Imperial College London – Department of Computing

MSc in Advanced Computing
MSc in Computing Science (Specialism)

531: Prolog
'Syllogisms' – Assessed

Issued: 14 November 2018 **Due:** 22 November 2018

Aims

This exercise will give you practice of writing both a definite clause grammar and a meta-interpreter. If you complete all the steps of this exercise you will have implemented a validity checker for syllogistic argument forms, also referred to as Aristotelian Logic after the Greek philosopher Aristotle (384 BC - 322 BC) who identified and formalised syllogistic argument.

Background

Sentence Forms

A syllogistic argument uses *sentences* of the following four forms:

"a B is a C"	(in the sense of "every B is a C")
"some B is a C"	
"no B is a C"	
"some B is not a C"	

where B and C are variables that can be replaced by words denoting classes of things. For instance "a sparrow is a bird", or "no table is a genius". In each case the "a" before the C is optional. (Any use of "a" could be also be "an", but you can ignore this for the purposes of the exercise.) In the first form, "a B" could also be "every B".

Each sentence form has an opposite form. They can be arranged in two pairs as follows:

"a B is C"	"some B is not C"
"no B is C"	"some B is C"

Argument Forms

A *syllogistic argument*, or *syllogism*, always consists of exactly three sentences: two *premises* and one *conclusion*. The three sentences make related statements about two or more classes of things. For example:

> “a man is mortal”
> “some philosopher is a man”
> *therefore*
> “some philosopher is mortal”

Valid Forms

Aristotle identified 18 *valid* forms of syllogism. One of them is the form of the example argument about philosophers being mortal shown above. The form of that argument is as follows:

“a B is C”	“a man is mortal”
“some D is a B”	“some philosopher is a man”
therefore	*therefore*
“some D is C”	“some philosopher is mortal”

This argument form is valid because no matter what classes of things B, C and D denote, whenever the two premises are true the conclusion is true.

Saying that this argument form is *valid* is equivalent to saying that any three sentences:

> “a B is C”
> “some D is a B”
> “no D is C”

in which the conclusion “some D is C” has been replaced by its opposite “no D is C”, are contradictory, or *inconsistent*. i.e. they cannot all be true, no matter what B, C and D denote.

Aristotle also identified *invalid* syllogism arguments. E.g.

> “a B is a C”
> “a B is a D”
> *therefore*
> “a D is a C”

For instance:

> “a man is a mammal”
> “a man is an animal”
> *therefore*
> “an animal is a mammal”   (i.e., “every animal is a mammal”)

which has true premises but a false conclusion.

Semantics

The semantics of a syllogism sentence in each of the four possible forms can be expressed by a set of first-order clauses, as detailed below:

Sentence Form	*Semantics*
"a B is a C"	$\forall X\ (C(X) \leftarrow B(X))$
"some B is a C"	$B(sk_{BC})$ and $C(sk_{BC})$
"no B is a C"	$\forall X\ (\leftarrow B(X) \wedge C(X))$
"some B is not a C"	$B(sk_{BC'})$ and $\leftarrow C(sk_{BC'})$

These clauses include some with empty heads, and are therefore Horn clauses, named after the American mathematician Alfred Horn (1918 – 2001). A headless clause cannot be true if all its conditions are true, so the clause $\forall X\ (\leftarrow B(X) \wedge C(X))$ tells us that there is no X that is both a B thing and a C thing.

The semantics of the second and fourth sentence forms are expressed using the terms sk_{BC} and $sk_{BC'}$. If you are familiar with natural deduction for predicate logic, you might recognise these as Skolem constants, named after Norwegian mathematician Thoralf Albert Skolem (1887 – 1963). If you are not familiar with natural deduction, don't worry, they are just symbols introduced to keep track of the (unnamed) things to which the sentences refer. In each of these cases the meaning of the sentence is captured by two clauses. For the second form the clauses tell us that the thing we have called sk_{BC} is both a B and a C, for the fourth form the clauses tell us that the thing we have called $sk_{BC'}$ is a B but not a C.

By proving that the clauses expressing the premises and the opposite of the conclusion of a syllogistic argument are inconsistent, we can demonstrate that the argument is valid.

The Problem

In this exercise you are provided with a file `arguments.pl` containing eight syllogistic arguments. The arguments are represented by facts defining the predicates `p/2` (premise) and `c/2` (conclusion). The second argument (term) of each fact is a syllogism sentence, represented by a list of words. The first argument of each fact is an integer identifying the syllogism to which the sentence belongs.

For example, the file contains:

```
p(1, [a,robin,is,a,bird]).
p(1, [no,bird,is,a,reptile]).
c(1, [no,robin,is,a,reptile]).
```

representing the syllogistic argument:

"a robin is a bird"
"no bird is a reptile"
therefore
"no robin is a reptile"

So, for each N ($1 \leq N \leq 8$), two `p(N,...)` facts give the premises of syllogism number `N`, and one `c(N,...)` fact gives its conclusion.

The Parser

You will first write a parser to translate syllogistic sentences into Prolog clauses. The validity of each argument can then be determined by evaluating these clauses.

The semantics of the four sentence forms will be represented by a list of Prolog clauses, as follows:

Sentence Form	*Semantics in Prolog*
"a *B* is a *C*"	`[ (C(X) :- B(X)) ]`
"some *B* is a *C*"	`[ (B( some(B,C) ) :- true), (C( some(B,C) ) :- true) ]`
"no *B* is a *C*"	`[ (false :- B(X),C(X)) ]`
"some *B* is not a *C*"	`[ (B( some(B,not(C)) ):- true), (false :- C( some(B,not(C)) )) ]`

Example: The semantic representation of the sentence `[no,bird,is,a,reptile]` is the list:

```
[ (false :- bird(X), reptile(X)) ]
```

The predicate `false` is used to represent the empty head of the first-order clause. Although `false` is a Prolog built-in that always fails, it will not have this behaviour in this exercise. The syllogism clauses will not be evaluated directly by Prolog, but rather by your meta-interpreter. That program will treat `false` as a normal user-defined predicate and evaluate it like any other.

Example: The semantic representation of the sentence `[some,boy,is,naughty]` is the list:

```
[ (boy( some(boy,naughty) ) :- true),
  (naughty( some(boy,naughty) ) :- true) ]
```

The predicate `true` is used to represent an empty clause body.

The Skolem constant is represented by the term `some(boy,naughty)`, where `some` is simply a function symbol. The arguments `boy` and `naughty` are added to differentiate between each such term, and to label what it represents. Note that `boy` and `naughty` are being used both as terms (subterms of `some(boy,naughty)`) and as predicate names within these clauses.

Example: The semantic representation of the sentence `[some,girl,is,not,a,student]` is the list:

```
[ (girl( some(girl,not(student)) ) :- true),
  (false :- student( some(girl,not(student)) ))]
```

Here the Skolem constant is represented by the term `some(girl,not(student))`. The second argument of this term is the subterm `not(student)`, in which `not` is simply a second function symbol.

The Dynamic Program

The translated clauses of the premises and the opposite of the conclusion of a syllogism are asserted as a dynamic program. Each clause is asserted as a fact for the predicate `cl/3`, such that a fact `cl(N,H,B)` records a clause for syllogism `N` with head `H` and body `B`.

Therefore, translation of the sentence `[some,girl,is,not,a,student]`, which is a premise of syllogism 7, in the third example above results in the following facts being asserted:

```
cl(7, girl(some(girl,not(student))), true ).
cl(7, false, student(some(girl,not(student))) ).
```

This form allows the meta-interpreter to determine to which syllogism each given clause belongs.

The Meta-Interpreter

In order to determine the validity of syllogism `N`, the meta-interpreter must evaluate the call `false` using only the clauses within the `cl(N,...)` facts. If `false` succeeds, the asserted clauses are proved to be inconsistent and the syllogism is valid. If `false` fails (finitely), the asserted clauses are consistent and the syllogism is invalid.

What To Do

Obtain the Exercise Files

- Clone your skeleton repository: (replacing the occurrence of *login* with your DoC login):

  ```
  git clone https://gitlab.doc.ic.ac.uk/lab1819_autumn/531_syllogisms_login.git
  ```

  Or, if you have set up ssh key access you can use:

  ```
  git clone git@gitlab.doc.ic.ac.uk:lab1819_autumn/531_syllogisms_login.git
  ```

- This will create a new directory called `531_syllogisms_`*login*. Inside you will find the following files / directories:
  - `syllogisms.pl` — this is the source file you should edit.
  - `arguments.pl` — this file contains the premises (`p/2` facts) and conclusions (`c/2` facts) for eight candidate syllogistic arguments.
  - `utilities.pl` — this file contains some definitions that you are free to use in your answers. You can consult this file, but your submitted answer should not. You can assume the file will be consulted for you when your program is tested.
  - `.git` and `.gitignore`

Write the Program

Write your answer in `syllogisms.pl`.

Step 1 (10% of marks)

Define a Prolog predicate `opposite/2` that will convert any list of words corresponding to a syllogism sentence to *just one* list of words that is its opposite. The following pairs of sentence forms are deemed to be opposites:

```
<article> B is <optional_article> C      some B is not <optional_article> C
       no B is <optional_article> C      some B is <optional_article> C
```

where `<article>` is `'a'` or `'every'` and `<optional_article>` is `'a'` or nothing.

For example, the queries:

```
?- opposite([no,robin,is,a,reptile], O1)
?- opposite([every,student,is,lazy], O2)
```

should produce `O1 = [some,robin,is,a,reptile]` and `O2 = [some,student,is,not,lazy]`.

Because some elements of syllogism sentence forms are optional, and (in our version) the word `'every'` can be replaced by the word `'a'`, there could be several possible opposites for any given sentence. Your program should produce *just one* of them.

(*HINT.* The `opposite` predicate is used to preprocess some of the sentences *before* translating them with the grammar you will write in Step 2. You do not need to write a grammar program in this step. Write the answer the simplest way possible.) You can assume that the list of words given in `p/2` and `c/2` will always be ground (and meaningful).

Test your answer by finding all solutions to the query:

```
?- c(N,S), opposite(S,O).
```

using the `c/2` facts of `arguments.pl`.

Step 2 (40% of marks)

Write DCG grammar rules that parse each of the four syllogism sentence forms into a list of clauses representing its semantics, as detailed above.

It is recommended that you complete this step in two stages. First, write DCG rules for `syllogism/0` that correctly identify each of the sentence forms. Once these rules are working correctly, convert them into rules for `syllogism/1` that also generate the translated clauses. These two stages are explained below.

Stage 2.1: `syllogism/0`. Write four DCG rules for `syllogism`, and any subordinate rules necessary, such that that each `syllogism` rule recognizes lists of words in one —and only one — of the sentence forms. Use the templates shown in Step 1 for each form. You should not try to generalise these templates any further, since each form must be defined by a distinct rule.

When writing these rules note that, unlike the English language grammar in the lecture notes, some (many!) of the words in the syllogism sentences (in particular the `B`s and `C`s) will not be listed in dictionary rules. Given the assumption from Step 1 that all input sentences are meaningful, the grammar rules can allow some positions in a sentence to be filled by any word.

Test your answer by finding all solutions to queries such as:

```
?- p(N,S), phrase(syllogism,S).
?- c(N,S), opposite(S,O), phrase(syllogism,O).
?- forall(p(N,S), phrase(syllogism,S)).
```

where `p/2` and `c/2` are as defined in `arguments.pl` and `forall/2` is defined in `utilities.pl`. The third query should simply succeed if all the rules are defined correctly.

Stage 2.2: `syllogism/1`. Convert your four rules for `syllogism` into DCG rules for `syllogism/1` each of which will translate a syllogism sentence into a corresponding list of clauses. More specifically, a query `?- phrase(syllogism(Cs),S)`, given a list `S` representing a syllogism sentence, should return `Cs` as the list of clauses defining the semantics of `S`.

HINT. You will find the `=..` Prolog primitive useful for composing terms from a list of their constituent parts. For example, the call `Term =.. [f,a,b,c]` will bind `Term` to `f(a,b,c)`. You may also find it useful to look at the arithmetic grammar example in the DCG lecture notes when attempting this part of the exercise.

Test your answer by finding all solutions to the queries:

```
?- p(N,S), phrase(syllogism(ClauseList),S).
?- c(N,S), opposite(S,O), phrase(syllogism(ClauseList),O).
```

Check your answers against the semantic representations given as comments in the `arguments.pl` file. Remember that names of variables do not matter, and nor does the order of conditions in the body of a clause.

Step 3 (15% of marks)

Write a program `translate/1` such that a call `translate(N)` will (a) convert the two premises and the opposite of the conclusion of syllogism `N` (as given by `p/2` and `c/2`) to clauses, and then (b) record all the generated clauses for syllogism `N` as facts for the predicate `cl/3`, with first argument `N`. You might find it useful to use the `assertall/2` program of the `utilities.pl` file.

As an example, `translate(1)` should cause the following facts to be asserted:

```
cl(1, bird(X), robin(X) ).
cl(1, false, (bird(X),reptile(X)) ).
cl(1, robin(some(robin,reptile)), true ).
cl(1, reptile(some(robin,reptile)), true ).
```

After you have executed a call to `translate/1`, say `translate(3)`, check that you have generated the correct `cl/3` facts by executing the query `?- cl(3,H,B)`. If they are not what you expect, you can remove them using the query:

```
?- retractall(cl(3,_,_)).
```

Then edit your program, reconsult, and try again.

Step 4 (20% of marks)

Write a program `eval/2` such that a call `eval(N,Calls)` succeeds if each condition in `Calls` can be derived using just the clauses recorded as `cl(N,H,B)` facts. For example:

```
?-eval(1, (bird(X),reptile(X)) ).
X = some(robin,reptile)

?-eval(1, false).
yes
```

Refer to the simple Prolog evaluator of the lecture notes for a guide. Note: `false` does not need a special `eval/2` clause. It is handled like any call: check if there is a `cl/3` clause with index `N` whose head is `false`; if there is, recursively evaluate the body of the clause.

Next write programs for `valid/1` and `invalid/1` such that `valid(N)` and `invalid(N)` succeed if and only if syllogism number `N` is valid or invalid, respectively. (The fact that query `eval(1, false)` succeeds demonstrates that syllogism number `1` is valid. See above.) Allow for the case that `N` is a variable. You can assume that the syllogisms have already been translated and asserted by calling `translate/1`, and that this translation has succeeded. This applies to `eval`, `valid` and `invalid`. These predicates are *NOT* responsible for translating the arguments.

Step 5 (15% of marks)

Using the answer to Step 4, and `show_clauses/1` defined in `utilites.pl`, define `test/1` such that the query `?- test(1)` will display

```
syllogism 1:
   a robin is a bird
   no bird is a reptile
   =>
   no robin is a reptile

Premises and opposite of conclusion converted to clauses:
   bird(_d90) :- robin(_d90).
   false :- reptile(_d9c), bird(_d9c).
   robin(some(robin,reptile)) :- true.
   reptile(some(robin,reptile)) :- true.

false can be derived, syllogism 1 is valid.
```

The variable name `_d9c` could be anything, of course. `?- test(5)` will produce:

```
syllogism 5:
   every philosopher is a logician
   every philosopher is a professor
   =>
   every logician is a professor

Premises and opposite of conclusion converted to clauses:
   logician(_a14) :- philosopher(_a14).
   professor(_a14) :- philosopher(_a14).
   logician(some(logician,not(professor))) :- true.
   false :- professor(some(logician,not(professor))).

false cannot be derived, syllogism 5 is invalid.
```

You might find it useful to define an auxiliary predicate `show_syllogism/1` such that a call `show_syllogism(N)` displays the (untranslated) premises and conclusion of the `N`th syllogism as required by `test/1`. For example:

```
?- show_syllogism(1).
syllogism 1:
   a robin is a bird
   no bird is a reptile
   =>
   no robin is a reptile
```

Testing

You should test your program on a range of suitable examples before submission. You can automate your testing using the `plunit` Sicstus library. See the Sicstus documentation for details.

Your work will be automatically tested. Before submitting it please ensure that:

- Your Prolog program is written in `syllogisms.pl`.
- Your program *COMPILES WITHOUT ERRORS* on the **Linux Sicstus** system installed on the lab machines.
- Do *not* load `utilities.pl` or `arguments.pl` from `syllogisms.pl`, nor copy nor redefine any of the predicates they define: `utilities.pl` will be loaded automatically during testing. Your programs will be evaluated on a different set of example syllogisms.
- Please do *not* include error messages and other forms of output in your submitted programs besides those specified above. (You may include them in your own versions of course, but please do not include them in the submitted version.)

You can verify that your code runs and produces sensible output by requesting an autotest online via `https://teaching.doc.ic.ac.uk/labts`. This is NOT the full set of tests that will be used in assessing your work, so you cannot assume that your program is correct if all the tests pass.

Submission

Submit By: 22 November 2018

Submission is a two stage process.

1. **Push To Gitlab.** Use `git add`, `git commit` and `git push` to update the Gitlab server with the changes you have made to the skeleton repository. Use `git status` to confirm that you have no local changes you have not pushed, and then inspect the files on Gitlab:

   `https://gitlab.doc.ic.ac.uk`

2. **Submit directly to CATE.** Go to LabTS (`https://teaching.doc.ic.ac.uk/labts`), find your list of commits for this exercise and click the **Submit to CATe** button for the commit you want to submit.

Assessment

This exercise is worth 20% of the marks allocated for coursework on this module. Your solutions will be marked on correctness, code design and readability.

Grades

`F-E:`	The answer shows a clear lack of understanding of how to write Prolog code. The behaviour of the program is likely to be incorrect.
`D-C:`	The submission shows that the student can write Prolog, but the behaviour of the program is incorrect. Solutions may have serious efficiency problems. Solutions may be grossly overcomplicated, or otherwise poorly designed. Code style may be poor.
`B-A:`	The behaviour of the program is mostly or entirely correct. Solutions may have efficiency problems, and/or some design flaws. Code style is good.
`A+:`	The behaviour of the program is correct. Some minor efficiency problems may exist in some predicates, but otherwise code is well designed, simple and elegant. Code style is good.
`A*:`	There are no obvious deficiencies in the solution, including efficiency of the program, program design and the student's coding style.

Return of Work and Feedback

This exercise will be marked and returned by **7 December 2018**. Feedback on your solution will be given on the returned copy.