Modal Symbolic Learning: Day 1

Propositional Logic

Formulas & Interpretations

In [1]:

```
using Pkg
Pkg.activate(".")
Pkg.instantiate()
Pkg.update()
Pkg.status()
```

```
  Activating project at `~/Desktop/modal-symbolic-learning-course`
    Updating registry at `~/.julia/registries/General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
   Installed FillArrays ─ v1.9.0
  No Changes to `~/Desktop/modal-symbolic-learning-course/Project.toml`
    Updating `~/Desktop/modal-symbolic-learning-course/Manifest.toml`
  [1a297f60] ↑ FillArrays v1.8.0 ⇒ v1.9.0
Precompiling project...
  ✓ FillArrays
  ✓ FillArrays → FillArraysPDMatsExt
  ✓ FillArrays → FillArraysSparseArraysExt
  ✓ FillArrays → FillArraysStatisticsExt
  ✓ Distributions
  ✓ Distributions → DistributionsTestExt
  ✓ ScientificTypes
  ✓ CategoricalDistributions
  ✓ SoleData
  ✓ MLJModels
  ✓ MLJBase
  ✓ MLJEnsembles
  ✓ MLJFlow
  ✓ MLJTuning
  ✓ MLJIteration
  ✓ MLJ
  ✓ SoleModels
  ✓ Sole
  ✓ ModalDecisionTrees
  19 dependencies successfully precompiled in 47 seconds. 270 already precom
piled.
```

```
Status `~/Desktop/modal-symbolic-learning-course/Project.toml`
  [a93c6f00] DataFrames v1.6.1
  [7806a523] DecisionTree v0.12.4
  [7073ff75] IJulia v1.24.2
⌅ [add582a8] MLJ v0.19.5
  [c6f25543] MLJDecisionTreeInterface v0.4.0
  [e54bda2e] ModalDecisionTrees v0.3.1
  [91a5bcdd] Plots v1.39.0
  [7b3b3b3f] Sole v0.3.1
  [b002da8f] SoleLogics v0.6.10
  [4249d9c7] SoleModels v0.5.2
  [2913bbd2] StatsBase v0.34.2
  [9a3f8284] Random
Info Packages marked with ⌅ have new versions available but compatibility co
nstraints restrict them from upgrading. To see why use `status --outdated`
```

In [ ]:
```julia
using SoleLogics
```

In [2]:
```julia
p = Atom("it's alive")
```

Out[2]:
```
Atom{String}: it's alive
```

In [3]:
```julia
q = Atom("it's mortal!")
```

Out[3]:
```
Atom{String}: it's mortal!
```

In [4]:
```julia
φ = p ∧ q
```

Out[4]:
```
SyntaxBranch{NamedConnective{:∧}}: it's alive ∧ it's mortal!
```

In [5]:
```julia
φ isa Formula && p isa Formula
```

Out[5]:
```
true
```

In [6]:
```julia
# φ is the root node of a syntax tree.
# Print the syntax token at the root node
println(token(φ))
# Print the children of the root node
println(children(φ))
```

```
∧
(Atom{String}: it's alive, Atom{String}: it's mortal!)
```

In [7]:
```julia
# Create a method for negating any formula
function negateformula(f::Formula)
    return ¬f
end
```

Out[7]:
```
negateformula (generic function with 1 method)
```

In [8]:
```julia
negateformula(φ)
```

Out[8]:
```
SyntaxBranch{NamedConnective{:¬}}: ¬(it's alive ∧ it's mortal!)
```

In [9]:
```julia
negateformula(p)
```

Out[9]: SyntaxBranch{NamedConnective{:¬}}: ¬it's alive

In [10]:
```julia
# Obtain the string representation of a Formula
syntaxstring(φ)
```

Out[10]: "it's alive ∧ it's mortal!"

In [11]:
```julia
# I can also parse Formula's from standard string representations
φ = parseformula("it's alive ∧ it's mortal!")
```

Out[11]: SyntaxBranch{NamedConnective{:∧}}: it's alive ∧ it's mortal!

In [12]:
```julia
function my_own_string_representation(f::SyntaxBranch{NamedConnective{:¬}})
    return "It is not the case that $(my_own_string_representation(first(chi
end
function my_own_string_representation(f::SyntaxBranch{NamedConnective{:∧}})
    subformula1, subformula2 = children(f)
    return "both $(my_own_string_representation(subformula1)) and $(my_own_s
end
function my_own_string_representation(f::SyntaxLeaf)
    return syntaxstring(f)
end
```

Out[12]: my_own_string_representation (generic function with 3 methods)

In [13]:
```julia
my_own_string_representation(¬φ)
```

Out[13]: "It is not the case that both it's alive and it's mortal!"

In [14]:
```julia
Guido = TruthDict([p => true, q => false])
```

Out[14]: TruthDict with values:

it's mortal! String	**it's alive** String
⊥	⊤

In [15]:
```julia
Giovanni = TruthDict([p => true, q => true])
```

Out[15]: TruthDict with values:

it's mortal! String	**it's alive** String
⊤	⊤

In [16]:
```julia
check(p, Guido) # Guido is alive
```

Out[16]: true

```julia
In [17]: check(φ, Guido) # But not both alive and mortal
```

Out[17]: false

```julia
In [18]: check(φ, Giovanni) # Giovanni is both alive and mortal!
```

Out[18]: true

```julia
In [19]: # These objects can actually be used as dictionaries from Formula to Truth v
         # For example, we both assign ⊤ (top) to the atom "alive"
         Guido[p], Giovanni[p]
```

Out[19]: (⊤, ⊤)

```julia
In [20]: # What is ⊤?
         ⊤ isa Truth      &&     ⊤ isa Formula
```

Out[20]: true

```julia
In [21]: # Now it's time to say it: these Unicode character are
         #  not too comfy for a Jupyter Notebook. Let's use their aliases.
         TOP, BOT, CONJUNCTION, DISJUNCTION, IMPLICATION
         #  (but try typing \top<tab> in the Julia REPL)
```

Out[21]: (⊤, ⊥, ∧, ∨, →)

```julia
In [22]: # Actually, indexing (with []) can be used to check generic Formula's
         Guido[φ]
```

Out[22]: ⊥

```julia
In [23]: # This is syntactic sugar for the *interpretation* algorithm, which is actua
         interpret(φ, Guido)
```

Out[23]: ⊥

```julia
In [24]: # In fact, it also works under incomplete information.
         # Notice how in this example, with an *unknown atom*, it uses the *known* in
         Giovanni[φ ∨ Atom("?Unknown property?")]
```

Out[24]: ⊤

```julia
In [25]: # So ultimately, `check` is just a shortcut for making sure that `interpret`
         check(φ, Guido) == istop(interpret(φ, Guido))
```

Out[25]: true

```julia
In [26]: # Let's generate random formulas
         treeheight = 3
```

```
@atoms a b
φ2 = randformula(treeheight, [a,b], SoleLogics.BASE_PROPOSITIONAL_CONNECTIVE
```

Out[26]: `SyntaxBranch{NamedConnective{:→}}: (¬b ∧ a ∧ a) → (b ∨ a ∨ (b → b))`

In [27]:
```
# Let's control reproducibility, though ;)
using Random
rng = Random.MersenneTwister(1)
φ2 = randformula(rng, treeheight, [a,b], SoleLogics.BASE_PROPOSITIONAL_CONNE
```

Out[27]: `SyntaxBranch{NamedConnective{:¬}}: ¬(b ∧ a ∧ ¬b)`

In [28]:
```
normalize(φ2)
```

Out[28]: `SyntaxBranch{NamedConnective{:∨}}: ¬a ∨ b ∨ ¬b`

HOMEWORK EXERCISE 1

Check many, randomly-generated formulas on the alphabet p, q on both `Guido` and `Giovanni`. Do `Guido` and `Giovanni` have the same probability of satisfying a generic formula? Can you estimate this probability?

Scalar logisets

Now, let's consider a propositional interpretation on scalar variables, and check formulas on an alphabet $\mathcal{A} \subseteq \{V < v, v \in \mathbb{R}\}$ on it.

In [29]:
```
import SoleLogics: syntaxstring

struct ConditionOnVariable
    i_variable::Integer
    threshold::Number
end

function syntaxstring(c::ConditionOnVariable; kwargs...)
    "V$(c.i_variable) < $(c.threshold)"
end

syntaxstring(ConditionOnVariable(2, 10))
```

Out[29]: `"V2 < 10"`

In [30]:
```
using SoleLogics: AbstractAssignment # Propositional Interpretations

struct TabularInterpretation{T<:Real} <: AbstractAssignment
    vals::Vector{T}
end

import SoleLogics: interpret

function interpret(a::Atom{ConditionOnVariable}, I::TabularInterpretation)
    cond = value(a)
```

```julia
    return (I.vals[cond.i_variable] < cond.threshold ? TOP : BOT)
end
```

Out[30]:
```
interpret (generic function with 12 methods)
```

In [31]:
```julia
rng = Random.MersenneTwister(1)
n_variables = 4

I = TabularInterpretation(rand(rng, n_variables))
```

Out[31]:
```
TabularInterpretation{Float64}([0.23603334566204692, 0.34651701419196046,
0.3127069683360675, 0.00790928339056074])
```

In [32]:
```julia
A = Atom.([ConditionOnVariable(v, t) for v in 1:n_variables for t in 0:0.1:1
syntaxstring.(A)
```

Out[32]:
```
44-element Vector{String}:
 "V1 < 0.0"
 "V1 < 0.1"
 "V1 < 0.2"
 "V1 < 0.3"
 "V1 < 0.4"
 "V1 < 0.5"
 "V1 < 0.6"
 "V1 < 0.7"
 "V1 < 0.8"
 "V1 < 0.9"
 "V1 < 1.0"
 "V2 < 0.0"
 "V2 < 0.1"
 ⋮
 "V3 < 1.0"
 "V4 < 0.0"
 "V4 < 0.1"
 "V4 < 0.2"
 "V4 < 0.3"
 "V4 < 0.4"
 "V4 < 0.5"
 "V4 < 0.6"
 "V4 < 0.7"
 "V4 < 0.8"
 "V4 < 0.9"
 "V4 < 1.0"
```

In [33]:
```julia
[interpret(cond, I) for cond in A]
```

Out[33]:
```
44-element Vector{BooleanTruth}:
 ⊥
 ⊥
 ⊥
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊥
 ⊥
 ⋮
 ⊤
 ⊥
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
 ⊤
```

In [34]:
```
rng = Random.MersenneTwister(32)

[begin
f = randformula(rng, 3, A, SoleLogics.BASE_PROPOSITIONAL_CONNECTIVES)
syntaxstring(f) => interpret(f, I)
end for _ in 1:10]
```

Out[34]:
```
10-element Vector{Pair{String}}:
                 "(¬V2 < 0.4 ∧ (V1 < 0.5 → V4 < 0.8)) → (V4 < 0.4 ∧ V4 < 0.
9 ∧ V2 < 0.9 ∧ V1 < 0.2)" => ⊤
                     "V4 < 0.1 ∨ V2 < 0.6 ∨ V4 < 0.8 ∨ V2 < 0.5 ∨ ¬V4 < 0.8
∧ (V2 < 0.4 → V1 < 0.5)" => ⊤
                                                    "¬(V3 < 0.6 ∧ V4 < 0.9)
∧ ¬(V2 < 0.3 → V1 < 1.0)" => ⊥
                                                 "¬¬V4 < 0.6 ∧ ((¬V1 < 0.0)
→ (V2 < 0.4 ∧ V2 < 0.3))" => ⊥
   "((V4 < 0.4 → V4 < 1.0) ∨ V1 < 0.7 ∨ V2 < 0.7) → ((V3 < 0.3 → V4 < 0.3)
∧ (V2 < 0.2 ∨ V3 < 0.5))" => ⊤
 "((V4 < 0.0 → V2 < 0.3) → (V4 < 0.2 ∧ V3 < 0.9)) ∨ ((V1 < 0.8 → V1 < 0.4)
→ (V2 < 0.7 → V2 < 0.0))" => ⊤
                           "¬V3 < 0.6 ∧ ¬V3 < 1.0 ∧ ((V3 < 0.5 → V3 < 0.5)
→ (V3 < 0.8 ∧ V4 < 0.3))" => ⊥
                                                                 "¬(V3 < 0.6
∧ V1 < 0.1 ∧ ¬V1 < 0.7)" => ⊤
                                                    "¬(V1 < 0.0 ∨ V3 < 0.5)
∧ ¬(V4 < 0.2 ∨ V1 < 0.8)" => ⊥

"¬(¬V1 < 0.5 ∨ ¬V3 < 0.8)" => ⊤
```

HOMEWORK EXERCISE 2

Check many, randomly-generated formulas on many, randomly-generated tabular interpretations, and store the formulas that satisfy the highest number of instances!

Modal Logic

In [35]:
```julia
# Instantiate a frame with 5 worlds and 5 edges
using Graphs

worlds = SoleLogics.World.(1:5)
edges = Edge.([(1,2), (1,3), (2,4), (3,4), (3,5)])
fr = SoleLogics.ExplicitCrispUniModalFrame(worlds, Graphs.SimpleDiGraph(edge
```

Out[35]:
```
SoleLogics.ExplicitCrispUniModalFrame{SoleLogics.World{Int64}, SimpleDiGrap
h{Int64}} with
- worlds = ["1", "2", "3", "4", "5"]
- accessibles =
        1 -> [2, 3]
        2 -> [4]
        3 -> [4, 5]
        4 -> []
        5 -> []
```

In [36]:
```julia
# Pick the first world
w1 = worlds[1]

# Enumerate the world that are accessible from the first world
accessibles(fr, w1)
```

Out[36]:
```
2-element Vector{SoleLogics.World{Int64}}:
 SoleLogics.World{Int64}(2)
 SoleLogics.World{Int64}(3)
```

In [37]:
```julia
# That's an iterator of worlds... If I want to see them, I'll collect them.
collect(accessibles(fr, w1))
```

Out[37]:
```
2-element Vector{SoleLogics.World{Int64}}:
 SoleLogics.World{Int64}(2)
 SoleLogics.World{Int64}(3)
```

In [38]:
```julia
# Assign each world a propositional interpretation
valuation = Dict([
        worlds[1] => TruthDict([p => TOP, q => BOT]),
        worlds[2] => TruthDict([p => TOP, q => TOP]),
        worlds[3] => TruthDict([p => TOP, q => BOT]),
        worlds[4] => TruthDict([p => BOT, q => BOT]),
        worlds[5] => TruthDict([p => BOT, q => TOP]),
     ])

# Instantiate a Kripke structure
K = KripkeStructure(fr, valuation)
```

Out[38]:
```
KripkeStructure{SoleLogics.ExplicitCrispUniModalFrame{SoleLogics.World{Int6
4}, SimpleDiGraph{Int64}}, Dict{SoleLogics.World{Int64}, TruthDict{Dict{Ato
m{String}, BooleanTruth}}}} with
- frame = SoleLogics.ExplicitCrispUniModalFrame{SoleLogics.World{Int64}, Si
mpleDiGraph{Int64}} with
- worlds = ["1", "2", "3", "4", "5"]
- accessibles =
        1 -> [2, 3]
        2 -> [4]
        3 -> [4, 5]
        4 -> []
        5 -> []
- valuations =
        1 -> TruthDict([it's mortal! => ⊥, it's alive => ⊤])
        2 -> TruthDict([it's mortal! => ⊤, it's alive => ⊤])
        3 -> TruthDict([it's mortal! => ⊥, it's alive => ⊤])
        4 -> TruthDict([it's mortal! => ⊥, it's alive => ⊥])
        5 -> TruthDict([it's mortal! => ⊤, it's alive => ⊥])
```

In [39]:
```julia
# Generate a random modal formula
φmodal = randformula(Random.MersenneTwister(14), 3, [p,q], SoleLogics.BASE_M

println(syntaxstring(φmodal))

# Check the formula on each world of the Kripke structure
[w => check(φmodal, K, w) for w in worlds]
```

```
□(it's alive → it's mortal!) ∨ ¬(it's alive → it's alive)
```

```
Out[39]: 5-element Vector{Pair{SoleLogics.World{Int64}, Bool}}:
 SoleLogics.World{Int64}(1) => 0
 SoleLogics.World{Int64}(2) => 1
 SoleLogics.World{Int64}(3) => 1
 SoleLogics.World{Int64}(4) => 1
 SoleLogics.World{Int64}(5) => 1
```

EXERCISE 3

Define a structure for representing a *modal* interpretation on scalar variables.

```
In [40]:
using SoleLogics: AbstractFrame, World, AbstractKripkeStructure

# TODO:
# struct ModalInterpretation{FR<:AbstractFrame,T<:Real} <: AbstractKripkeStr
#     frame::FR
#     vals::???
# end

import SoleLogics: interpret, frame

# Retrieve the interpretation's frame
frame(i::ModalInterpretation) = i.frame

# TODO:
# function interpret(a::Atom{ConditionOnVariable}, I::ModalInterpretation, w
#     cond = value(a)
#     v = ???
#     return (v < cond.threshold ? TOP : BOT)
# end
```

```
UndefVarError: `ModalInterpretation` not defined

Stacktrace:
 [1] top-level scope
   @ In[40]:12
```

```
In [41]:
rng = Random.MersenneTwister(1)
n_variables = 4
n_worlds = 5
n_edges = 7
n_formulas = 10

worlds = SoleLogics.World.(1:n_worlds)
g = SimpleDiGraph(n_worlds, n_edges; rng)
fr = SoleLogics.ExplicitCrispUniModalFrame(worlds, g)
variable_values = [rand(n_variables) for w in worlds]

Imodal = ModalInterpretation(fr, variable_values)

for i_formula in 1:n_formulas
    φmodal = randformula(Random.MersenneTwister(i_formula), 2, A, SoleLogics

    println(syntaxstring(φmodal) => ["w$(SoleLogics.name(w))" => check(φmoda
```

```julia
    println()
end
```

```
UndefVarError: `ModalInterpretation` not defined

Stacktrace:
 [1] top-level scope
   @ In[41]:12
```

EXERCISE 4

Check many, randomly-generated *modal* formulas on many, randomly-generated *modal* interpretations, and store the formulas that satisfy the highest number of instances!

In [42]:

```julia
# TODO
```