Declarative Programming Summative Assignment '21-'22
Second Session

A System for Linear Programming in English

1 Introduction

This practical is the second formally assessed exercise for MSc students on the Declarative Programming course. The intent is to implement an AI system for solving linear equation sets, derived from natural language input.

This practical counts for 30% of the marks for this course. In itself, it is marked out of 100, and the percentage points available are shown at each section.

This exercise uses ideas from the sections of the MSc Prolog course in definite clause grammars (DCGs), meta-programming, constraint logic programming over integer finite domains (CLP(FD)), and constraint meta-programming.

The exercise is in four parts:

1. The design and construction of a Definite Clause Grammar (DCG) in Prolog which will analyse a small subset of English used to describe mathematical equations (30%);
2. The interfacing of this DCG to the Finite Domains equation solver built into Prolog (this is not as hard as it sounds!) (35%);
3. Testing of your software to make sure that it covers only grammatically correct sentences (20%);
4. Building a small program to check the results of the deduction, reporting on, for example, whether or not there is a unique solution. (15%)

2 What is Linear Programming?

Linear programming is a means of finding a solution to sets of linear equations and inequations (formulae with $>$ or $\leq$ in the middle) which is normally done by plotting the different equations on a graph and looking to see which area(s) they enclose.

For example, if we had the equations

$$y = 2x$$

and

$$y = 4 - 2x$$

we can draw a graph of the two lines so described, such as that in Figure 1, and conclude that any simultaneous solution to both equations must lie at the point where the lines cross. So when there are only strict equations in the set of formulae the answer is a point, from which the x and y values can be read off; when inequations are included, the result can sometimes be an area of the graph, indicating a set of points any or all of which could be the answer.

It so happens that SWI Prolog' Finite Domains Constraint Solver is capable of doing this kind of reasoning without drawing graphs. Indeed, it can go further, and describe systems with arbitrarily many variables, which we could not draw on a graph because we only have two dimensions to work

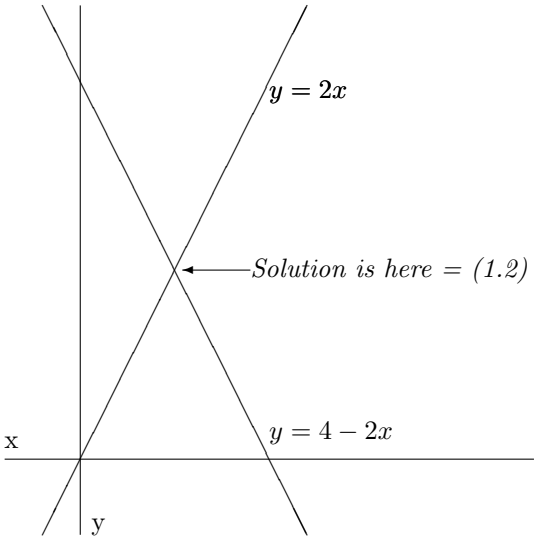

Figure 1: Linear programming example

in. Note, though, that this particular solver only works with integers, and not with floating-point numbers.

3 Input Syntax

The following are examples of the input syntax of the system. This constitutes the *minimum* coverage of the parser. If you wish to extend it further, do make sure that it covers at least the sentences and the kinds of reference below.

> The variable x lies between 0 and 10.
> Variable x varies from 1 to 20.
> A variable x is in the range 100 to 14.
> y is between -2 and 25.
> x equals a plus b.
> x is c times 2.
> The variable z contains the product of b and c.
> y is less than 5 + 2 * q.
> All these variables are greater than 5.
> Variable q is greater than or equal to the quotient of z and 2.
> Variable w holds the dividend of z and 2.
> It is greater than q.

Here are some notes on the sentences:

- The scope of the variable names is the entire set of input sentences – that is, once a variable is named, any use of its name refers to the same variable.
- The referring expression *It* refers to the variable which was the subject of the previous sentence, unless the previous sentence began with *All*, in which case an exception is generated (see the manual for details).
- The referring expression *All these variables* refers to the set of variables which have been named up until its occurrence in the input string. It does *not* refer to any variables named after it in the input string.

- The noun phrase *A variable x* always introduces a *new* variable called x; using it twice with the same variable name should generate an exception, because this would cause a scoping error.

Do not forget that mathematical expressions are not simply evaluated in left to right order – you have to allow for the fact that +/- and × and ÷ bind their arguments differently.

Make any assumptions which you suppose in designing your grammar clear in your comments.

4 NLP Methodology

Remember, when you are designing your parser, that the point of parsing is to find common patterns in the language and take advantage of them to provide generality. For example, a single DCG rule which matches the whole of the sentence "The man bites the dog." is not very useful because it can only match one sentence. Much more useful is to split up the sentence into a noun phrase ("The man") and a verb phrase ("bites the dog") because the rule that does this is very general, covering many English utterances.

So a good solution to the linear programming problem will break down the sentences of the input into significant lumps, for example classing "lies between ... and ...", "varies from ... to ..." and "is in the range ... to ..." all together; similarly, for things like "+" and "plus".

The best way to achieve these groupings is to list them all out and decide which mathematical symbol they match with; then you can write a set of rules which cover each symbol, methodically. Here is an example of some rules which might be used to cover "+" and "plus". Note that you would need to extend them with some extra arguments to make them work for you in the assignment.

```
expression -->           value,plus_symbol,expression.
expression -->           [the,sum,of],expression,[and],expression.

plus_symbol --> [plus].
plus_symbol --> [+].
```

5 Using the Parser

Once you have written out the rules that the parser will use, you need to be able to pass back the various expressions, including variable names. You will need to find a way to take an expression like "a plus b" and convert it into `_1345 + _4566` where `_1345` and `_4566` are Prolog variables which correspond with a and b. You will need to keep track of the variable names which have been used, and create a new Prolog variable for each one. This will mean keeping a list of all the variables used, and passing it around as your parser works. *Hint: remember that you can add extra arguments to DCG clauses just the same as in ordinary Prolog clauses, and pass values around in the way that you're used to in ordinary Prolog.*

There will be some situations where you need to use meta-programming predicates to test or create the sub-expressions which you work will. However, a good implementation will only do so when strictly necessary – normally, unification will suffice.

Once you have got the meaning of the sentence you're parsing, you can use the Prolog meta-predicate `call/1` to add it to the constraint system.

6 Designing the Program

Your program will consist mostly of DCG clauses (the ones with `-->` instead of `:-`). It will have some bits of Prolog to be executed independently of the parser, enclosed in {}. To give you a bit of a start, here's how the very top level predicate will look (i.e., put this in your source code!):

```
% front end:  lpsolve/2 succeeds when its first
% argument is a list of well formed sentences
% and its second is a list of constrained variables

lpsolve(  Data, Answers ) :-
          lpsolve( [], Answers, Data, [] ).

% lpsolve/4 succeeds when its first argument is a
% list of variable name/Prolog variable pairs, its
% second is the same list modified with any new
% variables introduced in the list of sentences
% in argument 3 which is parsed with the remainder
% given in argument 4

lpsolve(  VariablesIn, VariablesOut ) -->
          sentence( VariablesIn, VariablesOut ).
lpsolve(  VariablesIn, VariablesOut ) -->
          sentence( VariablesIn, VariablesBetween ),
          lpsolve( VariablesBetween, VariablesOut ).
```

Note that in the first of these clauses, we are passing two `[]`s to the DCG: the first to indicate that we start with no variables in the system (obviously, since we have read no sentences yet); the second to indicate that we want no symbols left after our data has been analysed.

It is left for you to work out what the structure of the grammar is. Try to work in the abstract, without thinking about Prolog to start with – just think about what different kinds of sentences your grammar has to cover, group them together, create names for the groups, and write them down, as above. For example, we can read the DCG rules for `lpsolve` above as

> "We can solve the set of equations if it consists of just one equation which we can solve, or if it consists of one equation which we can solve followed by some more."

You will need an auxiliary predicate, `look_up/4`, which succeeds when its first argument is a variable name, its second is a list of variable-name/Prolog-variable pairs, its third is the same list, with a new pair in it if the name hasn't been used before, and its last is the Prolog variable which corresponds with the name. Note that you should restrict your variable names to be one letter long to avoid confusion within the parsing process.

When you are working on this code, it may be useful to check up on the `clpfd:full_answer` switch, described in the SWI Prolog manual, section "Answer Constraints".

7 Testing Your Code

DCGs take lists of symbols as input. Remembering that everything has to start with a lower-case letter, you can set up your test data in a file as a predicate called, for example, `test/1`, whose argument is a list containing the words in your data, using the symbol `fullstop` to indicate the end of a sentence. So your test predicate might look something like this:

```
test(  [the,variable,x,is,between,a,and,10,fullstop,
        q,is,less,than,v,fullstop,
        ...,
       ] ).
```

Then, when you have written your solver, which should be called `lpsolve/2` as above, you can call it as follows:

```
test( Data ), lpsolve( Data, Variables ).
```

As part of this assignment, you should test your program on this set of sentences:

> The variable x lies between 0 and 10.
> Variable y varies from 10 to -10.
> A variable z is in the range 0 to 15.
> It equals x plus y.
> All these variables are greater than -20
> y is less than 5 + 2 * x.
> x is greater than y times 2.
> Variable y is greater than or equal to the quotient of z and 4.

Also include in your submission a short description of how you tested your program, including the tests being applied and the test results. The test results should be commented out so the file compiles. Discuss how you know that the test results are correct.

8 Analysing the Results

Write a predicate, `analyse/2`, which always succeeds when its first argument is a list of variable name/Prolog variable pairs, as in `lp_solve/4` and its second is a list in the format output by `call_residue/2`. It has the following side-effects:

- When any variable's domain is not finite, it prints out that there may be an infinite number of solutions;
- Otherwise, if there is a unique solution, it is printed out in the form of a sequence of lines of the form `a = N` where `a` is a variable name, as input (*i.e.,* not a Prolog variable name), and `N` is the value of `a`;
- Otherwise, any variables which have unique values (*i.e.,* whose domain has only one member) are printed out as above; the domains of the others are printed in standard Prolog notation; finally any surviving constraints are also printed out, in terms of the input variable names.

9 Documentation, Style, and Testing

Your code should have concise documentation in the comments for most predicates. This means you should describe the argument modes, like input (`+`), output (`-`), both (`?`), etc., (see https://www.swi-prolog.org/pldoc/man?section=preddesc), and a one or two line description of the predicate. There is no need for a paragraph description for each predicate. For example:

```
% append(?List1, ?List2, ?List1List2)
%   List1List2 is the concatenation of List1 and List2.
append(List1, List2, List1List2) :- ...
```

If a predicate has many different clauses, or the clauses are especially complex, it is also good to describe what each clause does separately, but this is not necessary in general. It is also helpful to indicate larger sections of the code that go together with a header, e.g., Definite Clause Grammar, Constraint System, Testing, etc.

Not only should your code conform to standard best practices of software engineering, like keeping predicates small, atomic, and non-redundant, but **your code should be written in declarative, idiomatic Prolog** as much as possible. This means taking advantage of unification and backtracking to leverage the built-in search in Prolog. If you find yourself frequently using `findall/3`, then rethink how you are approaching the problem to try to use backtracking and/or negation instead.

A common *faux pas* we often see is using a bunch of nested lists to hold data instead of using named tuples. For instance, when representing a line segment with two xy-coordinates, instead of

something like `[[0,1],[2,3]]`, you should use something like `segment(pair(0,1), pair(2,3))`. Not only will your code be much more readable, but you will better be able to leverage unification in your solution.

Avoid using cuts (`!`) and if-thens (`->`). Using these actually means your program is logically incorrect. So, unless you provide a very good reason, you will be penalized for their usage. If you think you need to use a cut to stop unwanted backtracking, you can usually solve this by making your clauses mutually exclusive. If you think you need to use an if-then, you can usually just split it into separate clauses instead.

Your program should have also good test coverage in order to be confident that it behaves as desired. This means there should be decent amount of unit tests for the most important predicates (you don't need to unit test every single predicate) and integration tests that test larger and larger portions together. The best way to do this is to write test predicates that call your other predicates to ensure they are working. This may also mean writing your own markup documents to test different layouts. At the very least, your code should correctly handle the provided poster markup examples. Please put these predicates at the bottom of the source code; do not interlace them throughout.

10 Submission

You should combine together whatever source code you develop for the different parts of the exercise into a **single file** and submit it **via Canvas by 29/08/2022**. During your testing, you may decide to create a few test markup files, which should also be submitted along with the source code. Of course, these can remain separate files.

Please ensure that you place any text in your code inside comment markers, so that the file will load without further editing. You will be penalized if you fail to do this. Also ensure that the program loads without errors and warnings. The submission must be made by the advertised deadline.

11 Referencing and Cheating

When programmers are stumped on a bug or issue, we often go straight to Google and adapt the resulting hits ot our needs, often directly from StackOverflow. Being able to specify and solve your issues in this way is often useful, especially when working in the industry. However, in the academic setting, this makes it difficult to not only evaluate your competency as a Prolog programmer, but also to identify which code is actually yours! Therefore, if you need to adapt code from the web, simply provide a link to that resource (i.e. URL) along with a one-line description of where it's from and what it is. For example,

```
% StackOverflow: All combination of the elements of a list
% https://stackoverflow.com/questions/41662963
combs([], [])
... and so on.
```

Doing this whenever you take code from online removes any suspicion of cheating. **Any failure to do so will be regarded as cheating!** In addition, **you cannot share code**; therefore, you cannot reference another student who has taken or is currently taking the course. There are a few places where refering in this manner is not required:

- Lecture slides (Encouraged!)
- SWIPL built-ins, libraries, and documentation (Encouraged!)
- Non-code or other educational resources (e.g. an algorithmic description from Wikipedia)