

3_PropositionalLogic

October 25, 2022

1 Propositional Logic

1.1 Syntax and Semantics Explained using a Prolog Implementation

1.2 Well-Formed Formulas (WFF)

All atomic propositions are WFF. If a and b are WFF then so are: $\neg a$ - $(a \wedge b)$ - $(a \vee b)$ - $(a \rightarrow b)$ - $(a \leftrightarrow b)$. No other formulas are WFF.

Note in the slides we use the single arrow \rightarrow instead of the double arrow \leftrightarrow for implication which is maybe more standard.

Comment: a , b are metavariables outside the syntax of propositional logic.

Maybe this reminds you of formal grammars from a theoretical computer science lecture. And indeed, the above can be written as a grammar using a non-terminal symbol wff.

In Prolog, grammars can actually be written using DCG notation, which we will see and understand much later in the course. Here we simply write the grammar in Prolog style and can then use it to check if a formula is a WFF:

```
[1]: :- set_prolog_flag(double_quotes, codes).
wff --> "p". % atomic proposition
wff --> "q". % atomic proposition
wff --> "(\neg,wff,)" .
wff --> "(,wff, ",wff, )" .
wff --> "(,wff, ",wff, )" .
wff --> "(,wff, "\rightarrow",wff, )" .
wff --> "(,wff, ",wff, )" .
```

```
[2]: ?- wff("p", "").
```

true

```
[3]: ?- wff("(\neg p)", "").
```

true

```
[4]: ?- wff("(p (q (\neg p)))", "").
%comment.
```

true

This grammar does not talk about whitespace and requires parentheses for every connective used. In practice, one typically uses operator precedences to avoid having to write too many parentheses: - negation binds strongest - then come conjunction and disjunction - then come implication and equivalence.

So instead of $((\neg p) (\neg q)) \rightarrow ((\neg p) (\neg q))$ we can write $\neg p \neg q \rightarrow \neg p \neg q$

We will try not to mix conjunction/disjunction and implication/equivalence without parentheses. We will also not try to improve our Prolog parser here to accomodate these precedences. This is the subject of another course (on compiler construction).

We will also see much later that Prolog allows one to annotate grammars with semantic values, here to generate a Prolog term representing the logical formal in the form of a syntax tree:

```
[5]: :- set_prolog_flag(double_quotes, codes).
wff(p) --> "p". % atomic proposition
wff(q) --> "q". % atomic proposition
wff(not(A)) --> "(\neg,wff(A),)".
wff(and(A,B)) --> "(", wff(A), " ", wff(B), ")".
wff(or(A,B)) --> "(", wff(A), " ", wff(B), ")".
wff(impl(A,B)) --> "(", wff(A), "\rightarrow", wff(B), ")".
wff(equiv(A,B)) --> "(", wff(A), " ", wff(B), ")".

parse_wff(String,Formula) :- wff(Formula,String,"").
```

```
[6]: ?- parse_wff("(\neg p)",Formula).
```

Formula = not(p)

```
[7]: ?- parse_wff("(\neg(p q))",Formula).
```

Formula = not(and(p,q))

```
[8]: ?- parse_wff("(p (q (\neg p)))",Formula).
```

Formula = and(p,or(q,not(p)))

The above Prolog term `and(p,or(q,not(p)))` represents the logical formula in tree form. We can display the tree as a dag (directed acyclic graph) using the following subsidiary code (in future the Jupyter kernel will probably have a dedicated command to show a term graphically):

```
[9]: subtree(Term,Nr,SubTerm) :-
    Term =.. [_|ArgList], %obtain arguments of the term
    nth1(Nr,ArgList,SubTerm). % get sub-argument and its position number

% recursive and transitive closure of subtree
rec_subtree(Term,Sub) :- Term = Sub.
rec_subtree(Term,Sub) :- subtree(Term,_,X), rec_subtree(X,Sub).
```

```

subnode(Sub,[shape/S, label/F],Formula) :-
    rec_subtree(Formula,Sub), % any sub-formula Sub of Formula is a node in the
    ↪graphical rendering
    functor(Sub,F,_), (atom(Sub) -> S=egg ; number(Sub) -> S=oval ; S=rect).

```

```

[10]: jupyter:print_table(subnode(Node,Dot,and(p,or(q,not(p))))

```

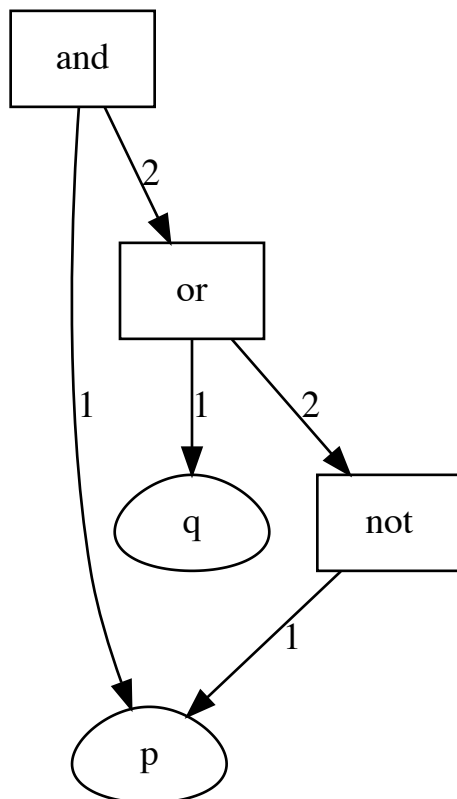
Node	Dot
and(p,or(q,not(p)))	[shape/rect,label/and]
p	[shape/egg,label/p]
or(q,not(p))	[shape/rect,label/or]
q	[shape/egg,label/q]
not(p)	[shape/rect,label/not]
p	[shape/egg,label/p]

true

```

[11]: jupyter:show_graph(subnode(_,_,and(p,or(q,not(p)))),subtree/3)

```



true

Below we will study how one can assign a truth-value to logical formulas like the one above.

1.3 Truth tables

Every logical connective has a truth table, indicating how it maps the truth of its arguments to its own truth value. For example, **and** maps inputs **true** and **false** to **false**.

Below we encode the truth table for five connectives as Prolog facts and rules.

```
[12]: not(true,false).
      not(false,true).

      and(true,V,V) :- truth_value(V).
      and(false,V,false) :- truth_value(V).

      or(true,V,true) :- truth_value(V).
      or(false,V,V) :- truth_value(V).

      implies(A,B,Res) :- not(A,NotA), or(NotA,B,Res).
      equiv(A,B,Res) :- implies(A,B,AiB), implies(B,A,BiA), and(AiB,BiA,Res).

      truth_value(true).
      truth_value(false).

      truth_table(A,B,NotA,AandB,AorB,AiB,AeB) :-
        truth_value(A), truth_value(B),
        not(A,NotA), and(A,B,AandB), or(A,B,AorB),
        implies(A,B,AiB), equiv(A,B,AeB).
```

We can now use the predicates **and/3**, ... to compute the truth value of the connectives for a particular input:

```
[13]: ?- and(true,false,X).
```

X = false

```
[14]: ?- append([a],[b],R), append(R,[c],R2).
```

R = [a,b],

R2 = [a,b,c]

```
[15]: ?- and(X,true,true).
```

X = true

```
[16]: ?- or(true,false,X).
```

X = true

We can also display the whole truth table using a Jupyter command:

```
[17]: jupyter:print_table(truth_table(A,B,NotA,AandB,AorB,AimpliesB,AequivB))
```

A	B	NotA	AandB	AorB	AimpliesB	AequivB
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

true

An interpretation is an assignment of all relevant proposition to truth values (true or false). For example, given the formula $(p \rightarrow (q \rightarrow \neg p))$ an interpretation needs to assign a truth value to p and q . In the Prolog code below we will represent an interpretation as a list of bindings, each binding is a term of the form `Proposition/TruthValue`. For example, one of the four possible interpretations for the formula above is `[p/true, q/false]`.

Using the built-in `member/2` predicate we can inspect an interpretation in Prolog. For example, here we retrieve the truth value of q :

```
[18]: ?- Interpretation=[p/true, q/false], member(p/P,Interpetation).
```

`Interpetation = [p/true,q/false],`

`P = true`

```
[19]: ?- Interpretation=[bind(p,true), bind(q,false)],
      ↪member(bind(q,Q),Interpetation).
```

`Interpetation = [bind(p,true),bind(q,false)],`

`Q = false`

Given an interpretation we can now compute the truth value for an entire formula in a recursive fashion. For propositions we look up the truth value in the interpretation (done using `member` below). For logical connectives we recursively compute the truth value of its arguments and then apply the truth tables we defined above.

```
[20]: value(X,Interpretation,Value) :-
      atomic(X), % we have a proposition
      member(X/Value,Interpretation).
value(and(A,B),I,Val) :-
      value(A,I,VA), value(B,I,VB),
      and(VA,VB,Val).
value(or(A,B),I,Val) :- value(A,I,VA), value(B,I,VB),
      or(VA,VB,Val).
value(not(A),I,Val) :- value(A,I,VA),
      not(VA,Val).
value(implies(A,B),I,Val) :- value(or(not(A),B),I,Val).
```

```
value(equiv(A,B),I,Val) :-
    value(and(implies(A,B),implies(B,A)),I,Val).
```

Using the above we can compute the truth value of $(p \vee (q \wedge \neg p))$ for the interpretation $[p/\text{true}, q/\text{false}]$:

```
[21]: ?- value(and(p,or(q,not(p))), [p/true, q/false],Res).
```

```
Res = false
```

We can also use our parser instead of writing the logical formulas as Prolog terms:

```
[22]: ?- parse_wff("(p (q (¬p)))",Formula), value(Formula, [p/true,q/false],Res).
```

```
Formula = and(p,or(q,not(p))),
```

```
Res = false
```

The truth value of $(p \vee (q \wedge \neg p))$ for the interpretation $[p/\text{true}, q/\text{true}]$ is true. In this case the interpretation is called a model of the formula.

```
[23]: ?- value(and(p,or(q,not(p))), [p/true, q/true],Res).
```

```
Res = true
```

We can also compute the truth table of the entire formula, by trying out all possible interpretations. Below we leave P and Q as Prolog variable which is instantiated by the code above:

```
[24]: jupyter:print_table(value(and(p,or(q,not(p))), [p/P, q/Q],Res))
```

P	Q	Res
true	true	true
true	false	false
false	true	false
false	false	false

```
true
```

Below we visualise graphically this computation, by displaying the syntax tree of a logical formula as a dag and by colouring the nodes using the `value` predicate:

```
[25]: subnode_val(Sub,[shape/S, label/F, style/filled, fillcolor/
    ↪C],Formula,Interpretation) :-
    rec_subtree(Formula,Sub), % any sub-formula Sub of Formula is a node in the
    ↪graphical rendering
    get_label(Sub,F), (atom(Sub) -> S=egg ; number(Sub) -> S=oval ; S=rect),
    (value(Sub,Interpretation,true) -> C=olive ; C=sienna1).

get_label(and(_,_),' ') :- !.
get_label(or(_,_),' ') :- !.
```

```

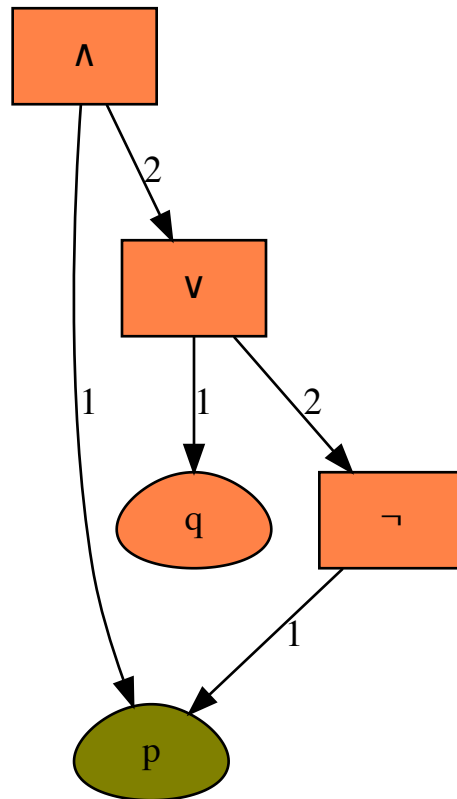
get_label(impl(_,_),' $\rightarrow$ ') :- !.
get_label(equiv(_,_),' $\leftrightarrow$ ') :- !.
get_label(not(_),' $\neg$ ') :- !.
get_label(Sub,F) :- functor(Sub,F,_).

```

```

[26]: jupyter:show_graph(subnode_val(_,_ ,and(p,or(q,not(p))),[p/true,q/
 $\rightarrow$ false]),subtree/3)

```

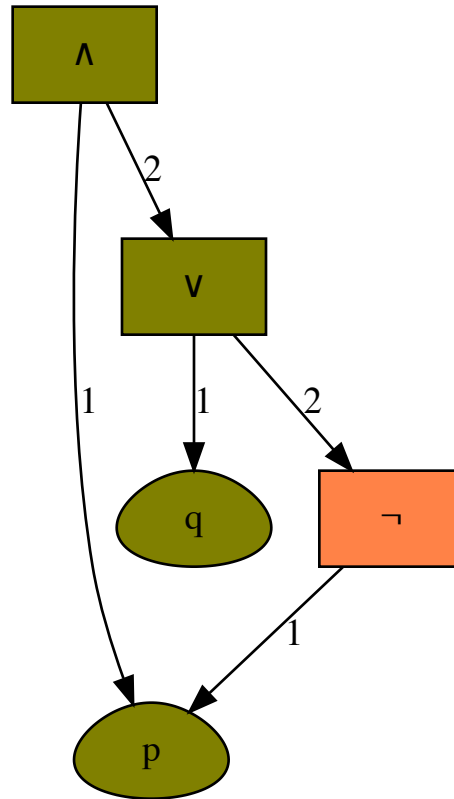


true

```

[27]: jupyter:show_graph(subnode_val(_,_ ,and(p,or(q,not(p))),[p/true,q/true]),subtree/
 $\rightarrow$ 3)

```



true

A formula for which all interpretations are models is called a tautology:

```
[28]: jupyter:print_table(value(or(p,not(p)), [p/P],Res))
```

P	Res
true	true
false	true

true

```
[29]: jupyter:print_table(value(or(p,or(q,not(q))), [p/P, q/Q],Res))
```

P	Q	Res
true	true	true
false	true	true
true	false	true
false	false	true

true

A formula which has no models is called a contradiction:

```
[30]: jupyter:print_table(value(and(p,not(p)), [p/P,Res]))
```

P	Res
true	false
false	false

true

A formula which has at least one model is called satisfiable.

We can use our code above to find models, by leaving the interpretation of propositions as Prolog variables and by requiring the result to be **true**:

```
[31]: ?-value(and(p,or(q,not(p))), [p/P, q/Q],true).
```

P = true,

Q = true

The above is not a very efficient way of finding models. We return to this issue later. A tool that determines whether a propositional logic formula is satisfiable and computes possible models is called a SAT solver.

Two formulas are called equivalent iff they have the same models. We write this as $A \equiv B$.

Below we can see that the models of $\neg p \vee q$ and $p \rightarrow q$ are identical, i.e., $\neg p \vee q \equiv p \rightarrow q$.

```
[32]: jupyter:print_table((user:value(or(not(p),q), [p/P, q/Q],NotPorQ),user:
    ↪value(implies(p,q), [p/P, q/Q],PimpliesQ)))
```

P	Q	NotPorQ	PimpliesQ
true	true	true	true
true	false	false	false
false	true	true	true
false	false	true	true

true

```
[33]: jupyter:print_table((user:value(and(p,q), [p/P, q/Q],PQ),user:value(and(q,p),
    ↪[p/P, q/Q],QP)))
```

P	Q	PQ	QP
true	true	true	true
true	false	false	false
false	true	false	false
false	false	false	false

true

A formula B is the logical consequence of A if all models of A are also models of B. We write this as $A \models B$.

Below we can see that all models of $p \vee q$ are also models of $p \wedge q$, i.e., $p \vee q \models p \wedge q$.

```
[34]: jupyter:print_table((user:value(and(p,q), [p/P, q/Q],PandQ),user:value(or(p,q),[p/P, q/Q],PorQ)))
```

P	Q	PandQ	PorQ
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

true

1.4 Improving our Prolog program

For convenience we now write some piece of Prolog code to find the atomic propositions used within a formula. This obviates the need to write interpretation skeletons like $[p/P, q/Q]$ ourselves.

```
[35]: get_aps(Formula,SortedPropositions) :-  
      findall(P, ap(Formula,P), Ps), sort(Ps,SortedPropositions).  
% extract atomic propositions used by formula  
ap(X,X) :- atomic(X).  
ap(and(A,B),AP) :- ap(A,AP) ; ap(B,AP).  
ap(or(A,B),AP) :- ap(A,AP) ; ap(B,AP).  
ap(implies(A,B),AP) :- ap(A,AP) ; ap(B,AP).  
ap(equiv(A,B),AP) :- ap(A,AP) ; ap(B,AP).  
ap(not(A),AP) :- ap(A,AP).
```

```
[36]: ?- ap(and(p,q),AP).
```

AP = p

```
[37]: jupyter:retry
```

AP = q

```
[38]: ?- get_aps(and(p,q),Ps).
```

Ps = [p,q]

We can now write a more convenient predicate to find models for a formula:

```
[39]: sat(Formula, Interpretation) :-
    get_aps(Formula, SPs),
    skel(SP, Interpretation), % set up interpretation skeleton
    value(Formula, Interpretation, true).

skel([], []).
skel([AP|T], [AP/_|ST]) :- skel(T, ST). % we replace p by p/_ so that we have an
    ↪ unbound logical variable for every proposition

unsat(Formula) :- \+ sat(Formula, _). % a formula is unsatisfiable if we can find
    ↪ no model, \+ is Prolog negation
```

The skel predicate simply takes a list of propositions and generates a skeleton interpretation with fresh logical variables for the truth values:

```
[40]: ?- skel([a,b,c], R).
```

```
R = [a/_55832,b/_55844,c/_55856]
```

We can now use sat/2 to compute models, without having to provide ourselves an interpretation skeleton:

```
[41]: ?- sat(and(p, or(q, not(p))), Model).
```

```
Model = [p/true,q/true]
```

```
[42]: ?- unsat(and(p, not(p))).
```

```
true
```

1.5 Proof by contradiction

To prove that B is a logical consequence of A we can employ a proof by contradiction: - assume that B is false and - show that this leads to a contradiction.

```
[43]: prove(A,B) :- /* prove that B follows from A */
    unsat(and(A, not(B))).

equivalent(A,B) :- prove(A,B), prove(B,A).
```

```
[44]: ?- prove(and(p,q), or(p,q)).
```

```
true
```

```
[ ]: ?- prove(or(p,q), and(p,q)).
```

```
false
```

```
[ ]: ?- equivalent(and(p,q),or(p,q)).
```

false

```
[49]: ?- prove(and(p,not(p)),and(q,not(q))).
```

true

In case something is not a logical consequence we can use this code to find an explanation (aka counter example):

```
[50]: disprove(A,B,CounterExample) :- /* show why B does not follow from A */  
      sat(and(A,not(B)),CounterExample).
```

```
[51]: ?- disprove(or(p,q),and(p,q),Counter).
```

Counter = [p/true,q/false]

1.6 Some Puzzles

We can return to our Knights & Knave puzzle from the first lecture. There is an island populated by only knights and knaves. Knights always say the truth and knaves always lie. We encounter three persons A,B,C on this island: - 1. A says: “B is a knave oder C is a knave” - 2. B says: “A is a knight”

```
[52]: ?- sat(and(equiv(a,or(not(b),not(c))),equiv(b,a)),I).
```

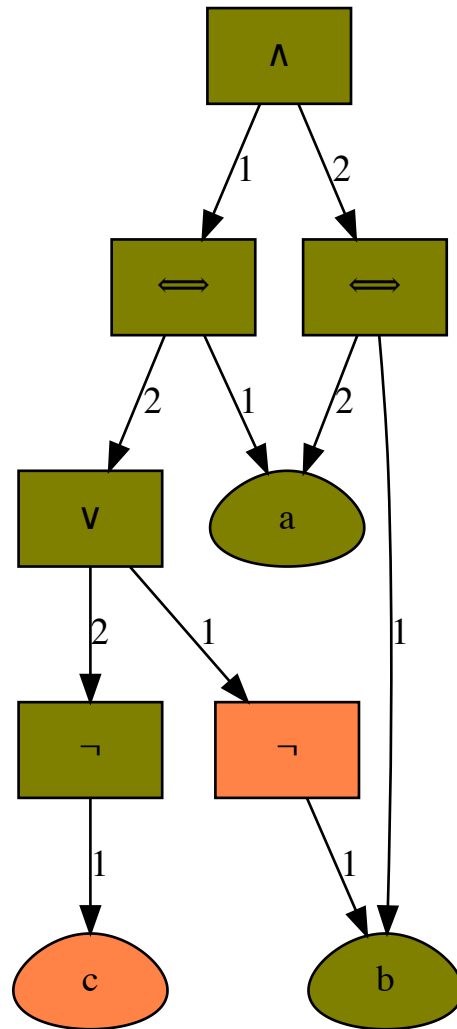
I = [a/true,b/true,c/false]

We can also prove that this is the only solution, i.e., our puzzle implies that A and B are knights and C is a knave:

```
[53]: ?- prove(and(equiv(a,or(not(b),not(c))),equiv(b,a)), and(a,and(b,not(c)))).
```

true

```
[54]: jupyter:  
      ↪show_graph(subnode_val(_,_ ,and(equiv(a,or(not(b),not(c))),equiv(b,a)),[a/  
      ↪true,b/true,c/false]),subtree/3).
```



true

Let us now model another puzzle from Raymond Smullyan.

- The King says: One note tells the truth and one does not
- Note on Door 1: This cell contains a princess and there is a tiger in the other cell
- Note on Door 2: One of the cells contains a princess and the other one contains a tiger.

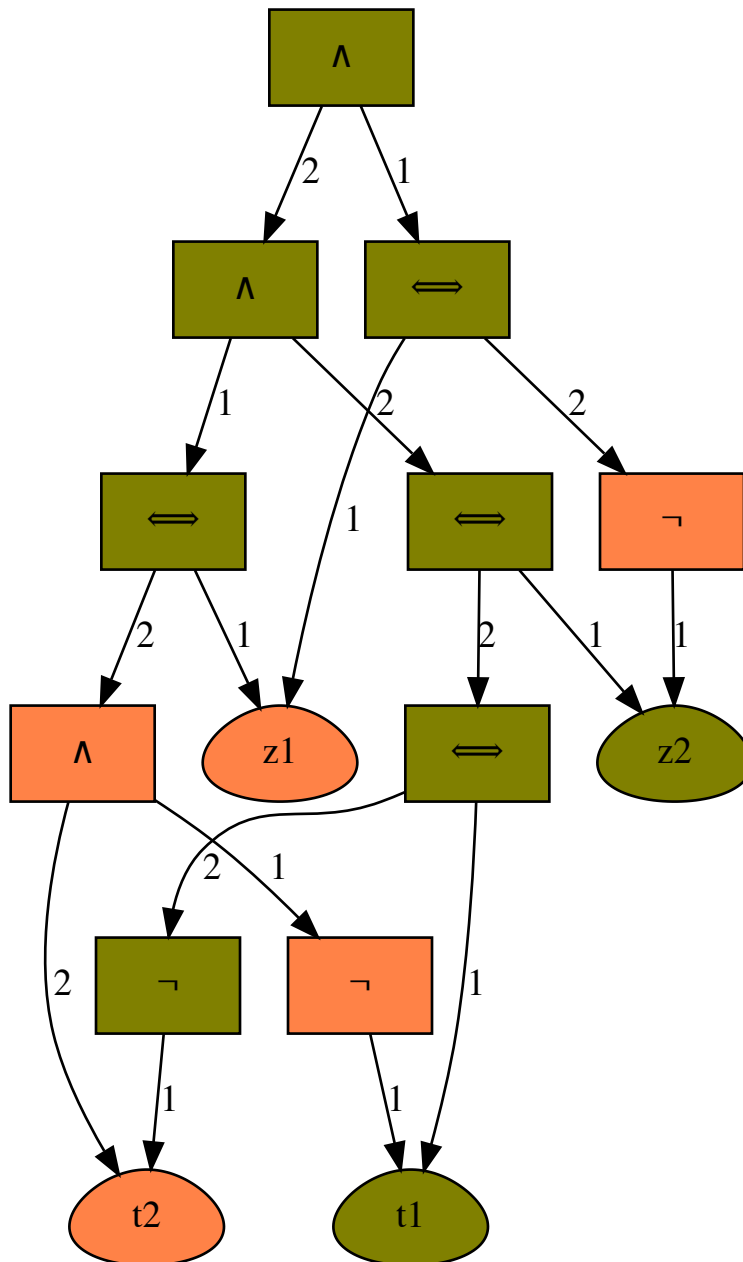
```
[55]: ?-⊥
      ↪sat(and(equiv(z1,not(z2)),and(equiv(z1,and(not(t1),t2)),equiv(z2,equiv(t1,not(t2))))),I).
      ↪
```

```
I = [t1/true,t2/false,z1/false,z2/true]
```

```
[ ]: jupyter:retry.
```

false

```
jupyter:
↪ show_graph(subnode_val(_,_,and(equiv(z1,not(z2)),and(equiv(z1,and(not(t1),t2))
[t1/true,t2/false,z1/false,z2/true]),subtree/3)).
```



true

1.6.1 Appendix: Generating equivalent formulas

The code below can be used to generate formulas in Prolog using a technique similar to iterative deepening. We will study this technique much later in the course in the lectures on search. All you need to know is that `generate_formula` generates formulas from smaller to deeper formulas.

```
[58]: generate_formula(X) :-
      peano(MaxDepth), gen(X,MaxDepth).
      peano(0).
      peano(s(X)) :- peano(X).

      proposition(p).
      proposition(q).
      proposition(r).

      gen(X,_) :- proposition(X).
      gen(and(A,B),s(MaxDepth)) :- gen(A,MaxDepth), gen(B,MaxDepth).
      gen(or(A,B),s(MaxDepth)) :- gen(A,MaxDepth), gen(B,MaxDepth).
      gen(implies(A,B),s(MaxDepth)) :- gen(A,MaxDepth), gen(B,MaxDepth).
      gen(equiv(A,B),s(MaxDepth)) :- gen(A,MaxDepth), gen(B,MaxDepth).
      gen(not(A),s(MaxDepth)) :- gen(A,MaxDepth).
```

```
[59]: ?-generate_formula(X).
```

X = p

```
[60]: jupyter:retry.
```

X = q

```
[61]: jupyter:retry.
```

X = r

We can now use this together with our other Prolog predicates to generate equivalent formulas:

```
[62]: ?- generate_formula(Formula), equivalent(and(p,q),Formula), Formula \= and(p,q).
```

Formula = and(q,p)

```
[63]: jupyter:retry.
```

Formula = and(p,and(p,q))

```
[64]: ?- generate_formula(Formula), equivalent(and(p,p),Formula).
```

Formula = p

```
[65]: ?- generate_formula(Formula), equivalent(and(p,or(p,q)),Formula).
```

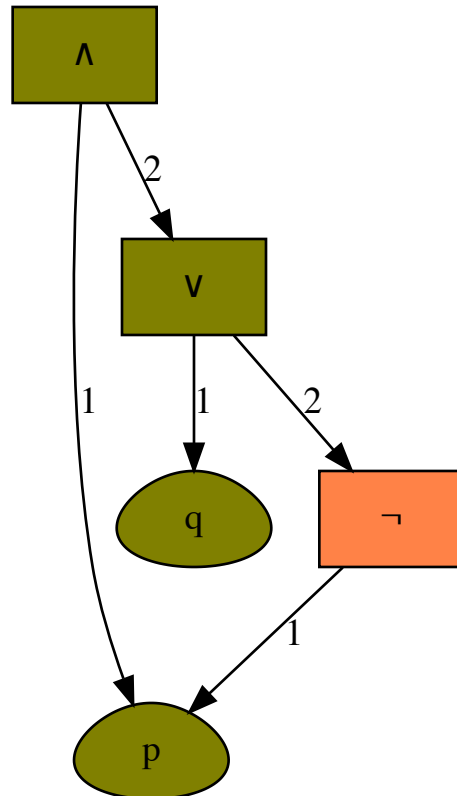
```
Formula = p
```

Another side note: we can use the \$Var feature of Jupyter Prolog to combine the parser and the show_graph feature:

```
[66]: ?- parse_wff("(p (q (¬p)))",Formula).
```

```
Formula = and(p,or(q,not(p)))
```

```
[67]: jupyter:show_graph(subnode_val(_,_,$Formula,[p/true,q/true]),subtree/3).
```



```
true
```

```
[ ]:
```

```
[ ]:
```