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

Concept	Explanation	Examples
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).	–
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.`
Stable model	The atoms that satisfy (i.e. are true within the confines of) a particular set of rules (there is also a complicated mathematical definition on the slies if that's your thing). There can be multiple stable models in your answer set if there are multiple "paths" through a set of rules, e.g. as provoked by a choice rule (see below). Represented as atoms between `{}` below a line of `-----` after all of the rules.	`a.` `----------` `{ a }`
Fact	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 }`
Mutual definition (Ex. 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. 1.2, 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 }`

Concept	Explanation	Examples	
"Even loops through negation"/ multiple paths (Ex. 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. 1.5, 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. 1.7, 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 }`
		`{ a; b }.` `----------` `{ }` `{ a }` `{ b }` `{ a, b }`	`1 { a; b } 1.` `----------` `{ a }` `{ b }`

Concept	Explanation	Examples	
Constraints (Ex. 1.8, 1.9)	Constraints eliminate those models from the answer set 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 answer set has no `a`" and `:- a.` as "eliminate if answer set 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 answer set 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 from the answer set.	`:- b, c.` `b :- c.` `c.` `----------` ~~`{ b, c }`~~ *[unsatisfiable]*	`:- b.` `:- c.` `c :- d.` `d.` `----------` ~~`{ c, d }`~~ *[unsatisfiable]*
Cardinality rules (Ex. 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 }`~~
Aggregates (sums) (Ex. 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)`

Concept	Explanation	Examples	
	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]*