Cheatsheet for Answer Set Programming Elizabeth Pankratz, WS 2019/20

1 Concepts, constructs, and notation

Concept	Explanation	Examples	

Aggregates (sums) (Ex. 1 Part 1 1.11)	The syntax `#sum{}` defines a function over the contained set of atoms. If an atom is true, then the number beside each atom is placed into a set (which, crucially, removes duplicates; left), and then each number in that set is added together. If an atom is false, then the number beside it is ignored.
	
	For example, `x(V) :- V = #sum{ 1:a; 2:b }` means `x(3)` if both `a` and `b` are true, `x(1)` if only `a` is true, and `x(2)` if only `b` is true.
	
	To avoid duplicates getting lost, we can construct a tuple with an arbitrary second element which only serves to distinguish the numbers. Then we add the first element in each tuple to get the sum (right).
		`a.`
		`b.`
		`x(V):-V=#sum{1:a; 1:b}`
		`-----------`
		`x(1)`
			`a.`
			`b.`
			`x(V):-V=#sum{1,m:a; 1,n:b}`
			`-----------`
			`x(2)`

	Combining choice rule notation with the `#sum{}` function is a way of making constraints and rules that only hold when the sum is in a particular range. Now the numbers on either side of the function have nothing to do with cardinality or power sets – they determine the range that the sum must (not) fall into. For example, the constraint `:- 1 #sum{…}.` means to eliminate a stable model if the sum it produces is ≥ 1, and the rule `c :- #sum{…} 3.` means that `c` is true iff the sum produced by the other true atoms is ≤ 3.	`{ a, b }.`
		`:- 1 #sum{ 1,x:a; 1,y:b }.`
		`----------`
		`{ }`
		~~`{ a }`~~
		~~`{ b }`~~
		~~`{ a, b }`~~
			`{ a, b }.`
			`:- #sum{ 1:a; 1:b } 1.`
			`----------`
			~~`{ }`~~
			~~`{ a }`~~
			~~`{ b }`~~
			~~`{ a, b }`~~
			[unsatisfiable]

Boolean literals	The directives `#true` and `#false` represent the respective truth values.	`t :- #true.`
		`f :- #false.`
		`notf :- not #false.`
		`----------`
		`{ t, notf }`	

Cardinality rules (Ex. 1 Part 1 1.10)	These are rules (if there's a head) or constraints (if the head is empty) that only apply when the conditions about the cardinality in the body (formatted like a choice rule) are true. For example, the constraint `:- 1 { a;b } 1.` means to eliminate a stable model if it contains either `{a}` or `{b}` (but not the empty set, nor `{a, b}`; see the choice rule syntax above). Also, the rule `c :- 1 { a;b } 1.` means that `c` is true if either `a` or `b` is true, and not true if the empty set or `{a, b}` are true.	`{ a; b }.`
		`:- 1 { a;b } 1.`
		`----------`
		`{ }`
		~~`{ a }`~~
		~~`{ b }`~~
		`{ a, b }`
			`1 { a; b }.`
			`c :- 1 { a;b } 1.`
			`:- not c.`
			`----------`
			`{ a, c }`
			`{ b, c }`
			~~`{ a, b }`~~

Choice rules (Ex. 1 Part 1 1.7, 1.9)	Another way to provoke multiple paths through the rules. The syntax is as follows: `x {…} y.`, where $0 \leq$ `x`, `y` $\leq$ cardinality of `{…}`. This means to take all combinations of atoms in `{…}` (i.e. the power set) to be true, but where the minimum cardinality of the atoms you take is `x` and the maximum cardinality is `y`. If no `x` or `y` are given, then the defaults are `x` $= 0$, i.e. $\varnothing$, and `y` $=$ cardinality of `{…}`. (If `x`, `y` $\geq$ cardinality of `{…}`, then the program is unsatisfiable.)	`0 { a } 1.`
		`----------`
		`{ }`
		`{ a }`
			`1 { a }.`
			`----------`
			`{ a }`

Concept	**Explanation**	**Examples**	

		`{ a; b }.`
		`----------`
		`{ }`
		`{ a }`
		`{ b }`
		`{ a, b }`
			`1 { a; b } 1.`
			`----------`
			`{ a }`
			`{ b }`

Clark's completion $CF(P)$	For each atom in the normal program P, take the disjunction of all bodies that share that atom as the head and put this disjunction $\leftrightarrow$ ("supporting") the head (using classical logic operators). All facts get $\top$, anything unjustifiable gets $\bot$. Models of the completion are supported models (not necessarily stable models). Walkthrough in 2.3.	$P = \left\{ \begin{array}{l} a \leftarrow b. \\ a \leftarrow \neg c. \\ d \leftarrow . \end{array} \right\}$	$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow b \vee (\neg c) \\ b \leftrightarrow \bot \\ c \leftrightarrow \bot \\ d \leftrightarrow \top \end{array} \right\}$

Conditional literals in body (Ex. 1 Part 2 1.4)	These rules hold if the condition in the body is met. The notation is `head :- body : condition.` If the condition is fulfilled by the body, then the head is true. Almost like a for loop.
	
	For example, if we have facts `a(1..2). b(1..2).` and the rule `c :- a(X) : b(X).`, the rule holds and `{c}` becomes part of the model because, for all `X` such that `b(X)` exists, there is also a corresponding `a(X)` (i.e. we have both `b(1), b(2)` and `a(1), a(2)`).
	
	When the condition contains multiple terms, only the variables that apply to all of those statements are checked with respect to the body. For example, given the rule `c :- a(X,Y) : b(X,Y), c(X).`, we ask: for all `X, Y` such that *both* `b(X,Y)` *and* `c(X)` hold, is `a(X,Y)` true? This rule holds in the left example, because the doubled condition ignores all `b(2,_)` since there is no `c(2)`. It does not hold in the right example because there's no `a(2,_)` to match the `b(2,_), c(2)` condition.
	
	Note: As soon as one of the preconditions is false, the body automatically becomes true and the rule holds. This is analogous to implications in logic – if the precondition is false, it doesn't matter, so the consequence is true.
		`a(1, 1..2).`
		`b(1..2, 1..2).`
		`c(1).`
		`c :- a(X,Y) : b(X,Y), c(X).`
		`#show c/0.`
		`----------`
		`{ c }`
			`a(1, 1..2).`
			`b(1..2, 1..2).`
			`c(2).`
			`c :- a(X,Y) : b(X,Y), c(X).`
			`#show c/0.`
			`----------`
			`{ }`

Constants	The `#const` directive lets you choose a constant (conventionally written in lowercase letters) as a placeholder for a particular value.	`#const n=2.`
		`a(n).`
		`----------`
		`{ a(2) }`	

Constraints (Ex. 1 Part 1 1.8, 1.9)	Constraints eliminate those models for which the condition in the body of the constraint is true. They are written as a rule without a head, i.e. starting with `:-`. Read `:- not a.` as "eliminate if model has no `a`" and `:- a.` as "eliminate if model has `a`". For the former, all models are eliminated that don't contain `a`, including ∅ (left). For the latter, all models are eliminated that *do* contain `a`, so ∅ survives (right).	`:- not a.`
		`----------`
		~~`{ }`~~
		[unsatisfiable]
			`:- b.`
			`----------`
			`{ }`

Concept	**Explanation**	**Examples**	

	If a constraint has multiple atoms in its body, then all of those must simultaneously be satisfied by the model for the constraint to apply (left). On the other hand, if several different constraints contain those atoms, then the constraints apply independently (right).
	
	Note: First work out all of the stable models that satisfy the program, and then check to see if constraints apply, and if yes, eliminate that model.
		`:- b, c.`
		`b :- c.`
		`c.`
		`----------`
		~~`{ b, c }`~~
		[unsatisfiable]
			`:- b.`
			`:- c.`
			`c :- d.`
			`d.`
			`----------`
			~~`{ c, d }`~~
			[unsatisfiable]

Dependency graph $G(P)$	Given a rule in a program, draw a directed edge in a graph going in the same direction as in the rule, from the atom(s) in the rule's positive body to the atom in the rule's head. For example, given $a \leftarrow b, \sim e$, remove the negative body and draw an arrow from b to a on the graph. Do this for all rules to create the graph of the program, $G(P)$ (a.k.a. positive dependency graph). Can be used to quickly identify loops.	$\left\{ \begin{array}{ll} a \leftarrow & b \leftarrow c, d \\ c \leftarrow a, \sim e & d \leftarrow b, c \\ b \leftarrow \sim a & d \leftarrow e, \sim a \end{array} \right\}$	

Facts	A type of rule: atoms stated in this way are unconditinoally true. (`a. b.` = `a :- . b :- .`)	`a.`
		`b.`
		`----------`
		`{ a, b }`	

Interpretation	An assignment of all atoms in a program to true and false.	–	
Interpretation, partial	An assignment of some atoms in a program to true and false, written $\langle\{\top\},\{\bot\}\rangle$, e.g. $\langle\{a\},\{b\}\rangle$ means that a is true and b is false.	–	

Intervals	Ranges of numbers are notated as follows: `start .. end`, where `end` $\geq$ `start`. Going in the other direction (i.e. starting with the larger value) returns no values.	`a(1..3).`
		`----------`
		`{ a(1), a(2), a(3) }`
			`a(3..1).`
			`----------`
			`{ }`

Loops	Two or more atoms that are defined in terms of each other. Can be easily identified by drawing a dependency graph. If two loops have a node in common, then those two loops can also be joined into a larger loop. A program can have multiple loops.	–	

Loops, even through negation/ multiple paths (Ex. 1 Part 1 1.4)	Triggered by a particular mutually contradictory statement (at right), take different paths through the rules, assuming for the first path that, say, `a` is true and `b` is false, and for the next path, that `a` is false and `b` is true. This can lead to multiple stable models.	Trigger:
		`a :- not b.`
		`b :- not a.`
		`----------`
		`{ a }`
		`{ b }`
			`a :- not b.`
			`b :- not a, d.`
			`d.`
			`-----------`
			`{ a, d }`
			`{ b, d }`

Concept	Explanation	Examples

	If there are multiple triggers, i.e. that mutually contradictory statement shows up once with, say, `a` and `b` and also once with `c` and `d`, then every combination of these has to be explored.	`a :- not b.`
		`b :- not a.`
		`c :- not d.`
		`d :- not c.`
		`----------`
		`{ a, c }`
		`{ a, d }`
		`{ b, c }`
		`{ b, d }`

Loops, odd through negation/ paradoxes (Ex. 1 Part 1 1.5, 1.6)	There is no way to resolve the statement `a :- not a.`, because if the body tells us one truth value for `a`, the head tells us the other. So this statement produces no stable models, i.e. it is unsatisfiable.	`a :- not a.`
		`----------`
		[unsatisfiable]

	If the rest of the body is true, then we reach the paradox, meaning that the model is unsatisfiable (left). If there is one false atom in the body, then the rule can no longer tell us anything about the head, so we avoid the paradox and can satisfy the program (right).	`a :- not a, d.`
		`d.`
		`----------`
		[unsatisfiable]
		
		`a :- not a, b.`
		`b :- c.`
		`----------`
		`{ }`

Loop formulas $LF(P)$	For atoms in a loop to be true, they need to have some kind of external support. i.e. be justifiable using a rule whose head contains a loop atom and whose So, loop formulas are used for identifying external support for loop atoms: they are the disjunctions of all bodies of rules (a) whose head contains a loop atom and (b) whose positive body does not contain any atoms from the loop (it's OK if the negative body contains loop atoms, those may be contained in loop formulas). There is one loop formula per loop in the program. They can also be calculated on unfounded sets.	(walkthrough for a program in 2.4, for unfounded sets in 2.7)
Model	A set is a model if it satisfies all rules of a program. A rule is satisfied if only its head is in the set, or if the head and the entire body is in the set. For example, given the program `a :- b, c, d.`: `{ c, d }` Not a model (does not satisfy body or head of rule) `{ b, c, d }` Not a model (satisfies body of rule but not head) `{ a, b, c, d }` A model (satisfies body and head of rule) Stable model (adds acyclic support) ⊆ supported model (adds support) ⊆ model	–

Model, stable	A set is a stable model (or "answer set") if it is a model (i.e. if it satisfies all the rules of a program) *and* if there is justification in the program for every atom (i.e. if each atom is provably true). The program `a :- b, c, d.` has no stable models, since none of the atoms are justified. There can be multiple stable models if there are multiple "paths" through a set of rules, e.g. as provoked by a choice rule (see below). You compute stable models using reducts (see below).
	
	Note: Stable models are represented in what follows as atoms between `{}` below `-----` after all of the rules of the program.
		`a.`
		`----------`
		`{ a }`

Concept	Explanation	Examples	
Model, supported	A supported model is one where the program offers support for every atom in the model. We get supported models from the Clark's Completion of a program. If a program is tight (i.e. there are no loops), then a supported model is a stable model. If the program is not tight, then loop formulas are used to determine whether a supported model is a stable model (if the supported model satisfies the implication in the loop formula, then it is a stable model).	(walkthrough in 2.3)	

Mutual definition (Ex. 1 Part 1: 1.1)	If two atoms are defined exclusively in terms of one another, both are considered false, since they cannot independently be proven true. The empty set is the result because it is the minimal stable model (but it can be removed by constraints; see below).	`a :- b.`
		`b :- a.`
		`----------`
		`{ }`	

Negation (Ex. 1 Part 1: 1.2, 1.3)	True if the contained atom is not true or not provable (see "Truth" above).	`a :- not b.`
		`----------`
		`{ a }`
			`a :- not b, c.`
			`b :- not a, d.`
			`c.`
			`----------`
			`{ a, c }`

Optimisation: maximise, minimise (Ex. 1 Part 2 1.5)	Optimisation lets us choose between several stable models to find one that meets a particular condition. The `#maximize` and `#minimize` directives do the same thing as the `#sum` directive above, but with one more step: they calculate the sums for all stable models and only let those models through into the final answer that have the largest (for `#maximize`) or smallest (for `#minimize`) sum.
	
	In the example, the second line generates the sets `{a(1)}`, `{a(2)}`, and `{a(1), a(2)}` (see set generators above). The first two of those have a sum of 1, while the second has a sum of 2. Because its sum is greater than the minimum of 1, `{a(1), a(2)}` is removed as a candidate by the `#minimize` directive.
		`b(1..2).`
		`1 { a(X) : b(X) }.`
		`#minimize{ 1,X : a(X) }.`
		`#show a/1.`
		`----------`
		`{ a(1) }`
		`{ a(2) }`	

Program	Defined as a set of rules. A positive program contains only rules that contain no negation. A normal program contains at least one rule containing negation.		

Python functions (Ex. 1 Part 2 1.6)	Between the `#script(python)` and `#end.` directives, you can define a Python function that returns some value. When you call that function later in the ASP program (prefacing the function name with the at symbol), the value that the function returns is what will be used, say, as a value for a variable.	`#script(python)`
		`def value():`
		`    return 1`
		`#end.`
		
		`a(X) :- X = [AT]value().`
		`----------`
		`{ a(1) }`	

Concept	Explanation	Examples
Reduct	The reduct of a set X is written as P^x and is used to determine which models of a program are stable. It makes a normal program P positive, wrt a given set of atoms X. Formally: $P^X = \{h(r) \leftarrow B^+(r) \mid r \in P \text{ such that } B^-(r) \cap X = \varnothing\}$ where h = head of rule, r = rule, $B^+(r)$ = atoms in body of rule that are positive (i.e. contain no negation), $B^-(r)$ = atoms in body of rule that are negative. What this says is that the reduct contains (a) no rules that have an atom in their negative body that is also in X (these are removed because we know we can't use them to justify anything, since the atom is negative in the rule but positive in our set X) and (b) only the positive parts of rules that have a negative atom in their body whose positive counterpart isn't in X (i.e. all negative atoms are removed). You are left with a positive program P^X.	(walkthrough in 2.2)
Reduct, consequence of	The atoms that you can justify using the reduct P^X, i.e. the left-over positive program (see above), are the consequence of the reduct, written $Cn(P^X)$. If $Cn(P^X) = X$, i.e. if what you put in is the same as what you get out, then you have yourself a stable model.	–
Rules	Consist of a head (to the left of the implication `:-`, the computer-readable version of $\leftarrow$) and a body (to the right of the implication) and ends with a period. The head is true if the body is true. If the body contains conjunctions, as soon as one conjunct is false, the rule cannot be used any more because it is not true. *Note:* As soon as one rule shows an atom to be true, even if other rules do not, then that atom is always true. If a rule's body is false, it does not mean that the head is false – it means we cannot use the rule to prove the head. So even if we have conflicting results (i.e. one rule says an atom is true, another does not say it's true), then the atom is true.	`a :- b.`
Safety	A rule is safe if all variables appearing anywhere within it also appear in its positive body, or in the condition of a set generator. A program is safe if all rules are safe. A program needs to be safe because an unsafe program means that there is no domain of the variables (i.e. no mapping of variables to atoms), and we need a domain in order to ground the program so it can be solved.	–

Set generators (Ex. 1 Part 2 1.2)	Generates candidate stable models using the following notation: `{set contents : condition}`, where numbers on either side indicate the required cardinality of the set (see cardinality rules above). For example, given the facts `b(1). b(2).`, the set generator `{ a(X) : b(X) }.` generates the following four sets: `{}`, `{a(1)}`, `{a(2)}`, and `{a(1), a(2)}`. The facts we know are also part of each model, so for example, the model formed by the set generator's second set is `{ b(1), b(2), a(1) }`.
	
	These can also be used in the heads of rules, as shown in the examples. Note that if there are facts of form `c(Y)` for multiple `Y`s, then the rule is applied for each candidate model as many times as there are `Y`s (right).
		`b(1). b(2). c(3).`
		`1 { a(X, Y) : b(X) } 1 :- c(Y).`
		`#show a/2.`
		`----------`
		`{ a(2, 3) }`
		`{ a(1, 3) }`
		
		`b(1). b(2). c(3). c(4).`
		`1 { a(X, Y) : b(X) } 1 :- c(Y).`
		`#show a/2.`
		`----------`
		`{ a(1, 3), a(1, 4) }`
		`{ a(1, 3), a(2, 4) }`
		`{ a(2, 3), a(1, 4) }`
		`{ a(2, 3), a(2, 4) }`

Concept	**Explanation**	**Examples**
Show statements (Ex. 1 Part 2 1.6)	Show statements can change the parts of the stable models that CLINGO displays. Simply printing `#show.` hides everything. `#show a/1.` shows only instances of the predicate `a` that has an arity (valence) of 1 (i.e. takes only one argument, e.g. `a(X)`). `#show a('yeah') : #true.` is a conditional show directive that shows what's left of the colon if the condition on the right holds (as it does here; see Boolean literals above).	–
Tightness	If a program P has no loops, then P is said to be tight, and in those cases, a supported model of P is also a stable model of P.	–
Truth	An atom is true if it can be proven by a fact or some other kind of rule, or if we assume it is as one path in a choice rule (see below). If we do not know whether the atom is true or cannot prove that it is, we consider it to be false (closed world assumption).	–
Unfounded sets	A set of atoms U is unfounded wrt some partial interpretation if all of the rules whose heads are atoms in U meet at least one of the following conditions: • one atom of the rule's positive body is false in the partial interpretation (i.e. $\langle\{\},\{\text{atom}\}\rangle$) • one atom of the rule's negative body is true in the partial interpretation (i.e. $\langle\{\text{atom}\},\{\}\rangle$) • one atom of the rule's positive body is also in U The easiest way to find unfounded sets is to first find the greatest unfounded set (GUS) and then check every subset in the GUS' power set for unfoundedness, using the above conditions. Also, $\varnothing$ is by definition an unfounded set.	(walkthrough in 2.6)
Unfounded set, greatest	aka GUS. The union of all unfounded sets. Computed by subtracting the minimal model of the program's reduct wrt some partial interpretation from the set of all atoms.	(walkthrough in 2.6)
Fitting Semantics Φ	A program's completion in the form of an operator (as such, Fitting Semantics results in supported models in the form of an interpretation, formatted like $\langle\{\top\},\{\bot\}\rangle$ where the stuff in $\{\top\}$ is the supported model. To get stable models from this output, somehow remove the unfounded sets from this?	–
Wellfounded Semantics Ω	a	–
Nogoods	a	–

2 Algorithms

2.1 Tips for determining stable models of normal programs

If there are no obvious starting points for solving a program, i.e. no facts, unjustifiable atoms (those not appearing in heads of rules), or choice rules or even loops through negation, then start by assuming that the heads of negative rules are true and looking for what that implies.
Also, atoms in stable models don't need to touch all rules—they can leave some of the rules untouched and that's fine. All they need is for every atom in the stable model to be justified in the program, not the other way around.

2.2 Computing the reduct

To compute the reduct P^X of a set X for a program P, just follow these two! easy! steps!

1. Remove all rules in P whose bodies contain one or more negative atoms whose positive counterparts are contained in X.
2. Then, remove any remaining negative atoms whose positive counterparts were not contained in X, but leave the rest of the rule intact.

This remaining set of positive rules is your reduct P^X. Then, to determine the consequence of the reduct $Cn(P^X)$, create a set out of only those atoms which are justified by the remaining rules ($Cn(P^X)$ = the "transitive closure" or "minimal model of the reduct"). If this closure matches the input set X, then X is a stable model of P.

Let's see this in action for the program $P =$ `b :- not a. a :- not b.` and the set $X =$ `{ a }`.

Step	Reduct-in-progress

0 (original program)	`b :- not a.`
	`a :- not b.`

1 (remove rules where any atom in X is negative)	~~`b :- not a.`~~ `a :- not b.`
2 (remove all remaining negative atoms)	~~`b :- not a.`~~ `a :-` ~~`not b.`~~
P^X	`a :- .`

The consequence of this stable model is $Cn(P^X) =$ `{ a }`. Because what went in, X, is the same as what came out, $Cn(P^X)$, we know that `{ a }` is a stable model of P.

2.3 Computing the Clark's Completion and getting its supported models

Given a program P, compute the completion $CF(P)$ as follows.

1. Begin by writing down every atom in the program followed by $\leftrightarrow$ on a new line.
2. Then, for each atom, find the rules with that atom in the head in P.
   (a) If there is only one, then write its body to the right of the $\leftrightarrow$ (using logical negation).
   (b) If there is more than one rule, create a disjunction out of all of the possible bodies and write this disjunction to the right of $\leftrightarrow$ (using logical negation and disjunction).
   (c) If there is a fact for this atom, write $\top$.
   (d) If there is no justification for this atom, write $\bot$.

For example, given the program $P = \left\{ \begin{array}{l} a \leftarrow b \\ a \leftarrow \neg c \\ b \leftarrow d \\ d \leftarrow \end{array} \right\}$, the completion is computed as shown.

Step	Completion-in-progress
1 (set-up)	$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \\ b \leftrightarrow \\ c \leftrightarrow \\ d \leftrightarrow \end{array} \right\}$
2a (one rule)	$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \\ b \leftrightarrow d \\ c \leftrightarrow \\ d \leftrightarrow \end{array} \right\}$
2b (multiple rules)	$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow b \vee (\neg c) \\ b \leftrightarrow d \\ c \leftrightarrow \\ d \leftrightarrow \end{array} \right\}$
2c (facts)	$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow b \vee (\neg c) \\ b \leftrightarrow d \\ c \leftrightarrow \\ d \leftrightarrow \top \end{array} \right\}$
2d (no justifcation)	$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow b \vee (\neg c) \\ b \leftrightarrow d \\ c \leftrightarrow \bot \\ d \leftrightarrow \top \end{array} \right\}$

To find the supported model of the completion: if the completion contains $\top$s or $\bot$s, start with those, seeing what they imply. Write out everything you know from the completion (i.e. positive and negative atoms), adding to this set iteratively as the known atoms imply others, and then take the positive atoms as the supported model. For the above example:

$$\{d, \neg c\} \rightarrow \{d, \neg c, b\} \rightarrow \{d, \neg c, b, a\} = \{a, b, d\}$$

Can also look for odd loops through negation (i.e. something like $\{d \leftrightarrow \neg e,\ e \leftrightarrow \neg d\}$, which starts you off with the beginnings of two supported models, one containing $\{d, \neg e\}$ and one $\{e, \neg d\}$). If nothing else, assume one atom and use trial-and-error. If the program contains a loop, there will be one supported model with the loop atoms and one without.

2.4 Computing loop formulas

Given a program P that contains a set of loops $loops(P)$, compute the loop formulas $LF(P)$ (= formulas that forbid a loop from being in a stable model) as follows.

1. Begin by writing out the disjunction of all atoms involved in each loop, followed by $\rightarrow$, on a new line.
2. Using the original program (not the completion):
   (a) Identify all rules whose heads contain an atom involved in the loop.
   (b) Disqualify any rules if their positive body contains any of the atoms involved in the loop.
   (c) Take the bodies of the remaining rules and form a disjunction out of them, writing this to the right of $\rightarrow$.

For example, for the program $P = \left\{ \begin{array}{lll} a \leftarrow & b \leftarrow \sim a & b \leftarrow c, d \\ c \leftarrow a, \sim e & d \leftarrow b, c & d \leftarrow e, \sim a \end{array} \right\}$, which contains $loops(P) = \{\{b, d\}\}$:

Step	Loop-formula-in-progress	Rules	
1 (set-up)	$b \vee d \rightarrow$		
2a (identify rules)		$b \leftarrow \sim a$ $d \leftarrow b, c$ $b \leftarrow c, d$ $d \leftarrow e, \sim a$	
2b (disqualify rules)		$b \leftarrow \sim a$ $d \leftarrow \underline{b}, c$ $b \leftarrow c, \underline{d}$ $d \leftarrow e, \sim a$	✓ × (loop atom in pos body) × (loop atom in pos body) ✓
2c (disjunction)	$b \vee d \rightarrow (\neg a) \vee (e \wedge \neg a)$		

2.5 Using loop formulas to identify stable models

To see if a supported model is also a stable model, we can check if the supported model satisfies the implication in the loop formula. If yes, then it is a stable model.

Formally: stable models = $CF(P) \cup$ loop formulas.

For the above example, the supported models of the completion are $\{a, c\}$ and $\{a, c, b, c\}$ (one with the loop atoms and one without).

✓	$\{a, c\}$	The precondition of the loop formula, $b \vee d$, is false, since neither b nor d is in the model. This makes the implication true. Thus $\{a, c\}$ is a stable model.
×	$\{a, c, b, c\}$	The precondition of the loop formula is true, since b or d is in the model (in fact, both are). However, the RHS is not fulfilled: we have $\neg e$ (by virtue of e not being in the model) and a, while the implication requires e and $\neg a$. Thus the loop formula is not satisfied, so $\{a, c, b, c\}$ is not a stable model.

It is possible for none of the supported models to be stable models if no models satisfy the loop formulas. Then the program has no stable models.

2.6 Finding unfounded sets

abc

2.7 Using loop formulas on unfounded sets

abc

2.8 Computing Fitting Semantics

abc

2.9 Computing Wellfounded Semantics

abc

2.10 Finding nogoods

abc

2.11 Using nogoods to do conflict analysis

abc

3 Answers to the quiz questions

2 Positive programs and stable models

- If a positive rule is satisfied by a set of (ground) atoms, then it is not true that each of its supersets satisfies the rule as well.
  - Because $\varnothing$ satisfies a positive rule like $a \leftarrow b$, but $\{b\}$ is a superset of $\varnothing$ and does not satisfy this rule (would need $\{b, a\}$).
- Each positive logic program has some model.
- Each positive logic program has some stable model.
- A positive rule with variables is not necessarily satisfied by a set of (ground) atoms if some ground instance of the rule is satisfied by the set of atoms (all ground instances of the rule have to be satisfied, not just one).
- The stable model of a positive logic program is contained in each model of the logic program (because a SM is the minimal requirement for a set to be a model).

3 Normal logic programs, stable models, reducts

- Each normal logic program has some model.
- However, each normal logic program does not necessarily have a stable model.
- Given a normal logic program P, it is not true that the reduct P^X is contained in P^Y for each subset X of a set of Y atoms.
  - Because the reduct gets smaller if there are more atoms, since more rules get crossed out. So X being a subset of Y (i.e. Y has more atoms, is more restrictive) means that the reduct P^Y is a subset of the reduct P^X, rather than the other way around.
- If a set X of atoms is a model of a normal logic program P, then X is a model of the reduct P^Y for each superset Y of X.
- Given a normal logic program P, each stable model of P is a proper subset minimal model of P.

4 Choice rules and integrity constraints

- Each interpretation (combination of atoms) satisfies a choice rule without lower and upper bounds, i.e. $0\{\ldots\}\infty$.
- It is not true that a logic program without choice rules has at most one stable model.
  - If a logic program has even loops through negation, then there can be two stable models, since $1\{a, b\}1$ is the same as $\{a \leftarrow\sim b, b \leftarrow\sim a\}$.
- The stable models of a logic program $P \cup C$, where C is a set of integrity constraints, are stable models of P.
  - Constraints can only remove answer sets / prune them, not change whether or not something is a stable model or not.
- A logic program without integrity constraints does not necessarily have a stable model.
  - For example, if there is an odd loop (a.k.a. a paradox), like $a \leftarrow \mathit{nota}$ – this can result in no SM without there being an integrity constraint.
- ASP systems do not first evaluate choice rules and then prune stable model candidates that violate integrity constraints.

5 Safety, grounding

- It is not true that a rule is safe if all variables in its head also appear in its body. It should be: A rule is safe if all variables anywhere within it also appear in its positive body.
  - Example of unsafe rule: $a(X) \leftarrow\sim b(X)$. X does not appear in a positive body.
- It is not true that each safe program has a finite number of finite stable models.
  - Example of a program that is safe but that gives an infinite SM: $\{P(X+1) \leftarrow P(X).P(0).\}$ gives $\{0, 1, 2, 3, \ldots, \infty\}$.
- It is not true that grounders (partially) evaluate the body literals of a rule in the order given by an encoding. Order of literals in body doesn't matter for grounder.

6 Clark's Completion and loop formulas

- It is not true that, if a set of atoms X is a model of a normal program P, then X is a model of $CF(P)$.
  - Example: The set $X = \{a\}$ is a model of $P = \{a \leftarrow\sim a\}$, but not of $CF(P) = \{a \leftrightarrow \neg a\}$
- Given a normal program P, if a set of atoms X is a model of $CF(P)$, then X is a model of P.
- Given a normal program P, it is not true that, if a set of atom X is a model of $CF(P)$, then X is a stable model of P.
  - Example: Given the program $P = \{a \leftarrow a\}$, $X = \{a\}$ is a model of the completion but not a stable model.
- If X is a stable model of a normal program P, then X is a model of $CF(P)$.
- Given a normal program P, it is not true that, if $CF(P)$ has (at least) one model, then P has (at least) one stable model.
  - Example: For the program $P = \{a \leftarrow a, a \leftarrow\sim a\}$, $X = \{a\}$ is a model of the completion $CF(P) = \{a \leftrightarrow (a \vee \neg a)\}$, but P has no stable models.
- Let P be a normal program and $L_1, L_2 \in \mathit{loop}(P)$. If $(L_1 \cap L_2) \neq \emptyset$, then $(L_1 \cup L_2) \in \mathit{loop}(P)$.

8 Nogoods, conflict-driven ASP solving

- It is not the case that, to compute the stable models of a logic program P, ASP systems first construct a flat internal representation of the nogoods in $\Delta_P \cup \Lambda_P$.
- For each tight logic program, the solutions of Δ_P coincide with the solutions of $\Delta_P \cup \Lambda_P$.
- A Unique Implication Point (UIP) of a decision level, along with literals assigned at smaller decision levels, lead to a conflict by propagation.
- Each decision level beyond 0 at which a conflict occurs has some UIP.
- If a logic program has no stable model, it is false that conflict-driven ASP solving procedures may loop infinitely.

This version compiled on January 3, 2020.