Cheatsheet for Answer Set Programming Elizabeth Pankratz, Winter Semester 2019/20

1 Concepts, constructs, and notation

Concept	**Explanation**	**Examples**
Rules	Consist of a head (to the left of the implication `:-`, the computer-readable version of ←) 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.`

Facts	The atoms stated in this way are true. (Can think of facts like rules without a body, meaning that they are unconditionally true: `a :- .  b :- .`)	`a.`
		`b.`
		`----------`
		`{ a, b }`

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).	–
Models	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 models	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
Reducts	The reduct of a set X is written as P^x and is used to determine which models of a program are stable. 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.	(A step-by-step walkthrough of computing a reduct is given in Section 2.)
Consequence of a reduct	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.	–
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).` `----------` `{ }`
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) }`
Boolean literals	The directives `#true` and `#false` represent the respective truth values.	`t :- #true.` `f :- #false.` `notf :- not #false.` `----------` `{ t, notf }`
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.` `----------` `{ }`

Concept	**Explanation**	**Examples**	

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 }`

“Even loops 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 }`

	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 }`	

“Odd loops 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.`
			`----------`
			`{ }`

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 }`

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.`
			`----------`
			`{ }`

	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]

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 }`~~

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).
	
	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.`
		`x(V):-V=#sum{1:a; 1:b}`
		`-----------`
		`x(1)`
		
		`{ a, b }.`
		`:- 1 #sum{ 1,x:a; 1,y:b }.`
		`----------`
		`{ }`
		~~`{ a }`~~
		~~`{ b }`~~
		~~`{ a, b }`~~
			`a.`
			`b.`
			`x(V):-V=#sum{1,m:a; 1,n:b}`
			`-----------`
			`x(2)`
			
			`{ a, b }.`
			`:- #sum{ 1:a; 1:b } 1.`
			`----------`
			~~`{ }`~~
			~~`{ a }`~~
			~~`{ b }`~~
			~~`{ a, b }`~~
			[unsatisfiable]

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).	–	

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	

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.`
			`----------`
			`{ }`

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) }`	

Concept	**Explanation**	**Examples**

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) }`

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 $Cn(P^X)$ (a.k.a. the "transitive closure" apparently), create a set out of only those atoms which are justified by the remaining rules. If this set 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.

This version compiled on November 5, 2019.