Homework 7: λProlog Programming and Type Checking

Due: 6 April, 1994 (although we'd appreciate it if you turn in problem 1 by Friday).
This is a group homework.
This homework combines ideas from both our studies of logic programming, functional programming (especially the "concrete data" we studied in SML) and type checking. For type checking, you might want to read Chapter 7 of Watt's book "Programming Languages Concepts and Paradigms".
For all λProlog programs, you must run your code with the elp interpreter. You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and your testing.
Unless we specifically request, your code does not have to run "backwards". (But you may want to see which ones can do that.) In the examples you will see "no more solutions" or "yes" printed; your program does not have to have exactly this output.
You can use cut (`!`) and **`not`** in your code.
You might want to split up the work on the first 3 problems, and then get together to criticize and evaluate your answers for them; be sure that each group member understands all of them. For the other problems, I would suggest each person designing a solution to one problem, meeting to discuss it, working it out together, and then doing the same for the following problems in sequence.

1. (46 points) For each of the terms: *logical variable, goal or query, term, ground term, closed world assumption, substitution, instance, common instance, soundness, completeness, unifier, unify, most general unifier (mgu), occurs check, unification, cut, resolution, backtracking* (i) define (both formally and informally), (ii) give one example or illustration or counterexample.

   For each of the terms: *closed world assumption, substitution, common instance, soundness, completeness, unifier, most general unifier (mgu), occurs check, cut, backtracking* (iii) state the importance of the term or why it is a good or bad thing in the context of logic programming and λProlog.

2. (10 points; extra credit only, but this might be on the test) Describe and compare the logical and procedural interpretations of a λProlog program.

3. (10 points; extra credit only, but this might be on the test) Compare logic programming to functional and object-oriented programming, and describe what each is best used for.

4. (30 points) As an example of the kind of thing you are to do in this problem, look in the directory

   `~leavens/src/lambda-prolog/mono_type_inference`

   at the files for the modules `mono_test`, `sml_abstract_syntax`, `mono_type_inference`, and `env`. These files encode the type inference algorithm for the subset of standard ML's core that we discussed in class, except that they do not deal with polymorphism. The module `mono_test` is a simple suite of tests, where you can see how the others are used. The module `sml_abstract_syntax` gives the abstract syntax of SML expressions, declarations, and types; note that all of these are λProlog functions. (For more about what abstract syntax is, see section 2.4 of Watt's *Programming Language Syntax and Semantics*.) The module `mono_type_inference` has the type inference rules. And the `env` provides a standard environment data type (from Watt's book).

   Your problem is to design and implement in λProlog, similar type checking rules for "mini Cecil". In the directory

```
~leavens/src/lambda-prolog/mini_cecil_problem
```

you will find files to get you started on this problem. The module `env` is the same as in the example. All of the other modules are different.

The module `signature` describes algebraic signatures, which can be thought of as holding information about types and messages. The type of a message can be extracted by using the `resType` predicate defined in this module. The concept of a signature is used because I like to think of the classes and methods of a program as modeled mathematically by an algebra. An algebra has a signature as described in section 6.1 of Watt's *Programming Language Syntax and Semantics*.

The signature of the built-in types of mini Cecil is provided in the module `visibles`. Note that the message `theVoid` is used to create a void object; this object would be returned as the result of a procedure or method that does not want to return anything useful. (So mini Cecil's type `Void` corresponds directly to C's.)

We have provided the abstract syntax for "mini Cecil" in the module `mini_cecil_abstract_syntax`. Several points should be made about this syntax. A program looks like list of class declarations, a list of method declarations, a list of procedure declarations, and a main procedure. The class declarations have a list of fields; these are like the instance variables in Smalltalk. In expressions, message sends, which invoke a method of a class, are distinguished from procedure calls. New objects are created by the use of a `newExpr` expression, which takes as its arguments a list of values for the fields of an object, in the order in which fields were declared in the class declaration. The fields of an object may be accessed by the `getExpr`. The fields of an object may be set by a `setFieldStmt`. The main program is divided into two parts for reasons that will become apparent during the discussion of the denotational semantics of mini Cecil. Essentially you can just think of these parts as two series of declarations and statements, where the second series executes in the scope of the first set of declarations.

An outline of the type checking (not inference) rules for mini Cecil is in the module `mini_cecil_type_checking`. You are to fill in wherever you see `edit_here`. You will have to decide on what the type checking rules should be, and then code them in λProlog.

Hand in your completed module `mini_cecil_type_checking`, and some tests of type checking. (Warning, there may be bugs in the code that you don't have to edit; especially in the context of whatever code you may put in the program. All such bugs are your ultimately your responsibility, but you may consult with us to check if they are bugs or if you have misunderstood something.)

5. (50 points) The version of "mini Cecil" in the previous problem doesn't have many object-oriented features. To start adding some, consider overloading and multiple implementations of a type. The way we'd like to add this is to do the following.

   - Add type declarations to the abstract syntax,
   - Let class declarations state the type that the class implements; class names are no longer type names. Where will your code keep track of the what type a class implements? What is the type of a `newExpr` with a given class name?
   - Allow overloading of method declarations based on the classes of arguments objects. To do this, change `methodFormalDecl` in the abstract syntax to add a "specializer" that gives a class name. The idea is objects know what class they have, and a method with specializers `C1` and `C1` for its first and second arguments, will be invoked in preference to one with classes `C2` and `C2` for its arguments if at run-time the actual argument objects have classes `C1` and `C1`. The class name `Any` may be used to indicate that an argument is not specialized; hence all classes are more specialized than `Any`. When is one method "more specialized than" another? Can you state that in terms of an ordering (binary relation) on sequences of classes? Let us say a "method can handle a call `m(x,y)`" if the

classes of the actual argument objects, `x` and `y`, are more specialized than the classes of some method defined for the message `m` with two arguments. Note that you don't have to consider all possible classes in the above definition. What classes have to be considered? When does one have "enough" methods to prevent errors from calling a method in a program that otherwise typechecks with objects of some classes, and not having a method that can handle the call?

For this part hand in answers to the above questions, perhaps with some other English describing other design decisions you felt were important; and also a working type checker coded in λProlog for this extension to mini Cecil, and tests. (You need only hand in modules that you added or changed.)

6. (60 points) Add subtyping to "mini Cecil". Hand in an English describing your design; and also a working type checker coded in λProlog for this extension to mini Cecil, and tests. (You need only hand in modules that you added or changed.) Hint: don't put in a rule that says that subtyping is reflexive; that will send λProlog into an infinite loop. You'll have to treat reflexivity some other way.

7. (10 points, extra credit; programming, semantics) In the setup for the first mini Cecil problem above, add code to check for duplicate declarations.

8. (15 points; extra credit; programming, semantics) In the setup for the second mini Cecil problem above, add code to check that all types names used in a program (except of course in one place) have been previously declared.