2

Answer Set Prolog (ASP)

Answer Set Prolog is a declarative language; thus, an ASP program is a collection of statements describing objects of a domain and relations between them. Its semantics defines the notion of an **answer set** – a possible set of beliefs of an agent associated with the program.[1] The valid consequences of the program are the statements that are true in all such sets of beliefs. A variety of tasks can be reduced to finding answer sets or subsets of answer sets, or computing the consequences of an ASP program.

2.1 Syntax

Whenever we define a formal language, we start with its alphabet. In logic, this alphabet is usually called a signature. Formally, a **signature**[2] is a four-tuple $\Sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{P}, \mathcal{V} \rangle$ of (disjoint) sets. These sets contain the names of the objects, functions, predicates, and variables used in the program. (Predicate is just a term logicians use instead of the word "relation." We use both words interchangeably.) Each function and predicate name is associated with its **arity** – a non-negative integer indicating the number of parameters. For simplicity we assume that functions always have at least one parameter. Normally, the arity is determined from the context. Elements of $\mathcal{O}, \mathcal{F}$, and $\mathcal{P}$ are often referred to as *object, function*, and *predicate constants*, respectively. Often the word *constant* in this context is replaced by the word *symbol*. In the family relations program from Chapter 1, our signature Σ_f is

$$\begin{aligned}
&\mathcal{O} = \{john, sam, alice, male, female\} \\
&\mathcal{F} = \emptyset \\
&\mathcal{P} = \{father, mother, parent, child, gender\} \\
&\mathcal{V} = \{X, Y\}
\end{aligned}$$

[1] Historically, belief sets – under the name of *stable models* – were defined on a special class of logic programs written in the syntax of the programming language Prolog. After the definition was extended to apply to the broader class of programs defined in this chapter, the term *answer set* was adopted.

[2] In some books the definition of a signature slightly differs from the one used here in that it does not contain $\mathcal{O}$; instead, object constants are identified with function constants of arity 0.

Whenever necessary we assume that our signatures contain standard names for non-negative integers, functions, and relations of arithmetic (e.g., $+, *, \leq$, etc.).

Sometimes it is convenient to expand the notion of signature by including in it another collection of symbols called **sorts**. Sorts are normally used to restrict the parameters of predicates, as well as the parameters and values of functions.[3] For this purpose every object constant and every parameter of a predicate constant is assigned a sort, just as for parameters and values of functions. The resulting five-tuple is called a **sorted signature**. For instance, signature Σ_f can be turned into a sorted signature Σ_s by viewing $gender$ as a sort (instead of a predicate symbol), introducing a new sort, $person$, and assigning proper sorts to object constants of the signature and to parameters of its predicate symbols. Sometimes these assignments are written as $gender = \{male, female\}, person = \{john, sam, alice\}$, $father(person, person)$, etc.

Object and function constants are used to construct terms. Terms not containing variables usually name objects of the domain. For instance, object constant sam is a name of a person, term $max(2, 1)$ is a name for the number 2, etc. Here is the definition.

Terms (over signature Σ) are defined as follows:

1. Variables and object constants are terms.
2. If $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is a term.

For simplicity arithmetic terms are written in the standard mathematical notation; for example, we write $2 + 3$ instead of $+(2, 3)$. Terms containing no symbols for arithmetic functions and no variables are called **ground**. Here are some examples from our family program:

- $john$, sam, and $alice$ are ground terms.
- X and Y are terms that are variables.
- $father(X, Y)$ is *not* a term.

If a program contains natural numbers and arithmetic functions, then both $2 + 3$ and 5 are terms; 5 is a ground term, whereas $2 + 3$ is not.

Let's extend our program signature to include the function symbol car. (Intuitively, we are assuming here that a person X has exactly one car, denoted by $car(X)$.) Now we can also make ground terms $car(john)$,

[3] This idea is familiar to users of procedural languages as parameters of procedures and functions are often associated with a type. Output types of parameters of functions are also common.

$car(sam)$, and $car(alice)$ and nonground terms $car(X)$ and $car(Y)$. So far our naming seems reasonable. There is, however, a complication – according to our definition $car(car(sam))$ is also a ground term, but it does not seem to denote any reasonable object of our domain. To avoid this difficulty consider the sorted signature Σ_s defined as follows:

- Object constants of Σ_s are divided into sorts: $gender = \{male, female\}$, $person = \{john, sam, alice\}$, and $thing = \{car(X) : person(X)\}$.[4]
- $\mathcal{F} = \{car\}$ where car is a function symbol that maps elements of sort $person$ into that of $thing$ (e.g., car maps $john$ into $car(john)$).
- In addition to predicate symbols with sorted parameters such as $father(person, person)$, $mother(person, person)$, and so on, of Σ_f, we also add a new predicate symbol with parameters of two different sorts denoted by $owns(person, thing)$.

The definition of a term for a sorted signature is only slightly more complex than the one for an unsorted signature. For $f(t_1, \ldots, t_n)$ to be a term, we simply require sorts of the values of terms $t_1, \ldots, t_n$ to be compatible with that of the corresponding parameter sorts of f. So the terms of a sorted signature Σ_s are $john, X, car(sam), car(alice), car(X)$, etc. Note, however, that, because $car(sam)$ is of sort $thing$ and function symbol car requires sort $person$ as a parameter, $car(car(sam))$ is not a term of Σ_s.

Term and predicate symbols of a signature are used to define statements of our language. An **atomic statement**, or simply an **atom**, is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms. If the signature is sorted, these terms should correspond to the sorts assigned to the parameters of p. (If p has arity 0 then parentheses are omitted.) For example, $father(john, sam)$ and $father(john, X)$ are atoms of signature Σ_s, whereas $father(john, car(sam))$ is not. If the signature were to contain a zero-arity predicate symbol $light\_is\_on$, then $light\_is\_on$ would be an atom. If the ts do not contain variables, then $p(t_1, \ldots, t_n)$ says that objects denoted by $t_1, \ldots, t_n$ satisfy property p. Otherwise, $p(t_1, \ldots, t_n)$ denotes a condition on its variables. Other statements of the language can be built from atoms using various logical connectives.

A **literal** is an atom, $p(t_1, \ldots, t_n)$ or its negation, $\neg p(t_1, \ldots, t_n)$; the latter is often read as $p(t_1, \ldots, t_n)$ *is false* and is referred to as a **negative**

[4] Here $\{t(X) : p(X)\}$, where $t(X)$ is a term and $p(X)$ a condition, is the standard set-building notation read as *the set of all* $t(X)$ *such that* X *satisfies* p.

literal. An atom and its negation are called **complementary**. The literal complementary to l is denoted by $\bar{l}$. An atom $p(t_1, \ldots, t_n)$ is called **ground** if every term $t_1, \ldots, t_n$ is ground. Ground atoms and their negations are referred to as **ground literals**.

Now we have enough vocabulary to describe the syntax of an ASP program. Such programs serve as an agent's knowledge base, so we often use the words "program" and "knowledge base" synonymously. A **program** Π of ASP consists of a signature Σ and a collection of **rules** of the form:

$$l_0 \ or \ \ldots \ or \ l_i \leftarrow l_{i+1}, \ldots, l_m, \ not \ l_{m+1}, \ldots, \ not \ l_n \qquad (2.1)$$

where ls are literals of Σ. (To make ASP programs executable, we replace $\neg$ with $-$, $\leftarrow$ with $:-$, and or with $|$.)

For simplicity we assume that, unless otherwise stated, signatures of programs consist only of symbols used in their rules.

Symbol not is a new logical connective called **default negation**, (or **negation as failure**); $not \ l$ is often read as "*it is not believed that* l *is true*." Note that this does not imply that l is believed to be false. It is conceivable, and in fact is quite normal, for a rational reasoner to believe neither statement p nor its negation, $\neg p$. Clearly default negation not is different from classical $\neg$. Whereas $\neg p$ states that p is false, $not \ p$ is a statement about belief.

The disjunction or is also a new connective, sometimes called **epistemic disjunction**. The statement $l_1 \ or \ l_2$ is often read as "l_1 *is believed to be true or* l_2 *is believed to be true*." It is also different from the classical disjunction $\vee$. The statement $p \vee \neg p$ of propositional logic, called *the law of the exclusive middle*, is a tautology; however, the statement $p \ or \ \neg p$ is not. The former states that proposition p is either true or false, whereas the latter states that p is believed to be true or believed to be false. Since a rational reasoner can remain undecided about the truth or falsity of propositions, this is certainly not a tautology.

The left-hand side of an ASP rule is called the **head** and the right-hand side is called the **body**. Literals, possibly preceded by default negation not, are often called **extended literals**. The body of the rule can be viewed as a set of extended literals (sometimes referred to as the *premises* of the rule).

The head or the body can be empty. A rule with an empty head is often referred to as a **constraint** and written as

$$\leftarrow l_{i+1}, \ldots, l_m, \ not \ l_{m+1}, \ldots, \ not \ l_n.$$

A rule with an empty body is often referred to as a **fact** and written as

$$l_0 \ or \ \ldots \ or \ l_i.$$

Following the Prolog convention, non-numeric object, function and predicate constants of Σ are denoted by identifiers starting with lowercase letters; variables are identifiers starting with capital letters. Variables of Π range over ground terms of Σ. A rule r with variables is viewed as the set of its **ground instantiations** – rules obtained from r by replacing r's variables by ground terms of Σ and by evaluating arithmetic terms (e.g., replacing $2 + 3$ by 5). The set of ground instantiations of rules of Π is called the **grounding** of Π; program Π with variables can be viewed simply as a shorthand for its grounding. This means that it is enough to define the semantics of ground programs. For example, consider the program Π_1 with signature Σ where

$$\begin{aligned}\mathcal{O} &= \{a, b\}\\ \mathcal{F} &= \emptyset\\ \mathcal{P} &= \{p, q\}\\ \mathcal{V} &= \{X\}\end{aligned}$$

and rule

$$p(X) \leftarrow q(X).$$

Its rule can be converted to the two ground rules,

$$\begin{aligned}p(a) &\leftarrow q(a).\\ p(b) &\leftarrow q(b).\end{aligned}$$

which constitute the grounding of Π_1, denoted by $gr(\Pi_1)$.

To proceed we also need to define what it means for a set of ground literals to satisfy a rule. We first define the notion for the parts that make up the rule and then show how the parts combine to define the satisfiability of the rule.

Definition 2.1.1. *(Satisfiability) A set S of ground literals* **satisfies***:*

1. l **if** $l \in S$*;*
2. $not\ l$ **if** $l \notin S$*;*
3. $l_1 \ or \ \ldots \ or \ l_n$ **if** *for some* $1 \leq i \leq n$, $l_i \in S$;
4. *a set of ground extended literals* **if** *S satisfies every element of this set;*
5. *rule r* **if***, whenever S satisfies r's body, it satisfies r's head.*

For example, let r be the rule

$$p(a)\ or\ p(b) \leftarrow q(b), \neg t(c),\ not\ t(b).$$

and let S be the set

$$\{\neg p(a), q(b), \neg t(c)\}.$$

Let's check if S satisfies r. First we check if the body of the rule is satisfied by S. The body consists of three extended literals: $q(b)$, $\neg t(c)$, and $not\ t(b)$. The first two are satisfied by clause (1) of the definition, and the last is satisfied by clause (2). By clause (4), we have that the body is satisfied. Since the body is satisfied, to satisfy the rule, S must satisfy the head (clause (5)). It does not, because neither $p(a)$ nor $p(b)$ is in S (clause(3)). Therefore, S does *not* satisfy r. There are many sets that do satisfy r including $\emptyset$, $\{p(a)\}$, $\{p(b)\}$, $\{q(b), t(c)\}$, and $\{p(a), q(b), \neg t(c)\}$. For practice, check to see that this is so.

2.2 Semantics

First we introduce the semantics of ASP informally to give a feeling for the nature of the reasoning involved. We state the basic principles and give a number of examples. Then we formally define the semantics by stating what it means for a program to entail a ground literal.

2.2.1 Informal Semantics

Informally, program Π can be viewed as a specification for answer sets – sets of beliefs that could be held by a rational reasoner associated with Π. Answer sets are represented by collections of ground literals. In forming such sets the reasoner must be guided by the following informal principles:

1. Satisfy the rules of Π. In other words, believe in the head of a rule if you believe in its body.
2. Do not believe in contradictions.
3. Adhere to the "Rationality Principle" that says, "Believe nothing you are not forced to believe."

Let's look at some examples. Recall that in accordance with our assumption, the signatures of programs in these examples consist only of symbols used in their rules.

Example 2.2.1.

$p(b) \leftarrow q(a).$ "Believe $p(b)$ if you believe $q(a)$."
$q(a).$ "Believe $q(a)$."

Note that the second rule is a fact. Its body is empty. Clearly any set of literals satisfies an empty collection, and hence, according to our first principle, we must believe $q(a)$. The same principle applied to the first rule forces us to believe $p(b)$. The resulting set $S_1 = \{q(a), p(b)\}$ is consistent and satisfies the rules of the program. Moreover, we had to believe in each of its elements. Therefore, it is an answer set of our program. Now consider set $S_2 = \{q(a), p(b), q(b)\}$. It is consistent, satisfies the rules of the program, but contains the literal $q(b)$, which we were not forced to believe in by our rules. Therefore, S_2 is not an answer set of the program. You might have noticed that, because we did not have any choices in the construction of S_1, it is the *only* answer set of the program.

Example 2.2.2. *(Classical Negation)*

$\neg p(b) \leftarrow \neg q(a).$ "Believe that $p(b)$ is false if you believe that $q(a)$ is false."
$\neg q(a).$ "Believe that $q(a)$ is false."

There is no difference in reasoning about negative literals. In this case, the only answer set of the program is $\{\neg p(b), \neg q(a)\}$.

Example 2.2.3. *(Epistemic Disjunction)*

$p(a)$ *or* $p(b)$. "Believe $p(a)$ or believe $p(b)$."

There are three sets satisfying this rule – $\{p(a)\}$, $\{p(b)\}$, and $\{p(a), p(b)\}$. It is easy to see, however, that only the first two are answer sets of the program. According to our rationality principle, it would be irrational to adopt the third set as a possible set of beliefs – if we do we would clearly believe more than necessary.

Consider now the program

$$p(a) \text{ or } p(b).$$
$$q(a) \leftarrow p(a).$$
$$q(a) \leftarrow p(b).$$

Here the first rule gives us two choices – believe $p(a)$ or believe $p(b)$. The next two rules force us to believe $q(a)$ regardless of this choice. This is

a typical example of so-called *reasoning by cases*. The program has two answer sets: $\{p(a), q(a)\}$ and $\{p(b), q(a)\}$.

Another important thing to notice about epistemic disjunction is that it is different from exclusive *or*. (Recall that if the disjunction between A and B is understood as exclusive, then A is true or B is true but not both. A regular disjunction, which is satisfied if at least one of its disjuncts is true, is sometimes called inclusive.)

Consider the following program:

$$p(a) \text{ or } p(b).$$
$$p(a).$$
$$p(b).$$

The answer set of this program is $\{p(a), p(b)\}$. Note that if *or* were exclusive, the program would be contradictory.[5]

Of course the exclusive or of $p(a)$ and $p(b)$ can also be easily expressed in our language. This can be done by using two rules:

$$p(a) \text{ or } p(b).$$
$$\neg p(a) \text{ or } \neg p(b).$$

which naturally correspond to the definition of exclusive or. It is easy to check that, as expected, the program has two answer sets: $\{p(a), \neg p(b)\}$ and $\{\neg p(a), p(b)\}$.

Example 2.2.4. *(Constraints)*

$p(a) \text{ or } p(b).$	"Believe $p(a)$ or believe $p(b)$."
$\leftarrow p(a).$	"It is impossible to believe $p(a)$."

The first rule forces us to believe $p(a)$ or to believe $p(b)$. The second rule is a constraint that prohibits the reasoner's belief in $p(a)$. Therefore, the first possibility is eliminated, which leaves $\{p(b)\}$ as the only answer set of the program. In this example you can see that the constraint limits the sets of beliefs an agent can have, but does not serve to derive any new information. Later we show that this is always the case.

[5] One may ask why anyone would bother putting in the disjunction when $p(a)$ and $p(b)$ are known. Remember, however, that the facts could have been added (or learned) later. Also, in reality, the program may not be so straightforward. The facts may be derived from other rules and seemingly unrelated new information.

Example 2.2.5. *(Default Negation)*
Sometimes agents can make conclusions based on the absence of information. For example, an agent might assume that with the absence of evidence to the contrary, a class has not been canceled. Or, it might wish to assume that if a person does not know whether she is going to class the next day, then that day is not a holiday. Such reasoning is captured by default negation. Here are two examples.

$p(a) \leftarrow not\ q(a).$ "If $q(a)$ does not belong to your set of beliefs, then $p(a)$ must."

No rule of the program has $q(a)$ in its head, and hence, nothing forces the reasoner, which uses the program as its knowledge base, to believe $q(a)$. So, by the rationality principle, he does not. To satisfy the only rule of the program, the reasoner must believe $p(a)$; thus, $\{p(a)\}$ is the only answer set of the program.

Now consider the following program:

$p(a) \leftarrow not\ q(a).$ "If $q(a)$ does not belong to your set of beliefs, then $p(a)$ must."

$p(b) \leftarrow not\ q(b).$ "If $q(b)$ does not belong to your set of beliefs, then $p(b)$ must."

$q(a).$ "Believe $q(a)$."

Clearly, $q(a)$ must be believed (i.e., must belong to every answer set of the program). This means that the body of the first rule is never satisfied; therefore, the first rule does not contribute to our construction. Since there is no rule in the program whose head contains $q(b)$, we cannot be forced to believe $q(b)$; the body of the second rule is satisfied, and hence, $p(b)$ must be believed. Thus, the only answer set of this program is $\{q(a), p(b)\}$.

Given a definition of an answer set of a program, one can easily define the notion of entailment:

Definition 2.2.1. *(ASP Entailment)*
A program Π **entails** *a literal* l *(*$\Pi \models l$*) if* l *belongs to all answer sets of* Π.

Π entails a set of literals if it entails every literal in this set. Often instead of saying that Π entails l we say that l is a **consequence** of Π.

This seemingly simple notion is really very novel and deserves careful study. To see its novelty, let us recall that the entailment relation of classical logic that forms the basis for mathematical reasoning has the important

property called **monotonicity**: The addition of new axioms to a theory T of classical logic cannot decrease the set of consequences of T. More formally, entailment relation $\models$ is called monotonic if for every A, B, and C if $A \models B$ then $A, C \models B$. This property guarantees that a mathematical theorem, once proven, stays proven. This is not the case for ASP entailment. Addition of new information to program Π may invalidate the previous conclusion. In other words for a nonmonotonic entailment relation, $\models$, $A \models B$ does not guarantee that $A, C \models B$. Clearly, program Π_1 consisting of the rule

$$p(a) \leftarrow \ not\ q(a).$$

entails $p(a)$, whereas program

$$\Pi_2 = \Pi_1 \cup \{q(a)\}$$

does not. Addition of $q(a)$ to the agent's knowledge base invalidates the previous conclusion. It forces the agent to stop believing in $p(a)$.

This feature seems to be typical of our commonsense reasoning, where our conclusions are often tentative. This quality of commonsense reasoning made a number of AI researchers doubt that the logical approach to AI would succeed. The discovery of **nonmonotonic** logics in the 1980s dispelled these doubts. It is exactly this nonmonotonic quality given to ASP by its unique connectives and entailment relation that makes it such a powerful knowledge representation language. Much more is said later about nonmonotonicity in this book, but for now we return to our definitions.

We use the notion of entailment to answer queries to program Π. By a **query** we mean a conjunction or disjunction of literals. Queries not containing variables are called ground.

Definition 2.2.2. *(Answer to a Query)*

- *The answer to a ground conjunctive query, $l_1 \wedge \cdots \wedge l_n$, where $n \geq 1$, is*
  - yes *if* $\Pi \models \{l_1, \ldots, l_n\}$,
  - no *if there is i such that* $\Pi \models \bar{l}_i$,
  - unknown *otherwise.*
- *The answer to a ground disjunctive query, l_1 or $\ldots$ or l_n, where $n \geq 1$, is*
  - yes *if there is i such that* $\Pi \models l_i$,
  - no *if* $\Pi \models \{\bar{l}_1, \ldots, \bar{l}_n\}$,
  - unknown *otherwise.*

- *An answer to a query $q(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ is the list of variables occurring in q, is a sequence of ground terms $t_1, \ldots, t_n$ such that $\Pi \models q(t_1, \ldots, t_n)$.*

(Note that the actual reasoning system we are going to use to answer queries refers to them as *epistemic queries*, uses a comma instead of $\wedge$ for conjunction, and does not directly support disjunctive queries.)

Example 2.2.6.
Consider again program Π_1 consisting of a rule

$$p(a) \leftarrow \ not\ q(a).$$

It has the answer set $\{p(a)\}$ and thus answers *yes* and *unknown* to queries $?p(a)$ and $?q(a)$, respectively. Query $?(p(a) \wedge q(a))$ is answered by *unknown*; $?(p(a)\ or\ q(a))$ is answered by *yes*. Query $?p(X)$ has exactly one answer: $X = a$. Let's add one more rule to this program:

$$\neg q(X) \leftarrow \ not\ q(X). \quad \text{"If } q(X) \text{ is not believed to be true, believe that it is false."}$$

This rule is known as the **Closed World Assumption (CWA)**. It guarantees that answer sets of a program are complete with respect to the given predicate (i.e., every answer set must contain either $q(t)$ or $\neg q(t)$ for every ground term t from the signature of the program). The new program's answer set is $\{p(a), \neg q(a)\}$. This time queries $?p(a)$ and $?(p(a)\ or\ q(a))$ are still answered by *yes*, whereas the answers to queries $?q(a)$ and $?(p(a) \wedge q(a))$ change to *no*.

2.2.2 Formal Semantics

We first refine the notion of consistency of a set of literals. Pairs of literals of the form $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called *contrary*. A set S of ground literals is called *consistent* if it contains no contrary literals.

All that is left is to precisely define the notion of an answer set. The definition consists of two parts. The first part of the definition is for programs without default negation. The second part explains how to remove default negation so that the first part of the definition can be applied.

Definition 2.2.3. *(Answer Sets, Part I)*
Let Π be a program not containing default negation (i.e., consisting of rules of the form)

$$l_0\ or\ \ldots\ or\ l_i \leftarrow l_{i+1}, \ldots, l_m.$$

An **answer set** *of* Π *is a consistent set* S *of ground literals such that*

- S *satisfies the rules of* Π *and*
- S *is minimal (i.e., there is no proper subset of* S *that satisfies the rules of* Π*).*

Let's look at some examples, this time employing the formal definition and *observe that it captures our intuition.*

Example 2.2.7. *(Example 2.2.1, Revisited)*
Let us now go back to Example 2.2.1 and check that our informal argument is compatible with Definition 2.2.3, namely that $S_1 = \{q(a), p(b)\}$ is the answer set of program Π_1:

$$p(b) \leftarrow q(a).$$
$$q(a).$$

S_1 satisfies fact $q(a)$ by clause (1) of the definition of satisfiability (Definition 2.1.1). Clause (4) of this definition guarantees that the (empty) body of the second rule is vacuously satisfied by S_1, and hence, by clause (5), the second rule is satisfied by S_1. Since the head $p(b)$ of the first rule is in S_1, S_1 also satisfies the first rule. Clearly S_1 is consistent and no proper subset of S_1 satisfies the rules of Π_1; therefore, S_1 is Π_1's answer set. Suppose now that S is an answer set of Π_1. It must satisfy the rules of Π_1, and hence, S contains S_1. From minimality we can conclude that $S = S_1$ (i.e., S_1 is the unique answer set of Π_1). Later we show that any program with neither or nor not has at most one answer set.

Now that we have the answer set, we can see that, by the definition of entailment, this program entails $q(a)$ and $p(b)$. The following table shows the answers to some possible ground queries created from the program's signature:

$?\ q(a)$	yes
$?\ \neg q(a)$	no
$?\ p(b)$	yes
$?\ \neg p(b)$	no

We skip Example 2.2.2 (the reader is encouraged to work it out as an exercise) and consider another example.

Example 2.2.8.
Consider a program consisting of two rules:

$$p(a) \leftarrow p(b).$$
$$\neg p(a).$$

Let $S_1 = \{\neg p(a)\}$. Clearly, S_1 is consistent. Since $p(b) \notin S_1$, by clause (1) of the definition of satisfiability, the body of the first rule is not satisfied by S_1. Hence, by clause (5), S_1 satisfies the first rule. The body of the second rule is empty. Hence, by clause (4), it is vacuously satisfied by S_1. Thus, by clause (5), S_1 satisfies the second rule of the program. The only proper subset, $\emptyset$, of S_1 does not satisfy the second rule; therefore, S_1 is an answer set of our program. To show that S_1 is the only answer set, consider an arbitrary answer set S. Clearly, S must contain $\neg p(a)$. This means that by the minimality requirement $S = S_1$.

By the definition of entailment we can see that this program entails $\neg p(a)$. The following table contains the answers to some possible ground queries created from the program's signature:

$?\ p(a)$	no
$?\ \neg p(a)$	yes
$?\ p(b)$	$unknown$
$?\ \neg p(b)$	$unknown$

It may be worth noticing that this example clearly illustrates the difference between logic programming connective $\leftarrow$ and classical implication (denoted by $\supset$). Recall that $p(b) \supset p(a)$ is classically equivalent to its *contrapositive*, $\neg p(a) \supset \neg p(b)$. Hence, the classical theory consisting of $p(b) \supset p(a)$ and $\neg p(a)$ entails $\neg p(b)$. Our example shows that this is not the case if $\supset$ is replaced by $\leftarrow$. If we want the contrapositive of $p(a) \leftarrow p(b)$, we need to explicitly add it to the program.[6] We suggest that the reader check that, as expected, the resulting program would have the answer set $\{\neg p(a), \neg p(b)\}$.

Example 2.2.9.

$$p(b) \leftarrow \neg p(a).$$
$$\neg p(a).$$

[6] Later chapters show why we do not want a built-in contrapositive for our $\leftarrow$ connective.

Let $S_1 = \{\neg p(a), p(b)\}$. S_1 is a consistent set of ground literals that clearly satisfies the rules of the program. To show that S_1 is an answer set we need to show that no proper subset of S_1 satisfies the program rules. To see that, it is enough to notice that $\neg p(a)$ must belong to every answer set of the program by clause (1) of the definition of satisfiability, and $p(b)$ is required by clause (5). Therefore, S_1 is minimal.

This time, the answer to query $?p(b)$ is *yes* and to $?\neg p(b)$ is *no*.

Example 2.2.10. *(Empty Answer Set)*

$$p(b) \leftarrow \neg p(a).$$

There are no facts, so we are not forced to include anything in S_1. Let's check if $S_1 = \emptyset$ is an answer set. S_1 satisfies the rule because it does not satisfy its body. Since $\emptyset$ has no proper subsets, it is the only answer set of the program. Note that an empty answer set is by no means the same as the absence of one.

Since neither $p(a) \in \emptyset$ nor $\neg p(a) \in \emptyset$, the truth of $p(a)$ is *unknown*; the same applies for $p(b)$.

Now let's look at several programs containing epistemic disjunction.

Example 2.2.11. *(Epistemic Disjunction, Revisited)*
We start with the first two programs from Example 2.2.3 from the previous section. Program

$$p(a) \ or \ p(b).$$

has two answer sets, $\{p(a)\}$ and $\{p(b)\}$. Each contains a literal required by clause (3) of the definition of satisfiability. Note that $\{p(a), p(b)\}$ is not minimal, so it is not an answer set. Since entailment requires literals to be true in *all* answer sets of a program, this program does not entail $p(a)$ nor $p(b)$ (i.e., their truth values, along with the truth values of their negative counterparts, are *unknown*).

Now consider program

$$p(a) \ or \ p(b).$$
$$q(a) \leftarrow p(a).$$
$$q(a) \leftarrow p(b).$$

In this case our definition clearly gives two answer sets: $\{p(a), q(a)\}$ and $\{p(b), q(a)\}$. This program entails $q(a)$. It entails no other literals (but does entail, say, the disjunction $p(a) \ or \ p(b)$).

Now let's look at a new example emphasizing the difference between epistemic and classical readings of disjunction.

Example 2.2.12. *($p(a)$ or $\neg p(a)$ Is Not a Tautology)*
Consider

$$p(b) \leftarrow \neg p(a).$$
$$p(b) \leftarrow p(a).$$
$$p(a) \text{ or } \neg p(a).$$

The program has two answer sets: $S_1 = \{p(a), p(b)\}$ and $S_2 = \{\neg p(a), p(b)\}$. S_1 satisfies the first rule because it does not satisfy the rule's body, the second rule because it contains $p(a)$ and $p(b)$, and the third rule because it contains $p(a)$. S_2 satisfies the first rule because it contains both $\neg p(a)$ and $p(b)$, the second because it does not satisfy the body, and the third because it contains $\neg p(a)$. Because each literal is required to satisfy some rule, the sets are minimal.

But now let us look at the program

$$p(b) \leftarrow \neg p(a).$$
$$p(b) \leftarrow p(a).$$

It is not difficult to check that $\emptyset$ is the only answer set of this program. This result is not surprising since, as we mentioned in Section 2.1 when we discussed syntax, $p(a)$ *or* $\neg p(a)$ is not a tautology. Without the presence of explicit disjunction, $p(a)$ *or* $\neg p(a)$, the reasoner remains undecided about the truth value of $p(a)$, neither believing that $p(a)$ is true nor that it is false. Thus $p(b)$ is not included in the answer set.

It may be strange that we cannot conclude $p(b)$. Indeed one may be tempted to reason as follows: Either $p(a)$ is true or $p(a)$ is false, and hence, $p(b)$ must be included in any answer set. But, of course, this reasoning is based on the wrong reading of epistemic *or* – it slides back to a classical reading of the disjunction. Since the reasoner may have no opinion on the truth values of $p(a)$, the argument fails.

Example 2.2.13. *(Constraints, Revisited)*

$$p(a) \text{ or } p(b).$$
$$\leftarrow p(a).$$

The first rule is (minimally) satisfied by either $S_1 = \{p(a)\}$ or $S_2 = \{p(b)\}$. However, S_1 does not satisfy the second rule because it is not possible to satisfy an empty head if the body is satisfied. Therefore, S_2 is the only

answer set of the program. Note that, although we cannot include $p(a)$ in any answer set, we do not have enough information to entail $\neg p(a)$. The answers to queries $p(a)$ and $p(b)$ are *unknown* and *yes*, respectively.

So far we have not had to address default negation. The second part of the definition of answer sets addresses this question.

Definition 2.2.4. *(Answer Sets, Part II)*
Let Π *be an arbitrary program and* S *be a set of ground literals. By* Π^S *we denote the program obtained from* Π *by*

1. *removing all rules containing* $not\ l$ *such that* $l \in S$*;*
2. *removing all other premises containing* not*.*

S *is an answer set of* Π *if* S *is an answer set of* Π^S*.*

We refer to Π^S as the **reduct** of Π with respect to S.

Example 2.2.14. *(Default Negation, Revisited)*
Consider a program Π from Example 2.2.5 (see the following table). Let's confirm that $S = \{q(a), p(b)\}$ is the answer set of Π. Π has default negation; therefore, we use Part II of our definition of answer sets to compute Π^S.

	Π	Π^S
r_1	$p(a) \leftarrow\ not\ q(a).$	$(deleted)$
r_2	$p(b) \leftarrow\ not\ q(b).$	$p(b).$
r_3	$q(a).$	$q(a).$

1. We remove r_1 from Π because it has $not\ q(a)$ in its premise, whereas $q(a) \in S$.
2. Then, we remove the premise of r_2.

In this way, we eliminate all occurrences of default negation from Π. Clearly, S is the answer set of Π^S and, hence, of Π.

Note that, as mentioned in the definition, a reduct is always computed with respect to a candidate set of ground literals S. The algorithm for computing answer sets is not presented until Chapter 7; thus, we must still come up with candidate sets based on our intuition given by the informal semantics. (Of course, theoretically, we could test all possible sets of ground literals from the signature, but in practice this approach works only on small programs.) Meanwhile, the following proposition will be of some help for reasoning about answer sets.

Proposition 2.2.1. *Let S be an answer set of a ground ASP program Π.*

(a) S satisfies every rule $r \in \Pi$.
(b) If literal $l \in S$ then there is a rule r from Π such that the body of r is satisfied by S and l is the only literal in the head of r satisfied by S. (It is often said that **rule** r **supports literal** l.*)*

The first part of the proposition guarantees that answer sets of a program satisfy its rules; the second guarantees that every element of an answer set of a program is supported by at least one of its rules.

Here are some more examples of answer sets of programs with default negation:

Example 2.2.15.

$$p(a) \leftarrow \ not\ p(a).$$

This program has no answer set. This result can be established by simply considering two available candidates, $S_1 = \emptyset$ and $S_2 = \{p(a)\}$. S_1 does not satisfy the program's single rule, and hence, according to the first clause of Proposotion 2.2.1, cannot be the program's answer set. The second clause of the proposition allows us to see that S_2 cannot be an answer set because $p(a)$ is not supported by any rule of the program.

Note that the absence of answer sets is not surprising. The rule, which tells the agent to believe $p(a)$ if it does not believe it, should naturally be rejected by a rational agent.

This is our first example of an ASP program without answer sets. We refer to such programs as **inconsistent**. There are, of course, many other inconsistent programs, such as

$$p(a).$$
$$\neg p(a).$$

or

$$p(a).$$
$$\leftarrow p(a).$$

but inconsistency normally appears only when the program is erroneous (i.e., it does not adequately represents the agent's knowledge). Later, however, we show how various interesting reasoning tasks can be reduced to discovering the inconsistency of a program.

Example 2.2.16.

$$p(a) \leftarrow \ not\ p(a).$$
$$p(a).$$

There are two candidate answer sets: $S_1 = \{p(a)\}$ and $S_2 = \emptyset$. Since the reduct of the program with respect to S_1 is $p(a)$, S_1 is an answer set. Clearly, S_2 does not satisfy the rules of the program; thus, S_1 is the only answer set.

Example 2.2.17.

$$p(a) \leftarrow \ not\ p(b).$$
$$p(b) \leftarrow \ not\ p(a).$$

This program has two answer sets: $\{p(a)\}$ and $\{p(b)\}$. Note that $\emptyset$ is not an answer set for this program because it does not satisfy the program's rules. The set $\{p(a), p(b)\}$ is not an answer set of the program since its elements are not supported by the rules of the program.

Example 2.2.18.

$$p(a) \leftarrow \ not\ p(b).$$
$$p(b) \leftarrow \ not\ p(a).$$
$$\leftarrow p(b).$$

The constraint eliminates $\{p(b)\}$, making $\{p(a)\}$ the only answer set of the program.

Example 2.2.19.

$$p(a) \leftarrow \ not\ p(b).$$
$$p(b) \leftarrow \ not\ p(a).$$
$$\leftarrow p(b).$$
$$\neg p(a).$$

This program has no answer set. Indeed by Proposition 2.2.1 an answer set must contain $\neg p(a)$ and cannot contain $p(b)$. Hence, by the first rule of the program it should contain $p(a)$, which is impossible.

In the next example we consider a slightly more complex program.

Example 2.2.20. Let Π consist of the rules

$$s(b).$$
$$r(a).$$
$$p(a)\ or\ p(b).$$
$$q(X) \leftarrow p(X), r(X),\ not\ s(X).$$

After grounding the program will have the form

$$s(b).$$
$$r(a).$$
$$p(a)\ or\ p(b).$$
$$q(a) \leftarrow p(a), r(a),\ not\ s(a).$$
$$q(b) \leftarrow p(b), r(b),\ not\ s(b).$$

Even though the number of candidate answer sets for this program is large, the answer sets of the program can still be obtained in a relatively simple way. It is easy to see that by Proposition 2.2.1 an answer set S of the program must satisfy the first three rules. Thus, S contains $\{s(b), r(a), p(a)\}$ or $\{s(b), r(a), p(b)\}$. The program has no rule that can possibly support $s(a)$, and hence, every answer set of the program satisfies $not\ s(a)$. This means that every answer set containing $\{p(a), r(a)\}$ must also contain $q(a)$. There is no rule that can support $r(b)$; thus, the body of the last rule cannot be satisfied and $q(b)$ cannot belong to any answer set of the program. This implies that the program may only have two answer sets: $\{s(b), r(a), p(a), q(a)\}$ and $\{s(b), r(a), p(b)\}$. Both of the candidates are indeed answer sets, which can be easily checked using the definition.

So far our computation of answer sets has been done by hand. There are a large number of efficient software systems, called **ASP solvers**, that automate this process. Appendix A contains instructions for downloading and using several such solvers. At this point we advise the readers to download an answer set solver and use it to compute answer sets for programs from the previous examples. For many of the programs in this book queries can be answered manually by examining these answer sets and applying Definition 2.2.2. STUDENT from Chapter 1 is implemented by a query-answering system based on ASP solvers as described in Appendix B.

The version of ASP described in this book and used by most of the existing ASP solvers is unsorted. Readers who, as do the authors of this book, prefer to program in a sorted language should consult Appendix C

describing a sorted version of ASP, its solver, and query-answering system that implement STUDENT.

2.3 A Note on Translation from Natural Language

So far we have discussed the semantics of logical connectives of ASP. The question of translating fragments of natural language into ASP theories was addressed only briefly.[7] It is important to realize that, as in any translation, the translation from even simple English sentences to statements in ASP may be a nontrivial task. Consider the simple English sentence, "All professors are adults." The direct (literal) translation of this sentence seems to be

$$(1)\ adult(X) \leftarrow prof(X).$$

Used in conjunction with a list of professors, it will allow us to make conclusions about their adulthood; e.g., the program consisting of that rule and the fact $prof(john)$ entails $adult(john)$. But what happens if we expand this program by $\neg adult(alice)$? Intuitively we should be able to conclude that $\neg prof(alice)$. The literal translation of the English statement does not allow us to do that. This happens because some information implicitly present in the English text is missing from our translation. One can argue, for instance, that the translation is missing something akin to classical logic's law of the exclusive middle with respect to professors and adults; thus a good translation to ASP should include statements

$$(2)\ adult(X)\ or\ \neg adult(X).$$
$$(3)\ prof(X)\ or\ \neg prof(X).$$

where X ranges over some sort $person$. For simplicity we assume that there are only two persons, $john$ and $alice$. Let us denote the program consisting of the rules (1)–(3) and the sort $person$ by Π_1. One can easily check that Π_1 combined with the two facts above has one answer set:

$$\{prof(john), adult(john), \neg prof(alice), \neg adult(alice)\}.$$

(We are not showing the sort $person$.) As is typical in natural-language translation, this is not the only reasonable representation of our statement

[7] Here we, of course, are only talking about *manual* translation. We do not discuss methods for *automatically* translating from English to a formal language; however, this task, known as *natural-language processing*, is also a very important part of the general problem of artificial intelligence.

in ASP. One can simply translate our English statement into program Π_2 consisting of the above sort $person$ and two rules:

$$adult(X) \leftarrow prof(X).$$
$$\neg prof(X) \leftarrow \neg adult(X).$$

The first rule is a direct translation and the second is its contrapositive. Again the program, used together with $prof(john)$ and $\neg adult(alice)$, produces the expected answers. Note, however, that programs Π_1 and Π_2 are not equivalent (i.e., they do not have the same answer sets). Indeed, if we do not include the sort, $\emptyset$ is the only answer set of Π_2, whereas Π_1 has answer sets including

$$\{prof(alice), adult(alice), prof(john), adult(john)\}$$
$$\{prof(alice), adult(alice), \neg prof(john), \neg adult(john)\}$$
$$\{\neg prof(alice), \neg adult(alice), prof(john), adult(john)\}$$
$$\{\neg prof(alice), \neg adult(alice), \neg prof(john), \neg adult(john)\}.$$

Which representation is better may depend on the context of the sentence, the purpose of our representation, and simply the taste of the translator. Any type of translation is an art, and translation from English into ASP is no exception. One of the goals of this book is to help you to become better translators and to better understand the consequences of your translation decisions.

2.4 Properties of ASP Programs

In this section we give several useful properties of ASP programs. (In the rest of this book, unless otherwise stated, we use the terms "ASP program" and "logic program" interchangeably.) We start with conditions that guarantee program consistency.

In what follows we consider programs consisting of rules of the form

$$p_0 \ or \ \ldots \ or \ p_i \leftarrow p_{i+1}, \ldots, p_m, \ not \ p_{m+1}, \ldots, \ not \ p_n \qquad (2.2)$$

where ps are atoms and $i \geq 0$ (in other words, programs containing no classical negation $\neg$ and no constraints).

Definition 2.4.1. *(Level Mapping)*
Let program Π *consist of rules of form (2.2). A function* $||\ ||$ *from ground atoms of* Π *to natural numbers*[8] *is called a* **level mapping** *of* Π. *Level*

[8] For simplicity we consider a special case of the more general original definition that allows arbitrary countable ordinals.

$||D||$, *where* D *is a disjunction of atoms, is defined as the* minimum level of D's members.

Definition 2.4.2. *(Local Stratification)*
A program Π *consisting of rules of form (2.2) is called* **locally stratified** *if there is a level mapping* $||\ ||$ *of* Π *such that for every rule* $r \in \Pi$*, the following is true:*

1. *for every* p_k *where* $i < k \leq m$, $||p_k|| \leq ||head(r)||$*; and*
2. *for every* p_k *where* $m < k \leq n$, $||p_k|| < ||head(r)||$.

It is easy to see that any program without classical and default negation is locally stratified. A function mapping all atoms of such a program into, say, 0 is a level mapping that satisfies the corresponding conditions. A function $||p(a)|| = ||r(a)|| = 1$ and $||q(a)|| = 0$ is a level mapping of a program consisting of rule

$$p(a) \leftarrow \ not\ q(a), r(a).$$

Clearly the mapping satisfies the conditions from the definition, and hence, the program is locally stratified.

Proposition 2.4.1. *(Properties of Locally Stratified Programs)*

- *A locally stratified program is consistent.*
- *A locally stratified program without disjunction has exactly one answer set.*
- *The above conditions hold for the union of a locally stratified program and any collection of closed world assumptions; i.e. rules of the form*

$$\neg p(X) \leftarrow \ not\ p(X).$$

Proposition 2.4.1 immediately implies the existence and uniqueness of answer sets of programs from Examples 2.2.5 and 2.2.6. It is, however, not applicable to the program from Example 2.2.15.

Let's consider two more examples of locally stratified programs.

Example 2.4.1.
Consider a program

$$p(0).$$
$$p(f(I)) \leftarrow p(I).$$

It is not difficult to see that the minimal set of ground literals satisfying rules of this program is an infinite set:

$$S = \{p(0), p(f(0)), p(f(f(0))), \dots\}$$

The program contains no negations and, hence, is locally stratified. Therefore, it has no answer sets except S.

The next example is only slightly more sophisticated.

Example 2.4.2.
Consider the program Π

$$p(0).$$
$$p(f(I)) \leftarrow \ not\ p(I).$$

and a set

$$S = \{p(0), p(f(f(I))), p(f(f(f(f(I))))), \dots\}.$$

Using the definition one can easily prove that S is an answer set of Π. Clearly, Π is locally stratified. (Consider a level mapping assigning to each atom $p(t)$ the number of fs occurring in t and check that it satisfies the corresponding conditions.) Therefore, S is the only answer set of the program.

Another approach to proving consistency and, sometimes, uniqueness of a logic program is based on the notion of the program's dependency graph. Let Π be a (not-necessarily) ground program consisting of rules of form (2.2). A **dependency graph** of Π is a collection of nodes labeled by predicate symbols from the signature of Π and a collection of arcs of the form $\langle p_1, p_2, s\rangle$ where s is $+$ (positive link) or $-$ (negative link). The graph contains $\langle p_1, p_2, +\rangle$ if there is a rule of Π containing an atom formed by p_1 in the head and an atom formed by p_2 in the body; it contains $\langle p_1, p_2, -\rangle$ if there is a rule of Π containing an atom formed by p_1 in the head and an extended literal of the form $not\ l$ where l is formed by p_2 in the body. Note that two nodes can be connected by both positive and negative links. A cycle in the graph is said to be negative if it contains at least one edge labeled by $-$. A program is called **stratified** if its dependency graph contains no negative cycles.

Proposition 2.4.2. *Let Π be a program consisting of rules of form (2.2).*

- *If the dependency graph of Π contains no cycles with an odd number of negative links then Π is consistent (i.e., has an answer set).*
- *A stratified program without disjunction has exactly one answer set.*

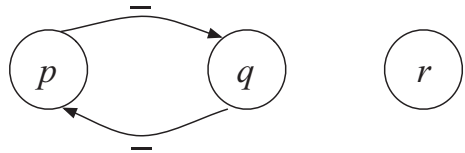

Figure 2.1. Dependency Graph.

The proposition can be used to prove consistency of programs

$$p(X) \leftarrow not\ q(X).$$
$$q(X) \leftarrow not\ p(X).$$
$$r(0).$$

and

$$p(X) \leftarrow not\ q(X).$$
$$r(0).$$

The first program has a dependency graph with an even number of negative cycles – see Figure 2.1. The second is stratified, thus having exactly one answer set. Despite substantial differences in their definitions, there is a close relationship between stratified and locally stratified programs. In fact, every stratified program is locally stratified.

The following proposition is useful for understanding the role of constraints.

Proposition 2.4.3. *Let Π be a logic program consisting of a collection R of rules with nonempty heads and a collection C of constraints. Then S is an answer set of Π iff S is an answer set of R that satisfies constraints from C.*

Last, we describe a procedure for removing negative literals and constraints from a program without changing its answer sets. For every predicate symbol p from the signature of Π,

1. Introduce a new predicate symbol p^+.
2. Replace every occurrence of a negative literal of the form $\neg p(\bar{t})$ in the program by a new, positive literal $p^+(\bar{t})$. We call this literal the *positive form* of $\neg p(\bar{t})$.
3. For every sequence of ground terms $\bar{t}$ that can serve as a parameter of p, expand Π by the axiom

$$\leftarrow p(\bar{t}), p^+(\bar{t})$$

4. Replace every constraint

$$\leftarrow body$$

of the program by rule

$$p \leftarrow body,\ not\ p$$

where p is a new atom.

It is not difficult to show that the new program Π^+, which contains neither classical negation nor constraints, has the following property:

Proposition 2.4.4. *Let S be a set of literals over signature Σ of Π. By S^+ we denote the set obtained by replacing negative literals of S by their positive forms. A consistent set S of literals from Σ is an answer set of Π iff S^+ is an answer set of Π^+.*

Summary

In this chapter we introduced our main knowledge representation tool – Answer Set Prolog. The logical connectives and the semantics of the language capture the intuition behind a particular notion of beliefs of rational agents as described by the rationality principle.

We hope that the reader agrees that the language satisfies the following important principles of good design: It has a simple syntax, its logical connectives have reasonably clear intuitive meaning, and the definition of its semantics is mathematically simple and transparent. The entailment relation of the language is nonmonotonic, which makes ASP dramatically different from the language of classical logic. In the following chapters we show the importance of this property for knowledge representation.

The chapter continued with a brief discussion of the subtleties of the translation of natural-language texts into Answer Set Prolog – a theme that permeates this book.

It concluded by describing several simple but important mathematical properties of the language. The first two give sufficient conditions guaranteeing the existence and uniqueness of answer sets of the programs. The last one illustrates the role of constraints as the means of limiting an agent's beliefs without generating new ones.

References and Further Reading

The syntax and semantics of Answer Set Prolog were introduced in Gelfond and Lifschitz (1991). The notion of stratification and its relation

to dependency graphs was introduced in Apt, Blair, and Walker (1988). Local stratification first appeared in Przymusinski (1988). The properties of stratified and locally stratified programs were proven in these papers. There are many interesting extensions to these notions (see for instance Przymusinska and Przymusinski [1990]). Clause 1 of Proposition 2.4.2 is a simple special case of a more general result from Fages (1994). There are a substantial number of generalizations of Answer Set Prolog that allow rules whose heads and bodies contain more-complex formulas; see, for instance, Ferraris, Lee, and Lifschitz (2007) and Pearce (1997, 2006). Some of the other generalizations of the language are described in later chapters. The closed world assumption and its importance for knowledge representation and reasoning were first discussed in Reiter (1978). (Representation of CWA in Answer Set Prolog is from Gelfond and Lifschitz (1991)). The first efficient ASP system computing answer sets of nondisjunctive logic programs is described in Niemela, Simons, and Soininen (2002) and Niemela and Simons (1997). For a description of the corresponding system for programs with disjunction, see Leone, Rullo, and Scarcello (1997) and Calimeri et al. (2002). An interesting overview can be found in Minker and Seipal (2002). For information on natural-language processing see, for instance, Blackburn and Bos (2005).

Exercises

1. Given the following signature

$$\mathcal{O} = \{a\}$$
$$\mathcal{F} = \{f\}$$
$$\mathcal{P} = \{p\}$$
$$\mathcal{V} = \{X\}$$

specify which of the following are terms, atoms, literals, or none.

(a) a
(b) $\neg a$
(c) X
(d) f
(e) $f(a)$
(f) $\neg f(a)$
(g) $f(p)$
(h) $f(X)$
(i) p
(j) $p(a)$
(k) $\neg p(a)$
(l) $p(f(a))$
(m) $p(\neg f(a))$
(n) $\neg p(f(a))$
(o) $p(X)$

2. Given the following sorted signature

$$\mathcal{O} = \{pele, namath, jordan, soccer, football, basketball\}$$
$$Sorts = \{player, sport\}$$
$$player = \{pele, namath, jordan\}$$
$$sport = \{soccer, football, basketball\}$$
$$\mathcal{F} = \emptyset$$
$$\mathcal{P} = \{plays(player, sport)\}$$
$$\mathcal{V} = \{X, Y\}$$

   (a) Is $\neg plays(pele, soccer)$ a literal?
   (b) Is $plays(football, namath)$ a literal?
   (c) Is $basketball$ a term?

3. Given program Π,

$$p(a) \leftarrow \ not\ p(b).$$
$$p(b) \leftarrow \ not\ p(c).$$
$$p(c) \leftarrow \ not\ p(a).$$

   and set $S = \{p(c)\}$

   (a) Construct the program Π^S from the definition of an answer set.
   (b) Check if S is an answer set of Π. Justify your answer.

4. (a) Compute the answer sets of the following program:

$$p\ or\ q\ or\ r.$$
$$\neg p \leftarrow \ not\ s.$$

   (b) How does the above program answer queries $?p$ and $?q$?

5. Compute the answer sets of the following program:

$$p(a) \leftarrow \ not\ p(b), \neg p(c).$$
$$p(b) \leftarrow \ not\ p(a), \neg p(c).$$
$$\neg p(X) \leftarrow \ not\ p(X).$$

6. Compute the answer sets of the following program:

$$p \leftarrow \ not\ q.$$
$$q \leftarrow \ not\ p.$$
$$r \leftarrow \ not\ s.$$
$$s \leftarrow \ not\ r.$$
$$\neg s \leftarrow q.$$

7. (a) Compute the answer sets of the following program. *Assume that a and b are the object constants of this program's signature.*

$$\neg s(a).$$
$$p(X) \leftarrow \ not\ q(X), \neg s(X).$$
$$q(X) \leftarrow \ not\ p(X).$$
$$r(X) \leftarrow p(X).$$
$$r(X) \leftarrow q(X).$$

   (b) How does the program answer queries $?s(a),\ ?r(a),\ ?s(b)$, and $?q(b)$?

8. (a) Compute the answer sets of the following program:

$$p(a)\ or\ \neg p(b).$$
$$q(X) \leftarrow \neg p(X).$$
$$\neg q(X) \leftarrow \ not\ q(X).$$
$$r(X) \leftarrow \ not\ p(X).$$

   (b) How does the program answer queries $?q(a),\ ?r(a),\ ?q(b)$, and $?r(b)$?

9. (a) Compute the answer sets of the following program:

$$p(X)\ or\ q(X) \leftarrow \ not\ r(X).$$
$$\neg p(X) \leftarrow h(X),\ not\ r(X).$$
$$h(a).$$
$$h(b).$$
$$r(a).$$

   (b) How does the program answer queries $?p(b),\ ?q(b)$, and $?r(b)$?

10. (a) Compute the answer sets of the following program:

$$p(a) \leftarrow \ not\ p(b).$$
$$p(b) \leftarrow \ not\ p(a).$$
$$q(a).$$
$$\neg q(b) \leftarrow p(X),\ not\ r(X).$$

    (b) How does the program answer queries $?q(a),\ ?q(b),\ ?p(a)$, and $?r(b)$?

11. Translate the following story about dealing with high prices into ASP. Ignore the time factor. "Either the supply was increased or price controls were instituted. Instituting price controls leads to shortages. There are no shortages." Use zero-arity predicate symbols $increased\_supply$, $price\_controls$ and $shortages$. Compute the answer sets of this program. How does your program answer these queries:

$$?\ increased\_supply$$
$$?\ increased\_supply \wedge \neg price\_controls$$

    Is that what you intended? (Note that the answer depends on your understanding of the disjunction in the first sentence of the story. Is it inclusive or exclusive?)

12. Consider the following story. "If Jim does not buy toys for his children, Jim's children will not receive toys for Christmas. If Jim's children do not write their Christmas letters, Jim will not buy them toys. Jim's children do receive toys for Christmas." Assume that the intended interpretation of this story implies that Jim's children wrote their Christmas letters.
    (a) Translate the story into an ASP program and compute the answer set. Use disjunction to encode the law of the exclusive middle to allow the program to come to the proper conclusion.
    (b) Translate the story into an ASP program and compute the answer set, this time making the contrapositive explicit for each statement.