Imperial College London – Department of Computing

MSc in Advanced Computing
MSc in Computing Science (Specialism)

531: Prolog
'Logical Formulas' – Assessed

Issued: 10 October 2018 **Due:** 22 October 2018

Part 1: Obtaining the Files and Using Git

The process of completing each exercise will consist of four main steps:

- Getting the skeleton files for the exercise with Git.
- Solving and testing your solution to the exercise.
- Committing and pushing your changes with Git.
- Checking the autotester, and submission via CATE.

This first exercise will work through the four sections above in detail.

1.1 Git and Gitlab

Your solution to this exercise will be housed in a code repository managed via the *Git* version control system. Git is a very powerful and flexible tool. We will cover all the basic commands you need for this exercise, but if you have not used Git before you are encouraged to find out more about it online. A recommended source is `http://git-scm.com/book`.

The department has its own *Gitlab* server (`https://gitlab.doc.ic.ac.uk`). This is a convenient central place to locate Git code repositories. You are free to go onto the department Gitlab and create your own project for any course assignment or private project. For this assignment a repository has already been created for you.

A note on plagiarism

You must never leave your solution to any assessed exercise where others can see it. This includes publicly visible Git servers such as 'GitHub'. Doing so can implicate **you** in cases of plagiarism if others copy your work.

1.2 Obtaining the Exercise Files

You will need to *clone* the exercise respository to your DoC home directory in order to work on it. Later you will need to *push* your changes back to the server.

1. Before you proceed, you will need to log into the Gitlab server just once so it validates your login. Please visit `https://gitlab.doc.ic.ac.uk` and log in (using 'LDAP', with your normal college login and password).

2. Open a terminal. Navigate to wherever you want to save your work for this exercise. You can create a local copy of the repository by issuing the following command (all on one line, replacing the occurrence of *`login`* with your DoC login):

   `git clone https://gitlab.doc.ic.ac.uk/lab1819_autumn/531_formulas_`*`login`*`.git`

   This will create a new directory called `531_formulas_`*`login`*. Inside you will find the following files and directories:

   - `formulas.pl` — this is the source file you should edit to implement the procedures required for this exercise.
   - `support.pl` — this file contains a provided Prolog procedure and some test examples.
   - `.git` — this directory contains information for Git to be able to track your changes locally. You should not normally need to edit the files in here.
   - `.gitignore` — this file tells Git what files it should ignore. We have set it up with some sensible defaults, but you are free to edit it if you need to.

1.3 Pushing Your Changes

Having successfully cloned the repository you can proceed with the exercise. Add definitions of the Prolog procedures described in Tasks 1–4 below to the file `formulas.pl`. In addition to saving your work regularly as you would normally do, you should also regularly *push* your work to the Gitlab server.

1. In a terminal that is in the `531_formulas_`*`login`* directory, use `git status` to show you what has changed:

```
> git status
On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   formulas.pl
```

2. You can tell Git you wish to *stage* the changes to the modified file(s) using `git add`:

```
> git add formulas.pl

> git status
# On branch master
# Changes to be committed:
#    (use "git reset HEAD <file>..." to unstage)
#
#        modified: formulas.pl
```

3. To store these changes in Git, you now need to *commit* them with a descriptive message. Running:

```
> git commit
```

will cause this to happen. A text editor window will pop up: use this to write a short, but meaningful, description of the changes you have made.

4. Finally, to send your changes back to the Gitlab server (so they are backed up elsewhere, available to access from home, etc.), issue the command `git push`:

```
> git push
Username for https://gitlab.doc.ic.ac.uk: <enter your login>
Password for https://<login>@gitlab.doc.ic.ac.uk: <enter your password>
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 4.00 KiB, done.
Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To https://gitlab.doc.ic.ac.uk/lab1819_autumn/531_formulas_login.git
    50db2c6..97cd58f master -> master
```

You can check the push succeeded by looking at the state of your repository in the Gitlab webpages. We will return to these webpages later during submission.

1.4 Working from Home

If you wish to work from home, the given `git` commands should work from outside college too. However, to manage multiple copies of your work you will need to know a little more about Git, such as the commands `git fetch` and `git merge`. Please ask a lab helper, or look at `http://git-scm.com/book` for help with these.

If you get fed up with typing your password to be able to push, it is possible to avoid this by adding *ssh keys* to Gitlab to enable ssh access. You do this via your profile on Gitlab. Please ask if you need help.

It is really, really, really bad practice to try to send lab work to/from home using sharing systems such as Dropbox or (much worse!) email. Git allows you to accurately keep track of your changes and scale from single to multi-user projects. It is worth spending the time now to get used to it.

Part 2: The Exercise

Introduction

This exercise requires you to write Prolog programs to analyse representations of formulas in propositional logic.

Propositional formulas will be represented by one of the following:

`p`, `q`, `r`, etc.	represent propositional atoms p, q, r, etc.
`neg(X)`	represents $\neg(X)$
`and(X,Y)`	represents $(X \wedge Y)$
`or(X,Y)`	represents $(X \vee Y)$
`imp(X,Y)`	represents $(X \rightarrow Y)$

For example, the Prolog term `and(q, imp(or(p, q), neg(p)))` represents the formula

$$(q \wedge ((p \vee q) \rightarrow \neg(p)))$$

For the purposes of the exercise, a *logical atom* is a string beginning with a lower case letter. This is not quite the same as Prolog's 'atom', which includes such things as '!', '*', and so on.

The file `support.pl` contains the Prolog code for `logical_atom/1`. It also contains some example formulas that you can use to save work when testing your programs. You can test your program by executing queries such as:

```
?- formula(1,F), ats(F,As).
?- formula(3,F), ats(F,As), subseq0(As,Val), t_value(F, Val, V).
```

`subseq0/2` is provided in the Sicstus `lists` library. You can use `support.pl` to load this library — see comments for details.

Task 1

In `formulas.pl`, write a program for the predicate `wff(F)` which holds when `F` is a (well-formed) formula. For instance, the query:

```
?- wff(and(q, imp(or(p, q), neg(p)))).
```

should succeed, whilst the following queries:

```
?- wff(and(q, geoff(or(p, q), neg(p)))).
?- wff(and(q, imp(or(p, 14), neg(p)))).
?- wff(and(q, imp(or(p, q, r), neg(p)))).
```

should fail. A call `wff(F)` should only succeed if `F` is ground, i.e. if `F` contains no variables. There is a Sicstus built-in primitive `ground/1` which you can use.

Task 2

In `formulas.pl`, write a program for the predicate `cls(F)` which holds when the formula `F` is a clause; a clause is either a literal or a disjunction of literals, and a literal is either an atom or a negated atom. For instance, the queries:

```
?- cls(or(or(p, q), neg(r))).
?- cls(or(p, or(q, neg(r)))).
```

should both succeed, whilst the following should fail

```
?- cls(or(or(p, q), neg(neg(r)))).
?- cls(or(imp(p, q), neg(r))).
```

A call `cls(F)` should succeed only if `F` is a (ground) well-formed formula, i.e. if `wff(F)` succeeds.

Task 3

In `formulas.pl`, write a program for the predicate `ats(F, As)` which, given the formula `F`, returns `As` as a *duplicate-free* list (in any order) of the atoms in `F`. For instance, the query

```
?- ats(and(q, imp(or(p, q), neg(p))), As).
```

should return just one solution for `As`, this being either `[p, q]` or `[q, p]`. A call `ats(F, As)` should succeed only if `F` is a (ground) well-formed formula.

Task 4

A 'valuation' is a function which assigns to every logical atom of a formula one of the two truth values `t` or `f`. Suppose, for the sake of the exercise, that a valuation is represented as a list of atoms: namely the atoms that have been assigned the value `t`; all other atoms not on the list are assigned value `f`.

In `formulas.pl`, write a program `t_value(F, Val, V)` to calculate the truth value `V` of the formula `F`, given the valuation `Val`. For example, the following queries should return solutions for `V` as shown:

```
?- t_value(p, [p], V).                t
?- t_value(p, [], V).                 f
?- t_value(neg(p), [], V).            t
?- t_value(and(q, or(p , neg(r))), [p,q], V).   t
?- t_value(and(q, or(p , neg(r))), [q], V).     t
?- t_value(and(q, or(p , neg(r))), [], V).      f
```

(You need to encode the usual truth-tables for `and`, `or`, and `neg`. Note that in propositional logic $p \rightarrow q$ is logically equivalent to (has the same truth value as) $\neg p \vee q$.)

A call `t_value(F, Val, V)` should succeed only if `F` and `Val` are ground, if `F` is a well-formed formula, if `Val` is a list of logical atoms, and if `Val` only contains atoms occurring in `F`.

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 submitted work will be automatically tested, so you must ensure that:

- Your Prolog program is written in `formulas.pl`.
- Your program *COMPILES WITHOUT ERRORS* on the **Linux Sicstus** system installed on the lab machines.
- You have *NOT* included any print or write statements in your submitted code. Use them by all means for tracing/debugging but do not include them in your submitted program.

You can verify that your code runs and produces sensible output by requesting an autotest online. Log into `https://teaching.doc.ic.ac.uk/labts` and look for the **531_Formulas** exercise. On its repository page, you will see the commits you have pushed, buttons for requesting autotests, and a button to "Submit to CATe". When you request an autotest your code will be loaded and a small set of tests will be executed. It may take a few minutes to give you back a result. 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. However, you will be able to check that your code was loaded and executed without errors. (In particular look at the log output as well as the test results.) The test results are at the end of the resulting pdf file. The pdf will also include your code, so you can make sure that it is clearly laid out and readable when it is printed. The full tests used after submission cover other cases outlined in the description above. So, a program that meets the exercise specification will pass all the tests.

Part 3: Submitting

Submit By 23 October 2018

Submission is entirely electronic — you do not need to submit hard copy.

Stage 1: Push To Gitlab

First you need to use Git to add, commit and push the changes you have made back to the Gitlab server. The autotester will be cloning this repository for testing.

1. Using the technique described in Section 1.3, you need to use `git add`, `git commit` and `git push` to send your modified files back to the Gitlab server.
2. Once this is done, you should check that you really have committed everything. You can do this by checking that `git status` reports everything is ok:

```
> git status
# On branch master
nothing to commit (working directory clean)
```

You can also check the state of your git repository using e.g. `git log`, or a repository browser such as `gitk` or `gitg` (try running these commands in a terminal that has `cd`'d into the `531_formulas_`*`login`* directory).

3. Check the state of your GitLab repository using the GitLab webpages:

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

   You can inspect the list of commits you have made and the contents of individual files.

Stage 2: Submit to CATE

1. Go to the LabTS website:

   `https://teaching.doc.ic.ac.uk/labts`

   and navigate to the page for your repository.

2. Select the version of your code you want to submit as your solution and click the appropriate **Submit to CATe** button. You can verify the submission has been made by looking at the exercise in CATE.

3. You will be taken to the submission page for the exercise in CATE, which will be pre-populated showing the commit you are submitting. Complete the declaration and confirm your submission.

4. The submission can be updated as many times as you like between the start and end of the exercise. It is advisable to make sure you have submitted *something* well before the deadline. This initial file can be overwritten and will be better than missing the deadline altogether.

Part 4: Assessment and Feedback

Your solutions will be marked on correctness, code design and readability. You should test your program on a range of suitable examples before submission. Do not overcomplicate the solution!

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 procedures, 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.

This exercise is worth 15% of the marks allocated for coursework on this module.

Return of Work and Feedback

This exercise will be marked and returned by **6th November 2018**. Feedback on your solution will be given on the returned copy, and feedback will be given to the class as a whole in the support lectures.