AI2 Practical 3 (Module 2)
First-order Representation and Reasoning (Version 2)

Deadline: 4pm, Friday, 19 November 2004

General Instructions

This document describes the first assessed assignment for AI2 Module 2. For this assignment, you will be creating several files. You should put them all in a single sub-directory which contains only the work for this assignment. All filenames must be as specified in this document. The format of the submission command for this assignment is:

```
    submit ai2 ai2a A3 DIR
```

where `DIR` is the name of the sub-directory in which you have put all your work. Make sure that if you are specifying a relative pathname that you are in the right directory to be doing so.

Some Comments on Plagiarism

1. Submitting another student's code (even if modified) as if it is your original work is plagiarism.
2. Assisting another student to plagiarise (eg. by sharing code without the borrowing student acknowledging the use) is also penalised.
3. Discussing the assignment in broad terms is ok, but not at the level of coding details.
4. If you cannot figure out how to code some portion of your program, you **could** borrow someone else's code, **provided** you clearly acknowledge whose code it was and what portion of your submission is theirs. You will not get credit for the other person's code.
5. Even partial, non-working submissions get partial credit.
6. A failed assignment does not mean a ruined career. Getting caught at plagiarism might.
7. We use various techniques to detect plagiarism, including automated tools .
8. If you can't do the assignments, discuss this with the course organiser or your director of studies.
9. Change the protections on your homework files and directories so potential plagiarists cannot access your solution. If your file is called XXX, then use the unix command: `chmod go-rwx XXX`. Don't assume that your flatmates or best friends would never plagiarise from you.

Otter

As mentioned in class and in Russell & Norvig, Chapter 9.5, Otter is a well-known theorem prover for FOL based on resolution, developed at the Argonne National Labs in Illinois. You can find more information about Otter at `http://www-unix.mcs.anl.gov/AR/otter/`. Otter has been installed on DICE machines under `/usr/bin/otter/`. If you want to run Otter on your own machine, it is freely available in UNIX, PC, and Mac versions at the previously mentioned URL, with documentation and examples.

Although Otter has a primitive interactive interface, I recommend you use it at the command-line. On UNIX and LINUX systems, you can use the `<` and `>` piping operators as follows:

```
% otter < inputfile > outputfile
```

No options are accepted at the command-line: They must all be specified in the input file. *outputfile* will contain either error reports or a record of Otter's attempt to prove the goal clause given in *inputfile* from the axioms also given there.

All the proofs in this exercise should be relatively simple. If Otter is taking a long time (i.e., more than a few seconds) to return, it is probably because the given axioms don't actually support a proof. Typing `Control-C` will interrupt Otter, allowing you to specify a number of options: Type "help." to see them. Among the options is "fail.", which will halt processing. *outputfile* will then contain a record of what has been tried. If Otter returns "Search stopped by Max_proofs option", that means that it has succeeded in finding a proof. (We can increase the value of max_proofs if we want Otter to find >1 proof.)

At the end of this exercise you will have had useful experience in:

1. translating problems and questions stated in Natural Language into formulae that Otter can try to prove;
2. identifying what implicit background knowledge needs to be made explicit for a proof to go through;
3. using Otter for obtaining simple proofs in FOL and getting answers to questions.

Task 1: Getting to know Otter (15%)

This first task is designed to familiarize you with Otter, and how you can use it to prove queries about problems stated in first-order logic (FOL).

Consider the following set of statements:

Not everyone who studies biology studies history.
Not everyone who studies history studies biology.
Everyone who studies something takes an exam in it.
The only ones who take an exam in something are those who study it.

and the query: *Is there some one who takes an exam in history who does not take an exam in biology?*

Represent both the statements and the query as a set of formulae in FOL, using the predicate symbols `study/2` and `exam_in/2` and the constants `bio` and `history`.

Now enter the problem specification (i.e., the statements) and the negation of the query into Otter and see whether the query is true, given the specification. To do this, use the sample input file for Otter to be found at

`http://www.informatics.ed.ac.uk/teaching/courses/ai2/module2/assessed/template.in`

As well as commands in this file, comments in the file demonstrate the syntax you will need to use for entering problems into Otter. You can either retain these comments to remind you of Otter's encoding conventions, or remove them. A copy of the template is given at the end of the assignment.

Once you fill out the template with your encoding of the statements and the negation of the query, save the file in your work directory as `task1.in` and run it through Otter using Unix piping: `%otter < task1.in > task1.out`. Otter will only provide a short message informing you of its conclusion. The full details, including the proof if it's found one, will be in `task1.out`.

Before you submit `task1.out`, add to the end of the file a short description of what Otter does if it also has particular facts about people studying different subjects and taking exams in them – e.g. `study(matt,swedish).  exam_in(sarah,bio).`

To be submitted:

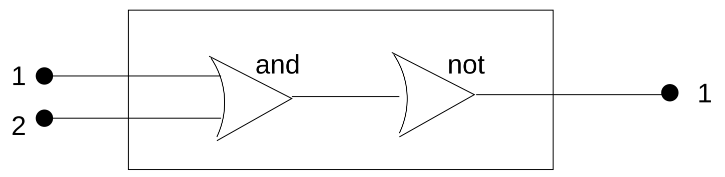
- `task1.in`
- `task1.out`, with your comments added to the end

Task 2: A simpler representation of digital circuits (70%)

One-bit adder

The digital circuit representation in R&N Chapter 8 and discussed in class is more detailed than necessary to reason only about *circuit functionality.* A simpler formulation describes any m-input, n-output gate or circuit using a predicate of arity $m+n$, such that the predicate is true exactly when the inputs and the outputs are consistent for the type of gate involved. For example, NOT gates can be represented simply using the predicate symbol `not/2` and the facts `not(0,1)` and `not(1,0)`, where the first argument represents the signal at the input terminal to the NOT gate and the second argument represents the signal at its output terminal.

Note that the predicate symbol `connected/2` is not needed in this simpler formulation. Direct connections between gates are indicated through *shared variables.* For example, a NAND circuit (i.e., a circuit consisting of an AND gate feeding into a NOT gate)



can be represented simply as:

$$\forall \text{ In1, In2, Oa, Out . nand(In1,In2,Out)} \Leftrightarrow \text{and(In1,In2,Oa)} \wedge \text{not(Oa,Out)}$$

Using this as a model, you should represent the ONE-BIT ADDER circuit shown in R&N Figure 8.4 (the same as the one-bit adder discussed in class), **as well as** the general facts about

AND, OR, XOR and NOT gates needed to reason about it. Enter this problem specification into the Otter template and save it under the name `task2.in`.

Now you will use Otter to verify that what you've specified is a one-bit adder. For this, you need to do three things:

- Figure out what *query* you need to pose to Otter in order to verify that you have specified a one bit adder. (If you need some help with this, re-read R&N pp. 262–6.)
- Attach an *answer literal* to your query so that you know what signal values allow Otter to prove the query. (An answer literal – p.42 of the Otter Reference Manual – is one whose name starts with $ans, $Ans, or $ANS.)

  For example, if the question were *What are the input signals to an AND gate when the signal at the output is 1?*, the query would be ∃ X1 ∃ X2 . and(X1,X21). In Otter syntax, the negation of the query with an appended answer literal, would be:

  ```
  all X1 X2 -(and(X1,X2,1) & $ans(X1,X2)).
  ```

  Otter will then give you the values for `X1` and `X2` that allow it to produce a refutation proof of the query.

  Add your negated query and answer literal to the end of `task2.in`, above the line `end_of_list.`
- In order to get more than one proof, you need to tell Otter the maximum number of proofs it should look for. (N.B. It might not find this many. Alternatively, it may find what is essentially the same proof using different means.) So you should give Otter a maximum number that is higher than the number of distinct proofs you want.

  I suggest you add the following line to `task2.in`, right below the statement `set(auto).`

  ```
  assign(max_proofs,20).
  ```

Run `task2.in` through Otter

```
otter < task2.in > task2.out
```

and then add your answers to the following three questions to the end of `task2.out`.

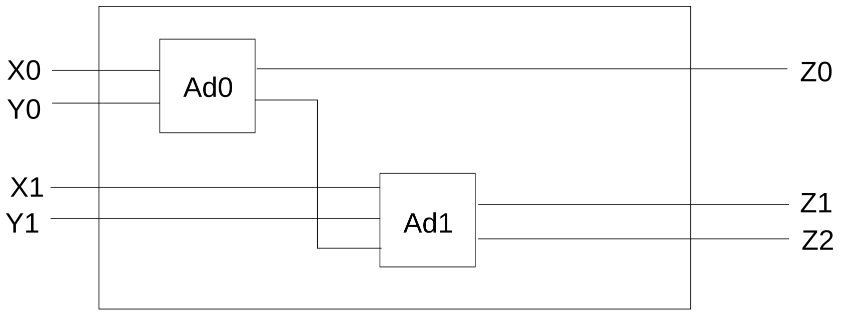
1. Explain how Otter's results do or don't confirm your design of a one-bit adder.
2. How many proofs did Otter find?
3. Is this what you expected? Explain why.

To be submitted:

- task2.in
- task2.out, with your answers to the above three questions.

Two-bit adder

Suppose we build a TWO-BIT ADDER circuit by linking two ONE-BIT ADDERS as in the following diagram.



Answer the following three questions in a file named `twobit.ans`.

1. How would you represent the new TWO-BIT ADDER circuit, given your representation of a ONE-BIT ADDER?
2. What query would you post to Otter (or another theorem prover) to verify that your representation is correct?
3. Can you generalise what would then need to be done to represent and verify an N-BIT ADDER for any N?

To be submitted:

> `twobit.ans`

Task 3: Description Logic (15%)

In R&N's description of the kinship domain (pp. 254–256), they give definitional axioms for *husband*, *grandparent* and and *sibling*. Can these terms be specified in the Description Logic (DL) whose syntax is given in R&N Figure 10.11 (p. 354)?

For each kinship term that can be specified in this DL, provide the equivalence between the kinship term and its specification in DL, just as with the term *bachelor* on p. 353:

bachelor $\equiv$ *And*(unmarried, adult, male)

For each kinship term that cannot be specified in this DL, explain what the problem is.

Call the file containing your answer to this question `task3.ans`

To be submitted:

> `task3.ans`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Otter input template file

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Let Otter decide the most appropriate search strategy to employ.
set(auto).

% Tell Otter that variables start with an uppercase letter and
% constants, with a lowercase letter, just as in Prolog.
set(prolog_style_variables).

% Tell Otter that what follows  are the first-order formulae that
% encode the problem, including the negated query at the end.
formula_list(usable).

  % Existential quantifiers take the form "exists VAR (formula)"
  % and conjunction is done with the "&" operator. For example,
  % exists X (dog(X) & owns(jack,X)).

  % Universal quantifiers take the form "all VAR (formula)"
  % and implication is done with the "->" operator. For example,
  % all X ((exists Y (dog(Y) & owns(X,Y))) -> animalLover(X)).

  % If you have a sequence of the same quantifier, you can abbreviate it:
  % Instead of ``all X1 (all X2  ...'', you can write ``all X1 X2 ...''

  % Negation is represented with a "-". For example,
  % all X (animalLover(X) -> (all Y (animal(Y) -> -kills(X,Y)))).

  % Disjunction is done with the "|" operator. For example,
  % kills(jack,tuna) | kills(curiosity,tuna).

  % Equivalence (bi-directional implication) is done with the <->
  % operator. For example, all X Y (above(X,Y) <-> below(Y,X)).

  % Your negated query comes last. For example,
  % -kills(curiosity,tuna).

% Indicate the end of the formula specification.
end_of_list.
```