist ∧
                    (∃ u ∈ T • ((q, u) ∈ dom δ ≠ (r, u) ∈ dom δ) ∨ ((q, u) ∈ dom δ ∧ (δ(q, u),
    δ(r, u)) ∈ dist)) then
                    dist := dist ∪ {(q, r)} ; done := false
        Q′ := {{q} ∪ {r | (q, r) ∉ dist} | q ∈ Q}
        R′ := {(q′, u, r′) | (q, u, r) ∈ R for some q ∈ q′, r ∈ r′}
        q′₀ :∈ Q′ such that q₀ ∈ q′₀
        F′ := {q′ ∈ Q' | q′ ∩ F ≠ {}}
```

The Python implementation closely follows the algorithm. A `dict` is used for `δ`:

In [ ]:
```python
def minimizeFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    δ = {(q, a): r for (q, a, r) in fsa.R}
    dist = {(q, r) for q in fsa.Q for r in fsa.Q if q != r and (q in fsa.F) != (r in fsa.F)}
    done = False
    while not done:
        done = True #; print(dist)
        for q in fsa.Q:
            for r in fsa.Q:
                if q != r and (q, r) not in dist and any(((q, u) in δ) != ((r, u) in δ) or \
                   ((q, u) in δ) and ((δ[(q, u)], δ[(r, u)]) in dist) for u in fsa.T):
                   dist |= {(q, r)}; done = False #; print('adding', q, r)
    Q′ = {set({q} | {r for r in fsa.Q if (q, r) not in dist}) for q in fsa.Q}
    R′ = {(q′, u, r′) for q′ in Q′ for r′ in Q′ for u in fsa.T if any((q, u, r) in fsa.R for q in q′ for r in r
    q′0 = {q′ for q′ in Q′ if fsa.q0 in q′}.pop()
    F′ = {q′ for q′ in Q′ if (q′ & fsa.F) != set()}
    return FiniteStateAutomaton(fsa.T, Q′, R′, q′0, F′)
```

Let us minimize `A₄`, the automaton accepting `aa*b|aa*c`, and then rename the states for readability:

In [ ]:
```python
A4simp
```

In [ ]:
```python
A4min = minimizeFSA(A4simp); A4min
```

In [ ]:
```python
A4opt = renameFSA(A4min); A4opt
```

Question. Consider $A_5 = (T, Q, R, q_0, F)$ with $T = \{a\}$, $Q = \{0, 1, 2, 3, 4, 5\}$, $q_0 = 0$, $F = \{1, 4\}$, and transitions as below. What is the accepted language? Minimize A_5!

```
0 a → 1
1 a → 2
2 a → 3
3 a → 4
4 a → 5
5 a → 0
```

Answer. The language are sequences over `a` of length 1, 4, 7, 10, ..., i.e. multiples of 3 plus 1. Minimization results in:

In [ ]:
```python
A5 = parseFSA("""
0
1 4
0 a → 1
1 a → 2
2 a → 3
3 a → 4
4 a → 5
5 a → 0
"""); minimizeFSA(A5)
```

Equivalence of Finite State Automata

The fact that minimal finite state automata are unique, up to the naming of states, can be used for checking their equivalence. For this, additionally their next-state function has to be *total*, i.e. there exists a transition for all `a ∈ T` in each state `q ∈ Q`. Every automaton can be made total by adding a *trap state*, say `t`, a non-final state with transitions to itself on all symbols. If in a state `q` there is no transition on `a`, the transition `q a → t` is added. As the vocabulary of the completed automaton we use all lower-case letter. A trap state is only added if the next-state function is not already total:

In [ ]:
```python
def totalFSA(A: FiniteStateAutomaton, t = -1) -> FiniteStateAutomaton:
    T = set('abcdefghijklmnopqrstuvwxyz') # T is vocabulary, t is trap state
    R = A.R | {(q, a, t) for q in A.Q for a in T if all((q, a, r) not in A.R for r in A.Q)}
    if any(r == t for (q, a, r) in R): # transition to t exists
        Q = A.Q | {t}
        R = R | {(t, a, t) for a in T}
    else: Q = A.Q
    return FiniteStateAutomaton(T, Q, R, A.q0, A.F)
```

Suppose $A = (T, Q, R, q_0, F)$ and $A' = (T, Q', R', q_0', F')$ over the same symbols `T` are given. The *equivalence* *checking algorithm* uses a bijection, called `m` below, that maps states of `Q` to states of `Q′`. It is successively constructed by visiting all states of `A` that are reachable from `q₀`. For each visited state `q` of `A` and corresponding state `m(q)` of `A′`, transitions on all symbols `u ∈ T` are considered:

- if `A` has a transition on `u` and `A′` not, or vice versa, `A` and `A′` are not equivalent;
- if `A` and `A′` have transitions on `u` to states that are not related by `m`, then `A` and `A′` are not equivalent; if the transitions lead to states that are not yet in the domain and range of `m`, respectively, then `m` is extended.

A set `v` of states that need to be visited is maintained; the notation `{a → b}` is used for a function that maps `a` to `b`:

```
procedure equivalentFSA(T, Q, R, q₀, F, Q′, R′, q₀′, F′): boolean
    m, v := {q₀ → q₀′}, {q₀}
    while v ≠ {} do
        q :∈ v; q′ := m(q)
        for u ∈ T do
            if (q, u) ∈ dom δ ≠ (q′, u) ∈ dom δ′ then return false
            else if (q, u) ∈ δ then
                r, r′ := δ(q, u), δ′(q, u)
                if r ∈ dom m then
                    if m(r) ≠ r′ then return false
                else if r′ ∈ ran m then return false
                else v := v ∪ {r} ; m(r) := r′
    return F′ = {m(q) | q ∈ F}
```

The last line checks that the final states are equivalent. The algorithm traverses all states of `A` in tree-like fashion; the use of a set for `v` leaves it open if the traversal is depth-first or breadth-first.

The Python implementation follows the algorithm closely; it checks that the symbols of both automata are the same and assumes that the automata are deterministic and minimal:

In [ ]:
```python
def equivalentFSA(a: FiniteStateAutomaton, a′: FiniteStateAutomaton, printMap = False) -> bool:
    a = minimizeFSA(totalFSA(a))
    a′ = minimizeFSA(totalFSA(a′))
    δ = {(q, u): r for (q, u, r) in a.R}
    δ′ = {(q, u): r for (q, u, r) in a′.R}
    m, v = {a.q0: a′.q0}, {a.q0}
    while v != set():
        q = v.pop(); q′ = m[q]
        for u in a.T:
            if ((q, u) in δ) != ((q′, u) in δ′): return False
            elif (q, u) in δ: # (q′, u) in δ′
                r, r′ = δ[(q, u)], δ′[(q′, u)]
                if r in m:
                    if m[r] != r′: return False
                elif r′ in m.values(): return False
                else: v.add(r); m[r] = r′
    if printMap: print(m)
    return a′.F == {m[q] for q in a.F}
```

We expect that `A₄` minimized is equivalent to the minimized and renamed version:

In [ ]:
```python
A4min
```

In [ ]:
```python
A4opt
```

In [ ]:
```python
equivalentFSA(A4min, A4opt, True) # modify to True
```

Checking the equivalence of two regular expression can be reduced to checking the equivalence of two finite state automata by first converting the regular expressions to finite state automata then making them deterministic. Following checks if `ac | bc` is the same as `(a | b) c`:

In [ ]:
```python
E7 = Choice(Conc('a', 'c'), Conc('b', 'c')) # ac | bc
A7 = deterministicFSA(convertRegExToFSA(E7)); A7
```

In [ ]:
```python
E8 = Conc(Choice('a', 'b'), 'c') # (a | b) c
A8 = deterministicFSA(convertRegExToFSA(E8)); A8
```

In [ ]:
```python
equivalentFSA(A7, A8)
```

Let us define `equalRegEx(e1, e2)` as follows:

In [ ]:
```python
def equalRegEx(e1, e2) -> bool:
    a1 = deterministicFSA(convertRegExToFSA(e1))
    a2 = deterministicFSA(convertRegExToFSA(e2))
    return equivalentFSA(a1, a2)
```

For example, `[a a*] = a*`:

```
In [ ]: equalRegEx(Choice(Conc('a', Star('a')), ''), Star('a'))
```

Question. How can you simplify `(a|b*)*`? Make a guess and check it!

Answer. This can be simplified to `(a|b)*`:

```
In [ ]: equalRegEx(Star(Choice('a', Star('b'))), Star(Choice('a', 'b')))
```

Historic Notes and Further Reading

Nondeterminism arises in programming languages with concurrency, abstracting from the specifics of scheduling on a single or multiple processors (e.g. "Threads can write the variable in any order"). Nondeterminism also arises as abstractions in sequential programs (e.g. "An arbitrary element of the set is removed"). This is expressed through guarded commands or, more generally, through nondeterministic assignments like `x :∈ S`. This kind of nondeterminism is called *demonic* as the programmer has be prepared for an arbitrary choice. By contrast, the nondeterminism of finite state automata is called *angelic* as it will accept an input whenever some choice between nondeterministic transitions exists. Demonic nondeterminism can be implemented by making an arbitrary choice, angelic nondeterminism only through backtracking. Finite state automata are sometimes called *finite state machines*; we use the term finite state automata to distinguishes them from finite state machines with demonic nondeterminism.

Bibliography