Computational Logic (320441) Fall 2015

Michael Kohlhase
Jacobs University Bremen
For Course Purposes Only

October 29, 2015

Contents

Assignment 0 (Getting started with ProLog) Given Sep. 16., Due Sep. 23.

We will now discuss how to use a `ProLog` interpreter to get to know the language. The SWI `ProLog` interpreter can be downloaded from `http://www.swi-prolog.org/`. To start the `ProLog` interpreter with `pl` or `prolog` or `swipl` from the shell. The SWI manual is available at `http://www.swi-prolog.org/pldoc/`

We will introduce working with the interpreter using unary natural numbers as examples: we first add the fact[1] to the knowledge base

```
unat(zero).
```

which asserts that the predicate `unat`[2] is `true` on the term `zero`. Generally, we can add a fact to the knowledge base either by writing it into a file (e.g. `example.pl`) and then "consulting it" by writing one of the following commands into the interpreter:

```
[example]
consult('example.pl').
```

or by directly typing

```
assert(unat(zero)).
```

into the `ProLog` interpreter. Next tell `ProLog` about the following rule

```
assert(unat(suc(X)) :- unat(X)).
```

which gives the `ProLog` runtime an initial (infinite) knowledge base, which can be queried by

```
?- unat(suc(suc(zero))).
Yes
```

Running `ProLog` in an `emacs` window is incredibly nicer than at the command line, because you can see the whole history of what you have done. Its better for debugging too. If you've never used `emacs` before, it still might be nicer, since its pretty easy to get used to the little bit of `emacs` that you need. (Just type "`emacs \&`" at the `UNIX` command line to run it; if you are on a remote terminal like `putty`, you can use "`emacs -nw`".).

If you don't already have a file in your home directory called "`.emacs`" (note the dot at the front), create one and put the following lines in it. Otherwise add the following to your existing `.emacs` file:

```
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
    (autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
    (setq prolog-program-name "swipl"); or whatever the prolog executable name is
    (add-to-list 'auto-mode-alist '("\\pl$" . prolog-mode))
```

[1]for "unary natural numbers"; we cannot use the predicate `nat` and the constructor functions here, since their meaning is predefined in `ProLog`

[2]for "unary natural numbers".

The file `prolog.el`, which provides `prolog-mode` should already be installed on your machine, otherwise download it at `http://turing.ubishops.ca/home/bruda/emacs-prolog/`

Now, once you're in `emacs`, you will need to figure out what your "meta" key is. Usually its the alt key. (Type "control" key together with "h" to get help on using `emacs`). So you'll need a "`meta-X`" command, then type "`run-prolog`". In other words, type the meta key, type "`x`", then there will be a little window at the bottom of your `emacs` window with "`M-x`", where you type `run-prolog`[3]. This will start up the `SWI ProLog` interpreter, ... et voilà!

The best thing is you can have two windows "within" your `emacs` window, one where you're editing your program and one where you're running `ProLog`. This makes debugging easier.

The exercises in this assignment will confront you with the main (conceptual) problems of programming `ProLog`, like relational programming, recursion, and a term language. You do not have to solve them (no points), but they could help you with the programming tasks in the logic assignment.

[3]Type "control" key together with "h" then press "m" to get an exhaustive mode help.

Problem 0.1 Program a predicate for addition in unary representation. The number 3 in unary representation is the ProLog term `s(s(s(o)))`, i.e. application of the arbitrary function `s` to an arbitrary value `o` iterated three times. Note that ProLog does not allow you to program (binary) functions,so you must come up with a three-place predicate. 0pt

You should use `add(?X,?Y,?Z)` to mean $X+Y=Z$ and program the recursive equations $X+0=X$ (base case) and $X+f(Y)=f(X+Y)$.

If you have mastered addition, try your luck on multiplication and exponentiation.

Solution:

```
uadd(X,o,X).
uadd(X,s(Y),s(Z)) :- add(X,Y,Z).
```

The problems for multiplication and exponentiation are quite similar

```
umult(_,o,o).
umult(X,s(Y),Z) :- umult(X,Y,W), uadd(X,W,Z).
uexpt(_,o,s(o)).
uexpt(X,s(Y),Z) :- uexpt(X,Y,W), umult(X,W,Z).
```

Problem 0.2 Write predicates for `mymember`, `myappend` and `myreverse` of lists in default ProLog, i. e. without using the built-in `member/append/reverse` predicates. 0pt

---

Solution:

```
mymember(X,[X]).
mymember(X,[_|R]):-mymember(X,R).
myappend([],L,L).
myappend([X|R],L,[X|S]):-myappend(R,L,S).
myreverse([],[]).
myreverse([X|R],L):-myreverse(R,S),myappend(S,[X],L).
```

Problem 0.3 (Trace of a ProLog Program)

With the `trace` command in ProLog you can look at the execution of a given program step by step. Try this command on the program below and explain the trace output. 0pt

```
a(X,Y):-b(X,Y),c(Y).
b(X,Y):-d(X,Y),e(Y).
b(X,_):-f(X).
c(4).
d(1,3).
d(2,4).
e(3).
f(4).
```

Problem 0.4 0pt

1. Write a program that computes the n^{th} Fibonacci Number (0, 1, 1, 2, 3, 5, 8, 13,... add the last two to get the next), using the addition predicate above.

2. Revise the program, so that it uses ProLog's internal arithmetic. Test your program. If you ask for another solution (by typing a `;`), does it loop? Why? How can you get around this?

Solution:

1.
```
ufib(zero,zero).
ufib(suc(zero),suc(zero)).
ufib(suc(suc(X)),Y):-ufib(suc(X),Z),ufib(X,W),uadd(Z,W,Y).
```

2. The naive solution

```
fib(0,0).
fib(1,1).
fib(X,Y):- D is X - 1, E is X - 2,fib(D,Z),fib(E,W), Y is Z + W.
```

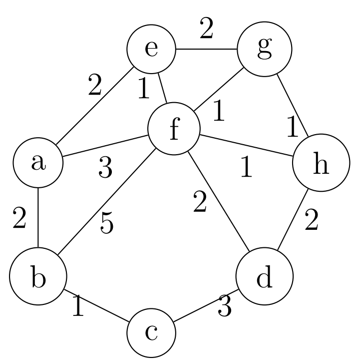
indeed loops for the second solution: For instance the query `ufib(2,Y).` will end up in the base cases after one call to the recursive clause. If we reject that base case, then `ProLog` goes back into the knowledge base and into the recursive clause again, proceeding to negative numbers and looping. If we change the last line to

```
fib(X,Y):- X > 1, D is X - 1, E is X - 2,fib(D,Z),fib(E,W), Y is Z + W.
```

the second recursive call will fail and we obtain the solution we are interested in.

Problem 0.5 (Path Cost)

Represent the graph below as facts in a `ProLog` knowledge base and write a predicate `has_path(I,G,C)` that is true if there exists a path from node I to node G that is of cost less or equal to C. Assume that every node has a path to itself with a cost of 0. 0pt



Here is sample run:

```
?-has_path(a,g,5).
Yes
```

Solution:

```
edge(a,b,2).
edge(a,e,2).
edge(a,f,3).
edge(b,a,2).
edge(b,c,3).
edge(b,f,5).
edge(c,b,3).
```

```
edge(c,d,3).
edge(d,c,3).
edge(d,f,2).
edge(d,h,2).
edge(e,a,2).
edge(e,f,1).
edge(e,g,2).
edge(f,a,3).
edge(f,b,5).
edge(f,d,2).
edge(f,e,1).
edge(f,g,1).
edge(f,h,1).
edge(g,e,2).
edge(g,f,1).
edge(g,h,1).
edge(h,f,1).
edge(h,d,2).
edge(h,g,1).

has_path(G,G,C) :- C >= 0.
has_path(I,G,C) :- C >= 0, edge(I,X,Y), Z is C-Y, has_path(X,G,Z).
```

Test Cases:

```
has_path(a,c,2). %-> No
has_path(g,b,5). %-> No
has_path(c,e,-3). %-> No
has_path(a,d,5). %-> Yes
has_path(d,a,5). %-> Yes
has_path(c,c,5). %-> Yes
has_path(h,e,2). %-> Yes
```

Problem 0.6 (Permutations in ProLog)

0pt

1. Construct a predicate `eliminate(X,Y,Z)` that eliminates the element X from the list Y (the result being list Z). If the element is not in the list, the predicate should yield no solution ($false$).

2. Use the predicate above to define another predicate, `permute(X,Y)`, that computes all the permutations of list X. `permute(X,Y)` is true if Y is a permutation of X.

Solution:

```
eliminate(_,[],[]).
eliminate(X,[X|A],B) :- eliminate(X,A,B).
eliminate(X,[Y|A],[Y|B]) :- eliminate(X,A,B), X\==Y.

permute([],[]).
permute([H|T],Y) :- length([H|T],L), length(Y,L),
               eliminate(H,[H|T],X1), eliminate(H,Y,Y1),
                              permute(X1, Y1).
```

Problem 0.7 (Binary search)

Implement a binary search predicate in ProLog bin_search(X,L,P). Where L is a sorted list of integers, X is an integer value we want to find in the list and P is the position of this value in the list. Do not concern yourself with the case when X appears multiple times. 0pt

Note: Check http://en.wikipedia.org/wiki/Binary_search if in doubt about the algorithm.

Solution:

```
bin_search_helper(X,S,Y,Y,Y) :- nth1(Y,S,X).
bin_search_helper(X,S,F,L,R) :- F < L, M is (F + L) // 2,
    nth1(M,S,E), X =< E, bin_search_helper(X,S,F,M,R).
bin_search_helper(X,S,F,L,R) :- F < L, M is (F + L) // 2,
    N is M +1, bin_search_helper(X,S,N,L,R).

bin_search(X,L,P) :- length(L,M), bin_search_helper(X,L,1,M,P).

?- bin_search(1,[1,2,3,14,15,16,17],1).
?- bin_search(17,[1,2,3,14,15,16,17],7).
?- bin_search(14,[1,2,3,14,15,16,17],4).
?- bin_search(15,[1,2,3,14,15,16,17],5).

?- bin_search(0,[1,2,3,14,15,16,17],P).
```

Assignment 1 (ProLog for Logics) Given Sep. 16., Due Sep. 23.

We will now consider a formulation of propositional logic, which we will call PL_{NQ} (**P**redicate **L**ogic with **N**o **Q**uantifiers). We have already seen this in class. The idea is to use very elaborate names for propositional logic: `ProLog` terms, which encode atomic formulae.

Use `ProLog` for Talking/Programming about Logics

- **Idea**: We will use PLNQ (prop. logic where prop. variables are ADT terms)
- represent the ADT as facts of the form

```
constant(mia).
pred(love,2).
pred(run,1).
fun(father,1)
```

  this licenses `ProLog` terms like `run(mia).` and `love(mia,father(mia)).`

- represent propositional connectives as `ProLog` operators, which we declare with the following declarations.

```
:- op(900,yfx,<>). % equivalence
:- op(900,yfx,>). % implication
:- op(850,yfx,\/). % disjunction
:- op(800,yfx,\&). % conjunction
:- op(750,fx,~). % negation
```

  The first argument of `op` is the operator precedence, the second the fixity. This licenses `ProLog` terms like `X > Y.` and `~(X).`

- Use the `ProLog` built-in predicate `=..` to deconstruct terms: a literal `f(a,b)=..Z` binds `Z` to the list `[f,a,b]`, i.e. the first element of the list is the function/predicate symbol, followed by the arguments.

Problem 1.1 Write a simple syntax checker that checks arities in function application and complex formulae by writing a predicate `wff/1`[4]. 10pt

Problem 1.2 Remember that we call a set $\mathcal{H}$ of atomic formulae in PLNQ a Herbrand model; it induces a valuation ν for PLNQ by $\nu(A) := \mathsf{T}$, iff $A \in \mathcal{H}$. 10pt

Write a couple of example Herbrand models (sets of atomic formulae), using a binary `model/2`[5] relation, given by `ProLog` facts like the following

```
model(3,[love(peter,mary),hate(mary,peter)]).
```

Check well-formedness of the models, using the predicate `wff/1` from Problem 1.1.

Problem 1.3 Write a simple evaluator for closed formulae 10pt

```
evaluate(love(peter,mary) \& hate(mary,peter),3)
```

should succeed. `evaluate` should fail if the input is not valid or ill-formed.

Hint: use the built-in predicates `\+` (not provable).

Problem 1.4 Write a translator predicate that translates away all logical connectives except `&` and `~`. 10pt

Problem 1.5 Extend the previous definition of a `wff` by an operator checking for syntactic equality to get PLNQ$^=$. 10pt

---

Hint: Define a new (infix) predicate `===`, and extend the predicates defined above by new clauses.

---

[4] the `/1` is the notation for a unary predicate.

[5] the first parameter just denotes the number of the model.

Assignment 2 (A logical Analysis of $\mathcal{ND}^1_=$) Given Sep 30., Due Oct 7.

The objective of this assignment is to perform a full logical analysis of first-order natural deduction with equality.

- **Definition 2.1 (First-Order Logic with Equality)** We extend PL^1 with a new logical symbol for equality $= \in \Sigma_2^p$ and fix its semantics to $\mathcal{I}(=) := \{\langle x, x\rangle \mid x \in \mathcal{D}_\iota\}$. We call the extended logic **first-order logic with equality** ($\mathrm{PL}^1_=$)

- We now extend natural deduction as well.

- **Definition 2.2** For the calculus of natural deduction with equality $\mathcal{ND}^1_=$ we add the following two equality rules to $\mathcal{ND}^1$ to deal with equality:

$$\frac{}{\mathbf{A} = \mathbf{A}} {=}I \qquad \frac{\mathbf{A} = \mathbf{B} \quad \mathbf{C}\,[\mathbf{A}]_p}{[\mathbf{B}/p]\mathbf{C}} {=}E$$

  where $\mathbf{C}\,[\mathbf{A}]_p$ if the formula $\mathbf{C}$ has a subterm $\mathbf{A}$ at position p and $[\mathbf{B}/p]\mathbf{C}$ is the result of replacing that subterm with $\mathbf{B}$.

Again, we have two rules that follow the introduction/elimination pattern of natural deduction calculi.

The biggest single problem in this assignment is Problem 2.2, you can work as a team of two on this. The other problems are warm-up problems or side-issues, they are to be solved individually.

Problem 2.1 (Soundness of $\mathcal{ND}^1_=$)
Establish formally that first-order natural deduction calculus $\mathcal{ND}^1_=$ is sound. 20pt

Problem 2.2 (Model Existence for $\mathcal{ND}^1_=$)
Show a model existence theorem for $\mathrm{PL}^1_=$ along the lines of the one for PL^1 we covered in class. In particular you will need to 50pt

1. come up with a notion of abstract consistency class for $\mathrm{PL}^1_=$. This will involve coming up with one or more conditions $\nabla^i_=$ that deal with the (semantic) properties of equality.

Hint: If you look ahead at ?prob.plnqeqtab-complete? and what you will have to prove there, this may give you ideas for the $\nabla^i_=$.

---

2. show that the abstract consistency class can be compactified.
3. establish Hintikka sets and their properties and show an extension result.
4. build a model for $\mathrm{PL}^1_=$ from Hintikka sets.

Hint: This is place, where things are different than in class. Think about how to construct an Herbrand model instead of a valuation; to get the interpretation of equality right, you will have to make a quotient construction.

Problem 2.3 (Completeness of $\mathcal{ND}^1_=$)

With the model existence theorem from Problem 2.2, establish the completeness of $\mathcal{ND}^1_=$. 20pt

If you cannot prove completeness for the calculus, extend it with suitable inference rules.

Assignment 3 (Getting your hands dirty with MMT) Given Oct. 5., Due Oct. 11.

Problem 3.1 (Completing $\mathcal{ND}^0$ in MMT)

We have developed an MMT encoding for propositional logic PL^0 and the propositional calculus $\mathcal{ND}^0$ of natural deduction. 20pt

1. Extend them with the remaining connectives and inference rules from the slides.
2. Test your encoding by theorems whose proofs use all inference rules in the encoding.

Problem 3.2 (Testing the MMT encoding of $\mathcal{ND}^1$)

We have developed an MMT encoding for the first-order logic PL^1 and the first order calculus $\mathcal{ND}^1$ of natural deduction. Test your encoding by theorems whose proofs use all inference rules. 10pt

Problem 3.3 ($\mathrm{PL}^1_=$ and $\mathcal{ND}^1_=$ in MMT)

Give MMT encodings for $\mathrm{PL}^1_=$ and $\mathcal{ND}^1_=$. Test them by theorems whose proofs use all inference rules. 20pt

Assignment 4 (Tableaux and Unification) Given Oct. 13, Due Oct. 21

Problem 4.1 Revise the evaluator from Assignment 1 to a tableau theorem prover/model generation procedure for closed PLNQ formulae that only contain the connectives for conjunction ($\wedge$) and negation ($\neg$). 15pt

Problem 4.2 Extend the model generator from Problem 4.1 to one that works on arbitrary PLNQ closed formulae. 15pt

---

Hint: You can use the translation predicate (function in `Scala`) from Assignment 1.

Problem 4.3 (Prolog only)
For `Prolog`, extend the representation of PLNQ to first-order logic, by adding variables and quantifiers. 15pt

---

Hint: Extend the signature by facts of the form **var(x).**. Yes, we will use constants for variables (at the moment).

Problem 4.4 (First-Order Unification)
Write your own `Prolog` predicate function for first-order unification using the unification algorithm $\mathcal{U}$ from the lecture. 30pt

Problem 4.5 (First-Order Tableaux)
Extend the tableau procedure from the previous exercises to first-order logic. Implement standard tableaux and free variable tableaux. 25pt

Assignment 4 (λ-Calculus) Given Oct. 20, Due Oct. 29

In this assignment, we will implement the λ-calculus in `ProLog` or `Scala`. We will build on our work from the assignment on first-order tableaux, and we will extend the formulae by types and λ-expressions.

Problem 4.1 (Types)
Represent types as `ProLog` terms or `Scala` classes. 10pt

- For `ProLog`, use constants `e` and `t` for the base types, and the infix operator `->` (use the appropriate `op` declaration). Write a predicate `wft/1` that succeeds if its argument is a well-formed type.
- For `Scala` define classes `E`, `T` (for base types) and `Arrow` for composite ones.

Problem 4.2 (λ-terms)
Represent function application and lambda abstraction in `ProLog` or `Scala`. 5pt

- For `ProLog`, the types of constants will be given by a functional predicate `tconst/2`, which maps every constant to a type, e.g. we represent the fact that the `love` is a binary predicate by `tconst(love, e -> e -> t)`. Function application is represented by the infix operator `@`, so that we would represent "*Peter loves Mary*" as `love @ peter @ mary`. λ-abstractions will be represented as triples of the form `lambda(x,e,B)`, where the first argument is the bound variable – we use a `ProLog` constant for it, the second is its type, and the third the body (another formula).

Hint: Note that application is left-associative in contrast to the type constructor `->` above, which is right-associative, use the right operator declaration, so that you can save brackets.

- For `Scala`, define case classes `Cons(name, type)` and `Var(name)` for constants and variables where `name` is a `string` and `type` and λ-type from the previous problem. Moreover, declare `Apply(f,x)` and `Lambda(x, e, B)` where the arguments are the same as for the `ProLog` description.

Problem 4.3 (Type-Checking)
10pt

- For `ProLog`, define a type checking predicate `tc/2`, where `tc(F,T)` checks the whether the type of the formula `F` is `T`.

Hint: As the λ-binder introduces type assumptions for bound variables, you will need an internal predicate `tcaux/3`, which takes a list of type assumptions for the bound variables as an argument to make the recursion go through.

Note that the `tc` can also compute the type of course.

- For `Scala` define a function `tc(f)` that returns the type of the formula `f`. Raise an exception if the input is ill-typed.

Problem 4.4 (Free in)
Find out whether a variable is free in a formula. 10pt

- For `ProLog`, we have represented variables in the λ-calculus by `ProLog` variables, so we will have to determine whether some variable is free in a formula. Write a predicate `freein/2` that does that.

- For `Scala`, write a function `freein(f,x)` that checks if `x` is free in `f`.

Problem 4.5 (Free/Bound Variables)
15pt

- For `ProLog`, we have represented variables in the λ-calculus by `ProLog` variables, so we will need to have functions (functional predicates) that give us the free and bound variables of a λ-term.

Hint: For the predicate `free` interpret any atom (`ProLog` constant) that is is not a constant as a variable.

- For `Scala`, define two functions `free(a)` and `bound(a)` that return the free and, respectively, bound variables from `a`.

Hint: Totally disregard types in these functions.

Problem 4.6 (Alphabetic Variants)
Check whether two λ-terms can be obtained from each other by renaming bound variables. 10pt
Write a `ProLog` predicate or a `Scala` function `alphavariants/2` that checks whether two λ-terms can be obtained from each other by renaming bound variables

---

Hint: The best way to do this is to recurse down the two formulae in parallel, keeping a table of variable equivalences.

Problem 4.7 (Substitution)
25pt

We will need a notion of substitution in our representation of the λ-calculus.

- For `ProLog`, write a predicate `subst/4`, such that the query `subst(a,x,b,R)` binds `R` to the result of substituting `a` for every free occurrence of `x` in `b`.

Hint: Remember that $[\mathbf{B}/X](\lambda X.\mathbf{A}) = \mathbf{A}$ and that for computing $[\mathbf{B}/X](\lambda Y.\mathbf{A})$, where $Y \in \mathrm{free}(\mathbf{B})$ we need to rename the variable Y in $\lambda Y.\mathbf{A}$ to avoid variable capture.

- For `Scala`, write a function `subst(a,x,b)` that returns result of substituting `a` for every free occurrence of `x` in `b`.

Problem 4.8 (β-Normalization)
Implement β-normalization in your λ-calculus. 15pt

- For `ProLog`, write a predicate `betanf/2`, so that the query `betanf(X,Y)` binds `Y` to the β-normal form of `X`.

- For `Scala`, write a function `betanf(x)` that returns the β-normal form of `x`.