Table of Contents I

Intro to the Logic-Based Approach to AI
- Language Choice
- Simple Knowledge Base Example
- History: A Part of the Big Picture

Motivation for ASP

Syntax and Semantics of ASP
- Syntax
- Informal Semantics
- Formal Semantics
- Translating from Natural Language
- Properties of ASP Programs

Reading

- Read Chapter 1 and Chapter 2 in *Knowledge Representation, Reasoning and the Design of Intelligent Agents* by Gelfond and Kahl. (Section 2.4 is optional.)

Logic-Based Approach to AI: Language Choice

- algorithmic — describe sequences of actions for a computer to perform
- declarative — describe properties of objects and relations between them
- logic-based approach to AI proposes to:
  - use a declarative language to describe the domain
  - express various tasks (like planning or explanations of unexpected observations) as queries to the resulting program
  - use an inference engine (a collection of reasoning algorithms) to answer these queries

What Does a Declarative Program Look Like?

$$
\begin{aligned}
&father(john, sam).\\
&mother(alice, sam).\\
&gender(john, male).\\
&gender(sam, male).\\
&gender(alice, female).\\
\\
&parent(X, Y) \leftarrow father(X, Y).\\
&parent(X, Y) \leftarrow mother(X, Y).\\
\\
&child(X, Y) \leftarrow parent(Y, X).
\end{aligned}
$$

This program is written in a variant of Answer Set Prolog.
Replace $\leftarrow$ by :- and you have an executable program.

What Does the Program Know?

Feed the program to an inference engine and ask it questions (queries): Is Sam the child of John? Who are Sam's parents?

$$? \; child(sam, john).$$
$$? \; parent(X, sam).$$

Note: This is similar to how we check to see what a human knows.

Does it know that Sam is John's son?

Typical Features of Logic-Based Programming

- Knowledge is represented in precise mathematical language.
- Search problems are expressed as queries.
- An inference engine is used to answer queries.
- The program is *elaboration tolerant*.

Elaboration Tolerance

- A program is *elaboration tolerant* if small changes in specifications do not cause global program changes.
- We added knowledge to accommodate the concepts of parent and child without having to change any of the original code.
- Let's add $padre(jose, maria)$ and teach our program that padre means father.
- Note that we do not have to change the original program.

Looking Back to Whence We Came

- Euclid in his *Elements* derives a huge body of geometric knowledge from five basic axioms.
- Early 20th century mathematicians develop the notion of a *formal language* and apply it to axiomatize set theory.
- Number, function, and shape are defined in terms of sets and their membership relations.
- (Almost) all of the mathematical knowledge of the early 20th century could be viewed as logical consequences of a collection of axioms that could fit on a medium-sized blackboard. This achievement demonstrated the high degree of development of logic.
- Also from logic — the notion of correct mathematical argument, proof.

The Leibniz Dream

- Gottfried Leibniz tried to apply this *axiomatic method* to the science of reasoning.
- Now known as the *Leibniz Dream*, the idea that we could axiomatize more than mathematics has inspired many computer scientist, including Edsger Dijkstra and John McCarthy.

Two Notions of Mathematics

In "Under the Spell of Leibniz's Dream," Edsger Dijkstra wrote that his view of mathematics changed from the dictionary definition of

"the science of space, number and quantity"

to its different understanding as noticed by mathematicians such as Leibniz, Boole, and DeMorgan

"the art and science of effective reasoning."

Programming as Mathematics

He explained that

> *This was refreshing because the methodological flavour gave it a much wider applicability. Take a sophisticated piece of basic software such as a programming language implementation or an operating system; we cannot see them as products engendered by the abstract science of space, number and quantity, but as artefacts they are logically so subtle that the art & science of effective reasoning has certainly something to do with their creation: in this more flexible conception of what mathematics is about, programming is of necessity a mathematical activity.*

Axiomatizing Artificial Intelligence

- In the 1950s, John McCarthy applied the axiomatic method to Artificial Intelligence and the idea of the *logic-based approach to AI* was born.
- Original idea: express knowledge in mathematical logic and use an inference engine that applies logical deduction.
- Prolog (1970s, Kowalski, Colmerauer, Roussel)
  - supply knowledge in the form of definite causes

    $$p \leftarrow q_1 \wedge q_2 \wedge \cdots \wedge q_n$$

  - prove that objects in the domain have given properties
  - SLD resolution devised to compute these inferences
  - Turing complete, but not fully declarative
- Datalog is a subset of Prolog that is fully declarative. It is used in deductive databases and significantly expands the more traditional query-answering languages of relational databases.

Is FOL Enough?

This issue is still being debated, but many researchers agree with the *Stanford Encyclopedia of Philosophy* in its statement that

> *One of the most significant developments both in logic and artificial intelligence is the emergence of a number of non-monotonic formalisms, which were devised expressly for the purpose of capturing defeasible reasoning in a mathematically precise manner.*

So, what is defeasible reasoning?

Defeasible Reasoning

Again from the *Stanford Encyclopedia of Philosophy*:

> *Reasoning is defeasible when the corresponding argument is rationally compelling but not deductively valid. The truth of the premises of a good defeasible argument provide support for the conclusion, even though it is possible for the premises to be true and the conclusion false. In other words, the relationship of support between premises and conclusion is a tentative one, potentially defeated by additional information.*

In other words, it is reasoning whose conclusions are based on statements such as

"*Normally*, X is true."

Defeasible Reasoning and Common Sense

- Common Sense:
  - good sense and sound judgement in practical matters (Google)
  - the ability to think and behave in a reasonable way and to make good decisions (Merriam Webster)
  - what -I- think people should know (Urban Dictionary)
- Claim: Much of common sense is based on being able to make smart generalizations while, at the same time, remembering that there are exceptions.
- Therefore, being able to encode the notion of "normally X is true" is vital to imparting a machine with common sense.

The Birth of Nonmonotonic Logic

- John McCarthy developed *circumscription*
- Drew McDermott and Jon Doyle developed *nonmonotonic logics*
- Ray Reiter developed *default logic*
- Robert Moore developed *autoepistemic logic* which served as a starting point for the development of ASP.

Answer-Set Programming

ASP is one particular manifestation the logic-based approach. I'd like to follow the development of more and more complex agents, from knowledge representation to reasoning such as planning and diagnostics, and show how some traditional problems of AI were solved with this approach.

Why Answer-Set Programming?

The ASP approach to AI

- separates knowledge representation and algorithm, allowing the same knowledge base to be used for a variety of reasoning tasks;
- is state-of-the-art and is explored by a lively community of researchers around the world;
- has applications in diverse domains;
- is elegant;
- is what I know the most about.

Areas of AI that Include Applications of ASP

From "Applications of Answer Set Programming" by Esra Erdem, Michael Gelfond, Nicola Leone, published in *AI Magazine*, Fall 2016.

- planning
- probabilistice reasoning
- data integration and query answering
- multiagent systems
- natural language processing and understanding
- learning
- theory update/revision
- preferences
- diagnostics
- semantic web
- and more

Other Areas that Include Applications of ASP

From "Applications of Answer Set Programming" by Esra Erdem, Michael Gelfond, Nicola Leone, published in *AI Magazine*, Fall 2016.

- bioinformatics
- automatic music composition
- assisted living
- software engineering
- robotics

Industry Applications of ASP

From "Applications of Answer Set Programming" by Esra Erdem, Michael Gelfond, Nicola Leone, published in *AI Magazine*, Fall 2016.

- decision support systems
- automated product configuration
- intelligent call routing
- configuration and reconfiguratin of railway safety systems

Answer-Set Prolog (ASP)

- 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.
- The valid consequences of the program are the statements that are true in all such sets of beliefs.
- Represent objects and their relations nicely, and you can solve a lot of practical problems (and perhaps learn about the human mind in the process.)

Syntax Overview: ASP Building Blocks

Signature + Sorts

Terms

Atoms

Connectives

$\neg$	classical negation
not	default negation
$\leftarrow$	if
or	disjunctive or

Atoms plus connectives allow us to construct rules.

Syntax: The Signature

- The building blocks of ASP programs are
  - objects
  - functions
  - predicates (i.e., relations)
  - variables
- This is know as the program **signature** (Σ).
- Functions and predicates have an arity associated with them.
- **arity** — a non-negative integer indicating the number of parameters.
- Whenever necessary, we assume that our signatures contain standard names for non-negative integers, functions, and relations of arithmetic (e.g., $+$, *, $\leq$).

What Is the Signature of this Program?

$$
\begin{array}{l}
father(john, sam).\\
mother(alice, sam).\\
gender(john, male).\\
gender(sam, male).\\
gender(alice, female).\\
\\
parent(X, Y) \leftarrow father(X, Y).\\
parent(X, Y) \leftarrow mother(X, Y).\\
\\
child(X, Y) \leftarrow parent(Y, X).
\end{array}
$$

Signature

$\Sigma = \{\mathcal{O}, \mathcal{F}, \mathcal{P}, \mathcal{V}\}$ where

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

Adding Sorts

- The notion of a *sort* in ASP is relatively new and is not a part of some implementations; however, in my experience, it improves the readability of the code and is pretty intuitive.
- Sorts are normally used to restrict the parameters of predicates, as well as parameters and values of functions. (Like types in procedural languages.)
- We can make *gender* a sort (male and female) and add a *person* sort (john, sam, and alice) to our original program. Then we would define relations with sorts like $father(person, person)$. We'll talk implementations later.

Syntax: Terms

- **term:**
  - Variables and object constants are terms.
  - 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.
- **Ground terms** are terms containing no symbols for arithmetic functions and no variables.
- Examples from our 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 while $2 + 3$ is not.

A Bit of Nonsense

- Suppose we had in our program signature the function symbol *car*:

$$\mathcal{O} = \{john, sam, alice, male, female\}$$
$$\mathcal{F} = \{car\}$$
$$\mathcal{P} = \{father, mother, parent, child, gender\}$$
$$\mathcal{V} = \{X, Y\}$$

- We could make ground terms $car(john)$, $car(sam)$, and $car(alice)$ and non-ground terms $car(X)$ and $car(Y)$.
- We could also make ground terms such as $car(car(sam))$.

Restricting Terms to a Sorted Signature

- We can restrict our signature by dividing our object constants into sorts.
- Our new signature, Σ_s, would have sorts

$$\begin{aligned} gender &= \{male, female\} \\ person &= \{john, sam, alice\} \\ thing &= \{car(X) : person(X)\} \end{aligned}$$

- Predicates father, mother, parent, and child would be restricted to sort *person*.
- The first parameter of *gender* would be restricted over *person* and the second, over *gender*.

Syntax: Atoms and Literals

- 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.
- Examples
  - $father(john, sam)$ is an atom of signatures Σ and Σ_s,
  - $father(john, X)$ is an atom of signature Σ and Σ_s,
  - $father(john, car(sam))$ is an atom over Σ but **not** Σ_s
- A **literal** is an atom or its negation.

Syntax: Rules and Programs

A **program** Π of ASP consists of a signature Σ and a collection of **rules** of the form:

$$l_0 \text{ or } \dots \text{ or } l_i \leftarrow l_{i+1}, \dots, l_m, \text{ not } l_{m+1}, \dots, \text{ not } l_n$$

where ls are literals of Σ.

Symbol *not* is a new logical connective called **default negation**; *not* l is often read as "*it is not believed that* l *is true*."

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*."

Facts and Constraints

- The left-hand side of an ASP rule is called the **head** and the right-hand side is called the **body**.
- A rule with an empty head is often referred to as a **constraint** and written as

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

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

$$l_0\ or\ \dots\ or\ l_i.$$

Rules with Variables

- 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.

Example of Grounding

Given 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

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

The grounding of Π_1 is simply two ground rules:

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

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

Satisfiability

When does a set of ground literals satisfy a rule?

$$l_0 \; or \; \ldots \; or \; l_i \leftarrow l_{i+1}, \ldots, l_m, \; not \; l_{m+1}, \ldots, \; not \; l_n$$

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.

Satisfiability Examples

Let r be the rule

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

Do any of these sets satisfy r?

- $\{\neg p(a), q(b), \neg t(c)\}$ no
- $\{q(b), \neg t(c)\}$ no
- $\emptyset$ yes
- $\{p(a)\}$ yes
- $\{p(b), q(b), \neg t(c)\}$ yes

Informal Semantics: Guiding Principles

- 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.
- We form these sets of ground literals by following these 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" which says: "Believe nothing you are not forced to believe."

Basic Example

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

Follow the guiding principles to compute the answer set.

Note that not every set that satisfies the rules is an answer set.

Example: 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.

Example: Epistemic Disjunction

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

There are two answer sets of this program.

The Rationality Principle eliminates the third possibility.

Example: Reasoning by Cases

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

$$A_1 = \{p(a), q(a)\} \text{ and } A_2 = \{p(b), q(a)\}$$

Example: Epistemic Disjunction Is Not XOR

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

$A = \{p(a), p(b)\}$ is not contradictory.

To get exclusive or, we say

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

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

$A_1 = \{p(a), \neg p(b)\}$ and $A_2 = \{\neg p(a), p(b)\}$.

Example: Constraints

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

The only answer set is $\{p(b)\}$.

A constraint limits the sets of beliefs an agent can have, but does not serve to derive any new information.

Example: Default Negation

Agents can make conclusions based on the absence of information.

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

We cannot prove $q(a)$, so we believe $p(a)$.

Example 2: Default Negation

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

The only answer set is {q(a),p(b)}.

ASP Entailment

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

Unlike the entailment relation of classical logic, ASP's entailment relation is **nonmonotonic.** This means that addition of new knowledge can invalidate the program's original conclusions.

Example: Nonmonotonicity

A program consisting only of rule

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

entails $p(a)$.

If we add fact $q(a)$ to it, the agent has to stop believing in $p(a)$.

Answering Queries

- A query is a conjunction or disjunction of literals.
- 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 $\Pi \models \overline{l}_i$,
  - *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 $\Pi \models \{\overline{l}_1, \ldots, \overline{l}_n\}$,
  - *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 $\Pi \models q(t_1, \ldots, t_n)$.

Example: Answer to a Query

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

This program has answer set $\{p(a)\}$.
What does it answer to the following queries?

1. $p(a)$
2. $q(a)$
3. $p(a) \wedge q(a)$
4. $p(a) \ or \ q(a)$
5. $p(X)$

Example 2: Answer to a Query

- Let's make a new program by adding a rule to the previous one:

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

- This rule is known as the Closed World Assumption because we assume that if we do not know anything about $q(X)$, then it is false.
- This program has answer set $\{p(a), \neg q(a)\}$.
- What does it answer to the following queries now?
  1. $p(a)$
  2. $q(a)$
  3. $p(a) \wedge q(a)$
  4. $p(a)\ or\ q(a)$
  5. $p(X)$

Formal Semantics: Answer Sets — A Two Part Definition

- The definition of answer sets has two parts.
- The first defines answer sets of programs without default negation.
- The second explains how to remove default negation so that we can apply the first part.

Consistency

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.

Answer Sets, Part I

Let Π be a program not containing default negation.
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 which satisfies the rules of Π.

Example: Basic Application of Formal Definition

Let's apply the formal definition to our first example and check that $\{q(a), p(b)\}$ is indeed the answer set of the following program:

$$\begin{array}{l} p(b) \leftarrow q(a). \\ q(a). \end{array}$$

What about entailment and answers to queries?

- ? $q(a)$
- ? $\neg q(a)$
- ? $p(b)$
- ? $\neg p(b)$

Example 2

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

What does the program believe? Compute the answer set and use entailment to answer queries:

- ? $p(a)$
- ? $\neg p(a)$
- ? $p(b)$
- ? $\neg p(b)$

Note that this example demonstrates that $\leftarrow$ is not classical implication.

Example: Empty Answer Set

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

Does the program have an answer set?

What do we believe about $\neg p(a)$? How about $p(b)$?

Example: Epistemic Disjunction

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

has two answer sets, $\{p(a)\}$ and $\{p(b)\}$.
Does it entail $p(a)$?

What does the following program entail?

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

Example: $p(a)$ *or* $\neg p(a)$ Is Not a Tautology

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

The addition of the last rule forces the agent to make a decision one way or the other, instead of remaining undecided. Instead of an empty set, we have two answer sets: $\{p(a), p(b)\}$ and $\{\neg p(a), p(b)\}$. What does this program entail?

Example: Constraints, Revisited

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

We have two answer sets from the first rule, but the second rule makes us exclude the possibility of $\{p(a)\}$ because it is impossible to satisfy an empty head if the body is satisfied.

Definition of 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.

Default Negation, Revisited
Creating a Reduct

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

Find the reduct of this program w.r.t. $S = \{q(a), p(b)\}$.

Is S an answer set of the program?

A Proposition for Our Intuition

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.

Example: No Answer Set

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

It is silly to ask an agent to believe in something simply because it does not believe it.

Consider the possible sets that we can create from the signature of the program. They are $S_1 = \{\}$, $S_2 = \{p(a)\}$, $S_3 = \{\neg p(a)\}$, and $S_4 = \{p(a), \neg p(a)\}$. None of these are answer sets.

Note that, thanks to our proposition, we do not really even need to consider S_3 because $\neg p(a)$ is not found in the head of any rule; thus, we know that there is no support for it. We can also ignore S_4 because it is inconsistent. The other two sets are tested by finding the reduct of the program w.r.t. those sets.

Example 2: No Answer Set

Other inconsistent programs (programs that have no answer sets) include

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

and

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

Exercise 1

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

- What are the possible, *consistent* sets that we can create from the signature of this program?
- Find the reduct w.r.t. to each of these sets.
- What is the answer set of this program?
- Can we be smarter about our search for the answer set?

Exercise 2

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

Find the answer set(s), if they exist.
Hint: Use the proposition to reduce the number of candidate sets.
Then find the reduct w.r.t. those candidates.

Exercise 3

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

Find the answer set(s), if they exist.

Exercise 4

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

Find the answer set(s), if they exist.

Exercise 5

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

Find the answer set(s), if they exist.

Exercise 6

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

Find the answer set(s), if they exist.

Example: Using the Notion of Support to Find the Answer Sets

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

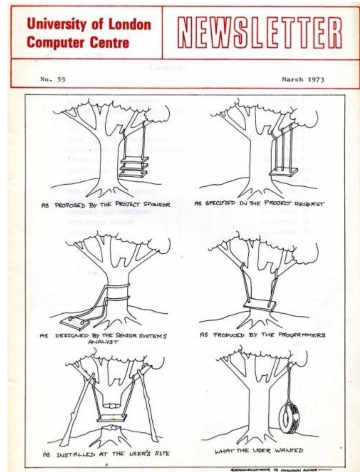
After grounding the program has the form

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

Find the answer sets and check that they are correct using the definition.

"Word Problems"

Any time we go from a natural language specification to a formal language, we face challenges. We are all familiar with this picture or at least this effect.



It Seems Easy

Let's translate the following statement into ASP:
"All professors are adults."

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

If we add a list of professors, this program would be able to figure out that they are adults.
What should it conclude if we tell it that Alice is not an adult?

We Have Choices

Here are two ways in which we might teach the program what we want.

$$\begin{array}{|l|} adult(X) \leftarrow prof(X). \\ adult(X) \text{ or } \neg adult(X). \\ prof(X) \text{ or } \neg prof(X). \end{array}$$

$$\begin{array}{|l|} adult(X) \leftarrow prof(X). \\ \neg prof(X) \leftarrow \neg adult(X). \end{array}$$

If we add $prof(john)$ and $\neg adult(alice)$, both programs have the same answer set:

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

What are the answer sets of the programs without the facts?

Representation is an Art

- Throughout the book, we'll try to point out qualities that we think a good program should have under particular sets of circumstances.
- Develop your taste.
- Pay attention to elegance.
- Ask yourself what the program knows at any given point.
  - Are its replies intuitive?
  - Is the code elaboration tolerant?

Properties of ASP Programs

We won't go into the details, but note these two things.

1. There is a property of ASP programs called local stratification. If we can show that a program is locally stratified, we can learn useful things about it.
2. There is a simple algorithm for removing classical negation and constraints from a program, so answer set solving algorithms can be devised for programs without negation and constraints and works just fine.