TDDD08 — Tutorial 1

Who? Victor Lagerkvist (& Włodek Drabent)

From? Theoretical Computer Science Laboratory, Linköpings Universitet, Sweden

When? 6 september 2015

Preparations

Before you start with the labs.

- Register yourself in Webreg to participate in the labs. Deadline 24th September.
- Read the lab instructions.
- Add module file (use module [init]add prog/sicstus).
- Modify your .emacs (see course page).

Strict deadline for completing the labs is 30th October.

Preparations

Using the system.

- Separate code (filename.pl) and query (*prolog*) buffers: all facts and rules in the file-buffer.
- Just open or create filename.pl to enter prolog mode automatically.
- Save file and press C-c C-b to “consult buffer” and create the query window.
- Quit command: “halt.”.

The labs

Five labs in total.

- Lab 1: Basic Prolog.
- Lab 2: Recursive Data Structures.
- Lab 3: Definite Clause Grammars.
- Lab 4: Search.
- Lab 5: Constraint Logic Programming.

Basic Prolog syntax

- **Functors** (function symbols), **predicate symbols**: "atoms" in Prolog, lower case letter first (a, anna, aX, +, -, 'any quoted string', []).

Basic Prolog syntax

- **Functors** (function symbols), **predicate symbols**: "atoms" in Prolog, lower case letter first (a, anna, aX, +, -, 'any quoted string', []).
- **Variables:** upper case letter first (X, Xs).

Basic Prolog syntax

- **Functors** (function symbols), **predicate symbols**: "atoms" in Prolog, lower case letter first (a, anna, aX, +, -, 'any quoted string', []).
- **Variables:** upper case letter first (X, Xs).
- **Special anonymous variable: _.**

Basic Prolog syntax

- **Functors** (function symbols), **predicate symbols**: "atoms" in Prolog, lower case letter first (a, anna, aX, +, -, 'any quoted string', []).
- **Variables:** upper case letter first (X, Xs).
- **Special anonymous variable: _**.
- **Comments:** % Single line comment /* Block comment */

We use P/n to denote a predicate P of a specific arity n, e.g. append/3.

Basic Prolog syntax

- **Facts**

```
mother(anna, bob). % anna is mother of bob.
likes(_, icecream). % Everybody likes icecream.
```

- **A rule** *Head* :- *Body*

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

Basic Prolog syntax

- **Facts**

```
mother(anna, bob). % anna is mother of bob.
likes(_, icecream). % Everybody likes icecream.
```

- **A rule** *Head* :- *Body*

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

X is a grandparent to Z if there is some Y such that X is a parent of Y and Y is a parent of Z.

Basic Prolog syntax

- **Facts**

```
mother(anna, bob). % anna is mother of bob.
likes(_, icecream). % Everybody likes icecream.
```

- **A rule** *Head* :- *Body*

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

X is a grandparent to Z if there is some Y such that X is a parent of Y and Y is a parent of Z.

- Both facts and rules end with a *period*.

Querying Prolog

A query ("goal"): (written in *prolog* buffer)

```
| ?- grandparent(X, julia).
no
```

Querying Prolog

A query ("goal"): (written in *prolog* buffer)

```
| ?- grandparent(X, julia).
no
```

Queries may be conjuctions of atomic queries separated by commas:

```
| ?- grandparent(X, jonatan),
     grandparent(X, skorpan).
X = astrid
```

Querying Prolog

A query ("goal"): (written in *prolog* buffer)

```
| ?- grandparent(X, julia).
no
```

Queries may be conjuctions of atomic queries separated by commas:

```
| ?- grandparent(X, jonatan),
     grandparent(X, skorpan).
X = astrid
```

Prolog answers with
- an answer substitution represented as equations,
- "yes" (empty answer substitution), or
- "no".

Querying Prolog

A query ("goal"): (written in *prolog* buffer)

```
| ?- grandparent(X, julia).
no
```

Queries may be conjuctions of atomic queries separated by commas:

```
| ?- grandparent(X, jonatan),
     grandparent(X, skorpan).
X = astrid
```

Prolog answers with

- an answer substitution represented as equations,
- "yes" (empty answer substitution), or
- "no".

To find all the answers, repeatedly press ; which forces Prolog to back-track and give another solution if it can find one. When debugging *always* use ; to check that your program does not give erroneous answers.

A simple example

Simple database example on whiteboard.

Data structures in Prolog

- Terms – the only data type in logic programming.
  Built out of constants, variables and function symbols.
- Some built-in predicates require that their arguments are restricted to certain classes of terms.

Data structures in Prolog. Lists

Single-linked lists just as in Lisp.

Formal object	Alternative notation		
.(a,t)	[a|t]		(cons of a and t)
.(a,[ ])	[a|[ ]]	[a]	
.(a, .(b,[ ]))	[a|[b|[ ]]]	[a,b]	
.(a, .(b, .(c,[ ])))	[a|[b|[c|[ ]]]]	[a,b,c]	

Basic list processing programs on whiteboard.

Data structures in Prolog. Ex.– binary trees.

```
is_tree(leaf(_)).
is_tree(tree(L, R)) :- is_tree(L), is_tree(R).
```

Data structures in Prolog. Ex. – binary trees.

```
is_tree(leaf(_)).
is_tree(tree(L, R)) :- is_tree(L), is_tree(R).

% search(Tree,X) - Tree has a leaf containing X

search(leaf(X), X).
search(tree(L, _R), X) :-
    search(L, X).
search(tree(_L, R), X) :-
    search(R, X).
```

Data structures in Prolog. Ex. – binary trees.

```
  is_tree(leaf(_)).
  is_tree(tree(L, R)) :- is_tree(L), is_tree(R).

% search(Tree,X) - Tree has a leaf containing X

  search(leaf(X), X).
  search(tree(L, _R), X) :-
      search(L, X).
  search(tree(_L, R), X) :-
      search(R, X).

?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), b).
 yes
?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), d).
 no
```

Data structures in Prolog. Ex. – binary trees.

```
  is_tree(leaf(_)).
  is_tree(tree(L, R)) :- is_tree(L), is_tree(R).

% search(Tree,X) - Tree has a leaf containing X

  search(leaf(X), X).
  search(tree(L, _R), X) :-
       search(L, X).
  search(tree(_L, R), X) :-
       search(R, X).

?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), b).
 yes
?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), d).
 no

?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), X).
 X = a ? ;
 X = b ? ;
 X = c ? ;
 no
```

Data structures in Prolog. Ex.– binary trees.

```
is_tree(leaf(_)).
is_tree(tree(L, R)) :- is_tree(L), is_tree(R).
```

% search(Tree,X) – Tree has a leaf containing X

```
search(leaf(X), X).
search(tree(L, _R), X) :-
    search(L, X).
search(tree(_L, R), X) :-
    search(R, X).
```

```
?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), b).
 yes
?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), d).
 no
```

```
?- search(tree(tree(leaf(a), leaf(b)), leaf(c)), X).
 X = a ? ;
 X = b ? ;
 X = c ? ;
 no
```

```
?- search( T, d ).
 T = leaf(d) ? ;
 T = tree(leaf(d),_A) ? ;
 T = tree(tree(leaf(d),_A),_B) ? ;
 T = tree(tree(tree(leaf(d),_A),_B),_C) ?
 yes
```

Checking whether a Boolean formula is true

- A Boolean formula F is a formula built out of the connectives $\wedge$ (conjunction), $\vee$ (disjunction) and $\neg$ (negation), where the basic propositional atoms are *true* and *false*.
- **Checking whether a Boolean formula is true**
  Input: A Boolean formula F.
  Question: Is F true?

  Program on white board.

Extra-logical and built-in predicates

- **Arithmetic:** Prolog has built in support for evaluating arithmetical expressions.
  is/2 computes the value of its 2nd argument
  (must be an arithmetical expression.).

Extra-logical and built-in predicates

- **Arithmetic:** Prolog has built in support for evaluating arithmetical expressions.
  is/2 computes the value of its 2nd argument
  (must be an arithmetical expression.). For example:

```
| ?-  X is 1+2*3.14.
X = 7.28 ? ;
no
```

Extra-logical and built-in predicates

- **Arithmetic:** Prolog has built in support for evaluating arithmetical expressions.
  is/2 computes the value of its 2nd argument
  (must be an arithmetical expression.). For example:

```
| ?-  X is 1+2*3.14.
X = 7.28 ? ;
no
| ?- X = 1+2*3.14.
X = 1+2*3.14 ? ;
no
```

Compare it with

Extra-logical and built-in predicates

- **Arithmetic:** Prolog has built in support for evaluating arithmetical expressions.
  is/2 computes the value of its 2nd argument
  (must be an arithmetical expression.). For example:

```
| ?- X is 1+2*3.14.
X = 7.28 ? ;
no
```

Compare it with

```
| ?- X = 1+2*3.14.
X = 1+2*3.14 ? ;
no
```

- </2 and >/2 evaluate both arguments as arithmetic expressions.

- **All-solution predicates:**
  Use findall/3 to find all solutions to a query. E.g.:

  findall(X, append(_, X, [a,b,c]), Xs)

  finds all suffixes of the list [a,b,c].

- **All-solution predicates:**
  Use findall/3 to find all solutions to a query. E.g.:

  findall(X, append(_, X, [a,b,c]), Xs)

  finds all suffixes of the list [a,b,c].
- **Sort:** Use sort/2 to sort a list and remove duplicates.
  Much more efficient than writing your own sorting algorithms.

Debugging logic programs

Logic programs introduces an extra dimension to debugging due to non-determinism. Need to ensure that your program only gives correct answers and that it gives all the correct answers.

Debugging logic programs

Logic programs introduces an extra dimension to debugging due to non-determinism. Need to ensure that your program only gives correct answers and that it gives all the correct answers.

- **trace:** Use trace/0 to trace all predicates on all ports (CALL, EXIT, REDO, FAIL). Not practical for large programs.
- **spy:** Use spy/1 to set a spy-point on a specific predicate.

Debugging logic programs

Logic programs introduces an extra dimension to debugging due to non-determinism. Need to ensure that your program only gives correct answers and that it gives all the correct answers.

- **trace:** Use trace/0 to trace all predicates on all ports (CALL, EXIT, REDO, FAIL). Not practical for large programs.
- **spy:** Use spy/1 to set a spy-point on a specific predicate.

For testing your own code you can also use write/1 to output relevant information at certain stages in the program. NB: do not forget to flush the buffer with a newline with nl/0 – otherwise it might not be displayed immediately.

Hints and tips

- **There are no “return values”!**
  Using the result of a predicate is done by reusing variables:

```
do_something(Input, Output) :-
    find_thing(Input, Thing),
    process(Thing, Output).
```

Not

```
do_something(Input) :-
    Thing = find_things(Input),
    process(Thing)
```

Hints and tips

- **There are no "return values"!**
  Using the result of a predicate is done by reusing variables:

```
do_something(Input, Output) :-
    find_thing(Input, Thing),
    process(Thing, Output).
```

  Not

```
do_something(Input) :-
    Thing = find_things(Input),
    process(Thing)
```

- **Prolog data are (uninterpreted) terms.**
  5+6 is not 11. (Canonically writen, it is +(5,6)).
  Arithmetic expressions are evaluated to numbers only in a special context of a few built-in predicates (e.g. is/2)

Hints and tips

- **Recursion, not for-loops.**

Hints and tips

- **Recursion, not for-loops.**
- **No space between a symbol and its arguments.**
  f(x) is a valid term, f (x) is not.

Hints and tips

- **Recursion, not for-loops.**
- **No space between a symbol and its arguments.**
  f(x) is a valid term, f (x) is not.
- **Facts, not functions.**
  Each fact and rule should be literally correct, taken as a statement by itself.

Hints and tips

- **Recursion, not for-loops.**
- **No space between a symbol and its arguments.**
  f(x) is a valid term, f (x) is not.
- **Facts, not functions.**
  Each fact and rule should be literally correct, taken as a statement by itself.
- **Prolog is not typed.**
  The query $member([1, 2, 3], 1)$ will just fail and not give any warning.

Hints and tips

- **Recursion, not for-loops.**
- **No space between a symbol and its arguments.**
  f(x) is a valid term, f (x) is not.
- **Facts, not functions.**
  Each fact and rule should be literally correct, taken as a statement by itself.
- **Prolog is not typed.**
  The query $member([1, 2, 3], 1)$ will just fail and not give any warning.
- **Upper case means variables, lower case for everything else.**
  $member(X, [x, y, z])$ and $member(x, [x, y, z])$ give completely different results.

Hints and tips

- **The Prolog buffer is for queries, not programming.** Unlike languages like Python, Perl, Ruby etc, you should not create new predicates in the Prolog buffer, only query the existing set. Use the evaluate region/file/buffer commands in emacs.

Hints and tips

- **The Prolog buffer is for queries, not programming.**
  Unlike languages like Python, Perl, Ruby etc, you should not create new predicates in the Prolog buffer, only query the existing set. Use the evaluate region/file/buffer commands in emacs.
- **Be careful when you reorder your program.**
  It is very easy to forget to change commas and periods in your subqueries.

Hints and tips

- **The Prolog buffer is for queries, not programming.** Unlike languages like Python, Perl, Ruby etc, you should not create new predicates in the Prolog buffer, only query the existing set. Use the evaluate region/file/buffer commands in emacs.
- **Be careful when you reorder your program.** It is very easy to forget to change commas and periods in your subqueries.
- **Sicstus allows infinite structures to be created.**

```
| ?- X = [1|X].

X = [1,1,1,1,1,1,1,1,1,1|...] ?
```

This can be occasionally be useful, but it can also be a source of confusing bugs, since they are meaningless in conventional logic.