Assignment 9 (last one)

Due: Friday, 14.07.2017, 15:59 via Git

For help, contact alp-staff@lists.iai.uni-bonn.de (staff only) or alp-course@lists.iai.uni-bonn.de (staff and participants).

Start working on the exercises early enough so that you can contact your tutor in time if you have problems. Don't expect your tutor to be available at midnight or during weekends!

Submit your implemented predicates as a file named "assignment09/solutions.pl" in the Git repository of your group. Add to each task not just the code that you implemented but also the console output of a session in which you test that **each** solution works for the provided input data and some queries that represent sensible test cases. If no input data is provided in the text of the task, create some sensible input data. If input data is represented as facts, include them into the "solutions.pl" file and add suitable comments.

Task 1. Analyse terms (7 Points)

Implement a predicate `term_type(+Term,-Type)` that unifies `Type` to the most specific type of `Term`. For instance, `?- term_type(1, T).` should unify T to 'integer', not to 'number' or 'atomic'. The following test must succeed:

```
:-begin_tests(assignment_9_task_1).

test(term_type) :-term_type(_, T0),     assertion(T0=var),
                  term_type(a, T1),     assertion(T1=atom),
                  term_type(1, T2),     assertion(T2=integer),
                  term_type(1.0, T3),   assertion(T3=float) ,
                  term_type(f(1), T4),  assertion(T4=compound),
                  X=f(X),
                  term_type(X, T5),     assertion(T5=cyclic).

:-end_tests(assignment_9_task_1).
```

To run the test, add its code to your Prolog file, consult it and call?- run_tests. The SWI-Prolog testing frameworkis documented in detail at http://www.swi-prolog.org/pldoc/package/plunit.html. See Section 2.2.5 One body with multiple tests using assertions for an explanation of assertion/1.

Submit your code (1 point per solved case) and the output of the successful test run (1 point).

Sample Solution:

```
%% term_type(+Term, -Type) is det
%
% succeeds if Term is of type Type

term_type(X, T) :- var(X), !, T=var.
term_type(X, T) :- cyclic_term(X), !, T=cyclic.
term_type(X, T) :- compound(X), !, T=compound.
term_type(X, T) :- integer(X), !, T=integer.
term_type(X, T) :- float(X), !, T=float.
term_type(X, T) :- atom(X), !, T=atom.
```

Task 2. Interactive IO predicates (7 Points)

Implement a predicate `interactive_type_analysis/0` that performs the following steps:

1. Ask the user for a term to be analysed.
2. Read from the console (standard input) a term terminated by a period.
3. Analyse the term with the predicate from task 1.
4. Write the corresponding term type nicely as an answer to the console (standard output). If you failed to solve task 1 just write 'This is the answer.'
5. Ask whether the user wants to continue or abort and tell him what to enter for which option.
6. Read the input and abort or start again from 1.

Tip: There are many helpful predefined predicates. See

- http://www.swi-prolog.org/pldoc/man?section=IO for file handling
- http://www.swi-prolog.org/pldoc/man?section=termrw for reading and writing terms
- http://www.swi-prolog.org/pldoc/man?section=format for formatted output.

Submit your code (1 point per solved case) and the output of a successful test session in which you tested at least two terms before aborting (1 point).

Sample Solution:

```
interactive_type_analysis :-
  repeat,
    write('Enter the term that you want to be analyzed: '),
    read(Term),
    term_type(Term, Type),    % from task 1
    format('This term is of type ~a~n', [Type]),
```

```
    write('Do you want to check other terms (y/n)? '),
    read(Answer),
  Answer=='n',
  !.
```

Task 3. File IO predicates (9 Points)

Implement a predicate `general_type_analysis/0` that behaves like `interactive_type_analysis/0` but additionally

7. asks the user at the start whether the session should also be logged in a file,
8. asks for a file name (if the user answered 'yes') and writes all subsequent interaction (questions and answers) also to the file, up to and including the final 'abort' choice of the user,
9. closes the file properly, no matter whether the interaction was terminated normally by entering 'abort' or abnormally by an exception or interrupt (like <Ctrl>-C).

Tip: Think first which type of IO (ISO, Edinburgh, SWI) is best suited for your task.

Submit your code (1 point per solved case).

Sample Solution:

We use ISO style since we need two explicit outputs: Every output needs to go to the user *and* to the logfile. Thus global redirection of the single output of interactive_type_analysis to a file would not be a sufficient solution (It would only write to the logfile but not anymore to the user). Thus, neither the Edingburh style nor the SWI-style is well-suited for this task.

Note that the sample solution for task 2 (interactive_type_analysis) uses read/1 and write/1, implicitly assuming that the current input and output is the Prolog console.

If we do not want to rely on this assumption, we can make the source and destination explicit by using read/2 and write/2. Each has the destination as additional first argument. The standard input is called 'user_input' and the standard output is called 'user_output'. The implementation below uses this approach. It also shows how to treat incorrect answers to the yes/no question (see last two lines):

```
general_type_analysis :-
  write('Do you want to log the session (y/n)? '),
  read(user_input,Answer),
 (  Answer=='n'
  -> interactive_type_analysis                    % See task 2
  ;  (  Answer=='y'
     -> ( write(user_output, 'Please enter the name of the log file: '),
          read(user_input,Filename),
          setup_call_cleanup(
              open(Filename, write, Log),
              logged_interactive_type_analysis(Log),  % See below
              close(Log)
          )
        )
```

```
      ; ( writeln(user_output, 'Please answer y or n.'),
          general_type_analysis                          % Try again
        )
      )
   ).

logged_interactive_type_analysis(File_Output) :-
  % Define message strings / templates:
  AskForInputTerm = 'Enter the term that you want to be analyzed: ',
  AskForContinuation = 'Do you want to continue (y/n)? ',
  ReportTermType = 'This term is of type ~a~n',
  % Backtracking-driven i/o loop:
  repeat,
    write(user_output, AskForInputTerm),          % Ask user
    write(File_Output, AskForInputTerm),          % Log asked question
    read(user_input, Term),                       % Read input term
    format(File_Output, '~w.~n', [Term]),         % Log read input

    term_type(Term, Type),                        % Determine the term's
type
    format(user_output, ReportTermType, [Type]), % Report result to user
    format(File_Output, ReportTermType, [Type]), % Log reported result

    write(user_output, AskForContinuation),       % Ask user
    write(File_Output, AskForContinuation),       % Log asked question
    read(user_input, Answer),                     % Read answer
    format(File_Output, '~w.~n', [Answer]),       % Log answer
  Answer=='n',
  !.
```

Task 4. Metaprogramming (2 points)

Task 2 of assignment sheet 6 required to find a way to compute and print out (on the console) all solutions of a given predicate p(X,Y). The sample solution is online meanwhile. However, it is unsatisfactory, since it applies only to the predicate p/2.

Your task is to improve that solution by writing a generic predicate

print_all_solutions(Goal)

that takes an arbitrary goal as a parameter, calls it exhaustively (that means, until all results have been computed via backtracking) and prints each result to the standard output.

```
print_all_solutions(Goal) :- % failure-based loop
      call(Goal),                    % 1 point
        writeln(Goal),               % 1 point
      fail.
print_all_solutions(_) :- % succeed in the end
      writeln('No (more) solutions').
```

Task 5. Metaprogramming (11 points)

Often, one changes the implementation of a predicate, believing that the new version is more understandable, better structured, more efficient, or better in some other way. In such cases, it would be good, however, to make sure that the changes did not affect the behavior of the predicate. A good way to do this is not to overwrite the predicate immediately but create a renamed version of it (let's assume, we add '_new' to the predicate name), implement our improvement idea and then compare the original and new version, to make sure that

a) the new version finds each result that the old one found
b) the new version does not find any result that the old one didn't find.

Your task is to automate this comparison by implementing a predicate

Your task is to improve that solution by writing a generic predicate

print_differences(PredName, Arity)

that takes the name and arity of the *original* predicate as input, calls each of the two versions (the original and the new one) and prints out only the problem cases, that is,

a) results found by the original version but not by the new one
b) results found by the new version but not by the original one.

```
print_differences(PredName, Arity) :-

   % Construct call to original predicate and new (renamed) one:
   atom_concat(PredName, '_new', NewName),                    % 1
   functor(NewCall,NewName,Arity),                            % 1
   NewCall =.. [NewName | Arguments ],
   OrigCall =.. [PredName | Arguments ],                      % 1

   % Print results missed by new version:
   call(OrigCall),                                          % 1
   not(call(NewCall)),                                      % 1
   format('Result missed by new version: ~w~n', [OrigCall]),  % 1
   % Print incorrect results of new version:
   call(NewCall),                                             % 1
   not(call(OrigCall)),                                       % 1
   format('Undesired result of new version: ~w~n', [NewCall]),% 1

   fail.                                                      % 1
print_differences(_, _) :-                                    % 1
   format('No more results').
```

Task 6. Metaprogramming (2 points)

Is print_differences/2 (previous task) sufficient to guarantee *identical* behavior of the old and new predicate? Explain why (or why not). (1 Point per argument pro or contra)

No. It does not detect, whether there are multiple identical results and whether the results are delivered in the same order. To do that one would need to use findall/3 (see prolog built-in metapredicates) to get a list of all results and traverse the lists to find mismatches.
Eg.

```
New: P(1),p(2),p(3),p(3)
Old: p(3),p(2),p(1)
```