TDDD08 Logic Programming
Lab instructions

2018/09

Version 1.2

Ulf Nilsson, Fredrik Eklund,
Andreas Kågedal, Pawel Pietrzak
Victor Lagerkvist, Włodek Drabent
Jonas Wallgren
Linköpings Universitet, IDA
Kjell Post
Mälardalens Högskola, IDt

Contents

1 General Information **3**

1.1 To Write Prolog Code 3
1.2 Comments 4
1.3 Handing In Solutions 5
1.4 Additional Information 6

2 To use SICStus Prolog **7**

2.1 Plain SICStus Prolog 7
2.2 The Emacs Interface 9
2.3 Manual 11

3 Lab Exercises **12**

Lab 1 Databases 12
Exercise 1.1 A Small Deductive Database 12
Exercise 1.2 Recursion 13
Lab 2 Recursive Data Structures 13
Exercise 2.1 Sorting 13
Exercise 2.2 Search Strategies 14
Exercise 2.3 An Abstract Machine 14
Exercise 2.4 Set Relations 15
Lab 3 Definite Clause Grammars 16
Exercise 3.1 Parser 16
Lab 4 Search 17
Exercise 4.1 Breadth-first Search 18
Exercise 4.2 Depth-first Search 18
Lab 5 Constraint Logic Programming 18
Exercise 5.1 Scheduling 18

Chapter 1

General Information

This chapter contains general information that will help you to complete the lab course successfully. It is important that you read chapters 1–2 *before* you start doing the exercises. In general it is also important that you are well prepared before coming to the labs. It is usually required that you have read parts of the course book.

1.1 To Write Prolog Code

There is no generally accepted way of writing Prolog programs but it is quite common to write clauses of the form:

```
nsort(Xs, Ys) :-
    perm(Xs, Ys),
    sorted(Ys).
```

That is, the head of the clause and each premise of the body is written on a row of its own. The premises are indented (e.g. with a tab character). Normally all clauses defining a predicate are kept together in the program file:

```
perm([], []).
perm([X|Xs], Ys) :-
    perm(Xs, Zs),
    insert(X, Zs, Ys).
```

The SICStus Prolog compiler (see next chapter) sometimes emits *warnings*. In particular, the compiler may complain about clauses that contain "singleton variables". This happens when a clause contains a variable that has only one occurrence. The reason why the compiler emits a warning is that a “singleton variable” is often the result of misspelling a variable name. In cases where you do want to write a variable in just one place in a clause the warning can be avoided by prefixing the variable name by an underscore. For instance, even if the variables `_X` and `_List` occur only once in a clause the compiler will not emit any “singleton variable warnings”. Sometimes you will not be interested in giving these singleton variables a name at all. In such a case you can use an “anonymous variable” which is written “`_`” (i.e. with a single “underscore” character). Every occurrence of “`_`” is regarded as a separate, new variable. So `p(_,_)`, p(X,Y) are basically the same formula, only with the variables renamed.

In your programs, you are not allowed to use non-logical built-ins of Prolog, like `var`/1, `\==`/2, etc, unless explicitly stated otherwise. Any possible exception should be well justified and preferably consulted with us. You are not allowed to use the cut (`!`/0). In most cases you should avoid to use the negation (`\+`/1, `\=`/2, `nonmember`/2, etc, this includes the Prolog if-then-else construct, `->`). Use `dif`/2 when you need an inequality predicate.

1.2 Comments

Regardless of programming language it is important to annotate the code with comments. In Prolog comments can be written in two ways:

- Everything that occurs between `%` and the end of the line is a comment.
- Everything that occurs between `/*` and `*/` (including end of lines) is a comment.

It is not always necessary to explain *how* a procedure/predicate works, but you should *always* explain *what* a predicate does, i.e. the relation that is described (or the *declarative semantics* of the predicate). A good idea is to put this on a few lines just before the predicate definition: An example is provided in Figure 1.1.

The comments do not only provide a way to document the code, they can also help us to convince ourselves of the correctness of the program. For instance, the second clause in `perm/2` says that “if `Zs` is a permutation of `Xs` *and* `Ys` is the list `Zs` with `X` inserted somewhere *then* `Ys` is a permutation of `[X|Xs]`”, something that seems reasonable.

```
% perm(Xs, Ys)
% The list Ys is a permutation of Xs
perm([], []).
perm([X|Xs], Ys) :-
    perm(Xs, Zs),
    insert(X, Zs, Ys).

% insert(X, Ys, Zs)
% If Ys is a list then Zs is a list and
% Zs is Ys with X inserted at some position
insert(X, Ys, [X|Ys]).
insert(X, [Y|Ys], [Y|Zs]) :-
    insert(X, Ys, Zs).
```

Figure 1.1: Example of comments describing the declarative semantics of predicates.

1.3 Handing In Solutions

You may hand in solutions to exercises incrementally or in one batch, but the first is preferable. You can also demonstrate your solutions directly to the lab assistant in the computer rooms. However delivering them on paper is preferable. When you hand in a solution to an exercise it must contain:

- A readable listing of the *commented* code according to the directives above. *Uncommented code will not be accepted.*

- Several test runs that make probable that the program works. Remember that a test run should show all the answers to each query (unless there are really many).

  Your programs are required to work with queries corresponding to the problem description. Assume, for instance, that the problem is sorting lists of numbers. Consider a query which, instead of a list to be sorted, contains a variable or a list with non-number elements. For such query the program is allowed to loop or produce run-time errors

In some of the exercises it is also required that you hand in additional material (see individual exercises). *Note that it is not sufficient to hand in a program that computes the right answers* – the solution must also be reasonable – readable and avoiding substantial inefficiencies (both in time and space).

You must be able to explain your program. When you provide a corrected solution after having received written comments, you should also hand in the previous, commented version

In order for us to be able to register your results it is required that every handed in solution to a lab exercise is accompanied by a cover page with the following information:

- Name, personal number, email.
- Course name.
- The number and name of the exercise.

Without this information we may have trouble registering your results.

Note that there is a deadline for handing in solutions to the exercises! If you do not meet this deadline (see the course web) you will be given two more chances to finish the lab course – in connection with the re-examination for the course. However, we may not grade or report results in between the re-examinations. Note also that if you do not finish the whole lab course within the year you will have to take next year's lab course in its entirety (which might differ from the one this year).

1.4 Additional Information

Additional information and course material is available on the web at the following URL:

`http://www.ida.liu.se/~TDDD08/`

Chapter 2

To use SICStus Prolog

The Prolog system that will be used for the laboratory classes is called SICStus Prolog. It can be used in several ways. The most primitive is to use the system directly in a “shell” (e.g. `csh` or `bash`). A better alternative is to use the Emacs interface which is provided. This chapter will give an introduction to the use of the SICStus Prolog environment and explain how to access the on-line documentation. However, the first step is to import the right module. To do this write

```
module add prog/sicstus
```

or

```
module initadd prog/sicstus
```

(to avoid having to type this every time you log in). For up-to-date information, see the course home page.

2.1 Plain SICStus Prolog

To start SICStus Prolog with a minimal user interface you write `sicstus` at the shell prompt:

```
$ sicstus
SICStus 4.3.0 (x86_64-linux-glibc2.12): Thu May  8 05:34:25 PDT 2014
Licensed to SP4.3ida.liu.se
| ?-
```

The Prolog prompt has the form “`| ?-`” and says that the Prolog system is ready to accept the users commands. Among other things, we could now compile and load Prolog programs. Assume for example that in the current directory we have a file `married.pl` containing the following description of who is married to whom:

```
married(adam, eva).
married(kungen, silvia).
married(fantomen, diana).
```

The file can now be loaded into the Prolog system with the command:

```
| ?- [married].
```

If the file name is not a Prolog "atom", use apostrophes:

```
| ?- ['mine/a.pl'].
```

Note that you do not need to include the "`.pl`" in the file name. For this reason *always* use "`.pl`" as the extension of all Prolog files. This will also be useful if you use one of the more advanced interfaces. Also note that the command (like all commands) must be terminated with a period "`.`" and a carriage return.

To run your program you type in a *query* at the prompt. If you for instance want to see "who is married to whom" you can write:

```
| ?- married(X,Y).
```

The Prolog system presents the first solution it finds:

```
X = adam
Y = eva ?
```

Type a semicolon ("`;`") and then carriage return, to obtain a next solution. In our case:

```
X = kungen
Y = silvia ?
```

and after yet another semicolon:

```
X = fantomen
Y = diana ?
```

When the Prolog system cannot find any more solutions it just answers:

```
no.
```

A carriage return without a semicolon would finish answering the current query.

There is little support for changing and saving programs directly in SICStus Prolog, so edit your program file using another window and then reload the program again.

To permanently leave SICStus Prolog you write the command:

```
| ?- halt.
```

2.2 The Emacs Interface

You can run Prolog inside GNU Emacs and load files directly from another buffer.[1] In order for this work you have to add a few lines to your `.emacs` file. The following should do the trick:

```
(setq prolog-system 'sicstus)
(setq auto-mode-alist
      (cons '("\\.pl$" . prolog-mode) auto-mode-alist))
```

The code can be downloaded from the course home page. When you have saved the new version of `.emacs`, exit and restart Emacs for the changes to take effect.

If you load a file with the suffix "`.pl`" into Emacs, it will print "`(Prolog)`" in the black line above the mini-buffer of the Emacs window. This indicates that Emacs is in *Prolog mode*. (If this does not work, please contact the lab assistant.)

Now restart Emacs and create a file named `append.pl` containing the following clauses:

```
app([], Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs, Ys, Zs).
```

(We do not use `append` as the name of the predicate, as SICStus does not allow to redefine built-in `append`/3.) The Emacs window should now look approximately as in Figure 2.1.

To send the buffer to the Prolog system you type "`ctrl-c`" and then "`ctrl-b`". (You can also select `Consult Buffer` in the Prolog menu.) The Emacs window is now divided in two parts with the buffer containing the program in the upper part and a new buffer – where the Prolog system is running – in the lower part. When you typed `ctrl-c ctrl-b` the program was loaded (or "*consulted*") into the Prolog system. (If there is no Prolog system running in Emacs – as in our case – Emacs first starts a new Prolog system.)

[1] This information may or may not apply to other versions of Emacs such as Xemacs.

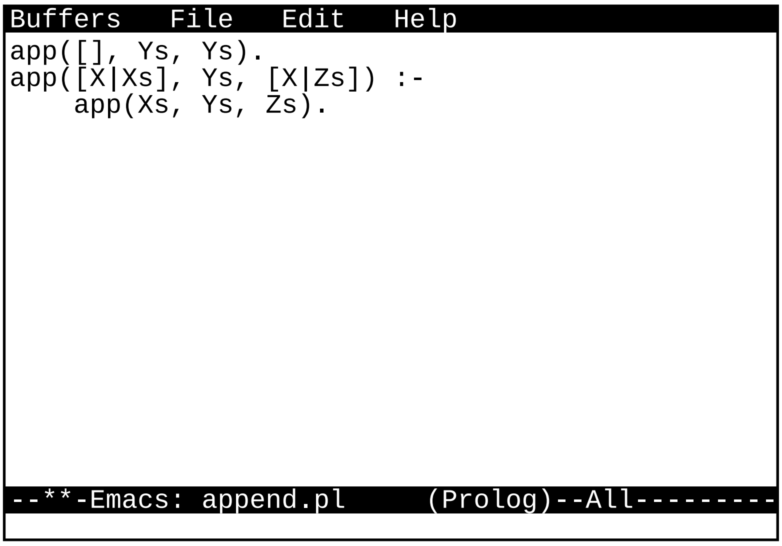


Figure 2.1: Emacs in Prolog mode.

To move between the upper and lower buffer you either click with your mouse or type "`ctrl-x`" and then "`o`".

Now move to the buffer containing the Prolog system and type in the following query:

```
?- app([a,b,c], [1,2,3], L).
```

followed by a carriage return. The Emacs window should now look approximately as in Figure 2.2.

Prolog programs can be loaded into the SICStus Prolog system in two ways, either by loading the code for interpretation, *consulting* it, or by compiling it. Compiled code is more efficient but if you want to debug your program or use the trace facility you must consult the program.

The following is a summary of the Emacs commands that can be used in the upper buffer (the one containing the program).

`ctrl-c ctrl-r`	consult the region.
`ctrl-c ctrl-b`	consult the buffer.
`ctrl-c ctrl-p`	consult a predicate.
`ctrl-c ctrl-c r`	compile the region.
`ctrl-c ctrl-c b`	compile the buffer.
`ctrl-c ctrl-c p`	compile a predicate.

```
Buffers   File   Edit   Help
app([], Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs, Ys, Zs).



--**-Emacs: append.pl     (Prolog)--All---------
SICStus 4.3.0: Thu May  8 05:34:25 PDT 2014
Licensed to SP4.3ida.liu.se
| ?- % consulting /tmp/append.pl...
% consulted /tmp/append.pl in module user, 10ms
yes
| ?- app([a,b,c], [1,2,3], L).
L = [a,b,c,1,2,3] ?

--**-Emacs: *prolog*      (Comint: run)---------
```

Figure 2.2: The source code (above) and the Prolog system (below).

Obviously, you can also consult files by writing:

```
| ?- [married].
```

2.3 Manual

The users manual for SICStus Prolog is accessible on-line. See the course home page for links.

Chapter 3

Lab Exercises

Lab 1 Databases

The objective of the following exercise is to get acquainted with the SICStus Prolog environment. You will practice writing and compiling programs and execute your programs by posing queries to the system. It is important that you prepare yourself by reading the previous chapters. Also, take the opportunity to try out the debugger and take a look in the manuals.

Exercise 1.1 A Small Deductive Database

Translate the following scenario to *definite clause form*:

1. Ulrika is beautiful.
2. Nisse is beautiful and rich.
3. Anne is rich and strong.
4. Peter is strong and beautiful.
5. Bosse is kind and strong.
6. All men like beautiful women.
7. All rich persons are happy.
8. All men who like a woman who likes him are happy.
9. All women who like a man who likes her are happy.
10. Nisse likes all women who like him.

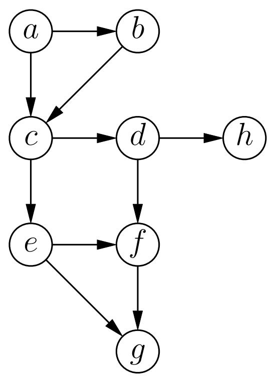
11. Peter likes everyone who is kind.

12. Ulrika likes all men who likes her, provided they are either (1) rich and kind, or (2) beautiful and strong.

Use the program to answer the questions "Who is happy?" and "Who likes who?". Also make the program answer the question "How many persons like Ulrika?". For this query you probably need to use the built-in predicates `findall/3` and `length/2` about which you can read in the manual.

Explain your choice of the order of the clauses in your program, and the order of premises in the rules.

Exercise 1.2 Recursion



Consider the graph to the right. Represent the graph as a set of facts. Then write a predicate `path/2` which relates two nodes to each another if and only if there is a path from the former to the latter in the graph. Then first write a predicate `path/3` that describes the relation between two nodes and also records the path between them, and second a predicate `npath/3` which describes the relationship between two nodes and also returns the length of the path between them. Hint: use the predicate `path/3` and the built-in predicate `length/2` for finding the length of a list. Your program is required to work with arbitrary graphs without cycles.

Lab 2 Recursive Data Structures

The purpose of the following exercises is to study how to manipulate recursive data structures in Prolog and to get a feeling for how the search strategy of Prolog affects the structure of the program. You are not allowed to use built-in or library predicates defining list operations, except for `member`/2, `append`/3.

Exercise 2.1 Sorting

Write a predicate `issorted`/1 checking whether a given list of numbers is sorted in the ascending order.

Write two recursive sorting programs, `ssort/2` and `qsort/2`, that sort lists of integers according to the following two principles:

- **Selection sort.** Transform a given list L into a list $L1$ with the same elements, but with the first element being the smallest one. The first element of $L1$ is the head of the sorted list LS to be obtained, the tail of LS is obtained by sorting the tail of $L1$.
- **Quicksort.** Pick an integer N that occurs in the list that should be sorted (for instance the first element in the list) and partition the list in two lists – one that contain all elements less than N and the other with the rest of the elements. Sort the lists and concatenate the resulting (sorted) lists.

Use the built-in predicates `</2` and `>=/2` to compare integers.

Exercise 2.2 Search Strategies

Consider the recursive predicate `middle/2` below:

```
% middle(X,Xs)
% X is the middle element in the list Xs
middle(X, [X]).
middle(X, [First|Xs]) :-
    append(Middle, [Last], Xs),
    middle(X, Middle).
```

Examine how the execution of the following queries is affected by the ordering of the clauses and the premises within clauses.

```
| ?- middle(X, [a,b,c]).
| ?- middle(a, X).
```

Try all four possible orderings and (1) explain what happens, (2) why and (3) sketch the SLD tree (you don't have to draw the whole tree...). Don't forget to explain what happens when you ask for more than one answer! Which version is preferable for each type of query?

Exercise 2.3 An Abstract Machine

In this exercise you will write an interpreter for a small imperative programming language which we will call IMP. We will assume that the language is given by the following abstract syntax. We first assume the existence of primitive symbols

```
I - Identifiers
N - Natural numbers
```

A *binding environment* is a mapping from identifiers to natural numbers and is an abstraction of a memory. A binding environment can be represented as a list $[i_1 = n_1, \ldots i_k = n_k]$ where identifier i_1 has the value n_1, identifier i_2 has the value n_2 etc. IMP also has the following language constructs

```
% Boolean expressions
B ::= tt | ff | E > E | ...
% Arithmetic expressions
E ::= id(I) | num(N) | E + E | E - E | E * E | ...
% Commands
C ::= skip | set(I, E) | if(B,C,C) | while(B,C) | seq(C, C)
```

Here `tt` and `ff` are constants for Boolean true and false, `skip` is a command that does nothing, `set(I,E)` is a command that assigns the value of `E` to the identifier `I`, and `seq(C,C)` means the sequential execution of the first argument followed by the execution of the second argument.

Choose a suitable Prolog representation of identifiers and natural numbers. Then define the semantics of IMP by first defining predicates describing the values of arithmetic and Boolean expressions, and then a predicate `execute`/3 describing the semantics of commands. Use Prolog arithmetic to evaluate arithmetic expressions. Predicate `execute`/3 should relate an initial binding environment `S0` and an IMP program command `P` with the final binding environment `Sn` that is the result of executing P. For example, if `S0` is `[x=3]` and `P` is the program

```
seq(set(y,num(1)),
    while(x > num(1),
        seq(set(y, id(y) * id(x)),
            set(x, id(x) - num(1)))))
```

then the relation should hold if and only if `x` is bound to 1 and `y` is bound to 6 in `Sn`.

Exercise 2.4 Set Relations

Prolog defines an order relation on all terms. The built-in (and infix) predicates `@<`, `@>=`, etc can be used to compare terms according to this order (see the manual for more information). Assume now that we want to represent sets of ground terms as ordered lists without copies. The set `{c,b,a}` is then represented with the list `[a,b,c]`. Write a Prolog procedure that computes the union of two sets. The same for intersection. Also write a procedure that produces the powerset of a set, e.g., using the above set we get the powerset:

$$\{\emptyset, \{\mathtt{a}\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{c}\}\}$$

represented by list `[[],[a],[a,b],[a,b,c],[a,c],[b],[b,c],[c]]`. (Remember that the resulting lists are to be sorted and without repetitions.)

Note: In the last case it might be a good idea to use the built-in predicate `findall/3`.

Lab 3 Definite Clause Grammars

In the following exercises you will write a "front end" to the IMP interpreter in the previous exercise. The program will consist of two parts: a *scanner*, that takes a string of characters and produces a list of *tokens*, and a *parser*, that transforms the list of tokens into a term that can be executed by the interpreter together with an initial binding environment. Hence, the idea is the following:

```
run(In, String, Out) :-
    scan(String, Tokens),
    parse(Tokens, AbstStx),
    execute(In, AbstStx, Out).

| ?- run([x=3],
         "y:=1; z:=0; while x>z do z:=z+1; y:=y*z od",
         Res).
```

The result `Res` should be a binding environment where `y=6` (that is, the factorial of 3).

Exercise 3.1 Parser

In this exercise you will write a parser (in the form of a DCG) which recognizes the following strings of tokens:

```
<pgm>   ::= <cmd>
         |  <cmd> ; <pgm>
<cmd>   ::= skip
         |  <id> := <expr>
         |  if <bool> then <pgm> else <pgm> fi
```

```
          |  while <bool> do <pgm> od
<bool>   ::= <expr> > <expr>
          |  ...
<expr>   ::= <factor> * <expr>
          |  <factor>
<factor> ::= <term> + <factor>
          |  <term>
<term>   ::= <id>
          |  <num>
```

The parser should return the corresponding abstract syntax object. That object will then be used as input to the abstract machine previously. To make the problem somewhat easier we assume that all operators are right-associative. (This is fine with associative operators such as `+` and `*`, but yields non-standard results e.g. for `-`.)

To make the job somewhat easier you may use an existing Prolog scanner which translates a list of ASCII-characters to a list of tokens (which can be given to the parser). The scanner can be downloaded from the course home page. Play around with the scanner before attempting to write the parser. (In Prolog you may write strings in the form `"Prolog"` but this is only syntactic sugar for a list of integers corresponding to the ASCII values of the string.)

Lab 4 Search

Three missionaries and three cannibals are standing on the bank of a river. There is a boat but it is only big enough to carry two persons. If the cannibals, at any time, outnumber the missionaries on either side of the river the cannibals will eat the missionaries. We now want to find a way to transport all missionaries and cannibals over the river without having any of the missionaries eaten.

Thus, we have a *search problem* – a set of states and transitions between them. The problem is to find a sequence of states which takes us from the initial state to a final state without passing an illegal state (a state where the cannibals outnumbers the missionaries on a bank of the river). You will write two Prolog program which, using two different search strategies, helps the missionaries and cannibals to get over to the other side of the river without anyone being eaten.

The programs should display the solution (i.e. the sequence of states) in a readable way. (See the SICStus Prolog manual for printing built-ins.)

As both programs solve the same problem, they should produce the same solutions, maybe in different order.

Exercise 4.1 Breadth-first Search

Write a program that searches breadth-first in the state space. Note that there are infinitely many solutions to the problem. The program can therefore stop after finding the first (i.e. shortest) solution. However, the program should be able to enumerate all possible (loop-free) solutions to the problem by means of backtracking.

Clue. See pages 185–186 in the course book.

Exercise 4.2 Depth-first Search

Write a program that searches depth-first in the state space. Note that the space contains “loops” and that it is necessary for the program to detect these and break them. The program should be able to enumerate all loop-free solutions by means of backtracking. How many are there?

Clue. See pages 181–182 in the course book.

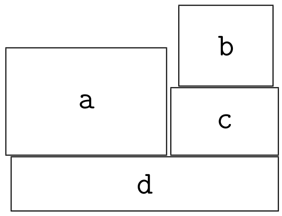
Lab 5 Constraint Logic Programming

Exercise 5.1 Scheduling

A ship carrying a number of containers has arrived to a port and all of the containers are to be unloaded. Assume that we have a database describing these containers by means of a predicate `container(B,M,D)`, where `B` is a container's identifier, `M` is the number of persons required to unload the container and `D` is the duration of unloading (`D` is assumed to be integer valued). A sample database may thus look as follows:

```
container(a,2,2).
container(b,4,1).
container(c,2,2).
container(d,1,1).
```

Additionally some containers may have been put on top of others, for instance:



The fact that a container is put on top of another is expressed by the predicate `on(B1,B2)`. For the containers placed as shown in the picture, the database will contain the following facts:

```
on(a,d).
on(b,c).
on(c,d).
```

Hence, the container `d` cannot be drawn before containers `a` and `c` and so forth.

Design a CLP(FD)-program that for *any* database consisting of definitions of the predicates `container/3` and `on/2` minimizes the cost of unloading the containers. The cost is defined as the number of workers hired times the required time (in whole hours); and where all workers are hired for the whole duration of the unloading. The result should show (at least) the cost, the number of workers and the time.

You should use the global constraint `cumulative/2` to solve the problem. Do not forget to load the finite domain solver library of SICStus by the directive `:- use_module(library(clpfd))` at the beginning of your program.