lab session 3: Functional Programming: uProlog

The objective of this lab session is to implement an interpreter for a (very) small subset of the Prolog Programming language. We will call this language `uProlog` (micro-Prolog).

Before you start, you should download a zip-file (named `lab3FPuProlog.zip`) from brightSpace and unzip it. The zip archive contains (skeleton) haskell files and a few example prolog programs.

The uProlog interpreter consists of two major parts. The first part, the so-called *front-end* parses the input (a program), while the second part (called the *back-end*) performs the actual execution.

The first two exercises focus on the front-end. The remaining exercises focus on the back-end. Exercise 1 and 3 are both worth 1 grade point. The remaining exercises are worth 2 grade points each, totalling 10 grade points. No manual assessment by the TAs will be performed.

Introduction: The `uProlog` language

`uProlog` (like standard Prolog) has its roots in first-order logic. The language is a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running queries over these relations.
For example, the following code is a valid `uProlog` program (moreover, it is also a valid Prolog program). It is included in the zip archive (filename `family.upl`):

```
% some facts
child(john,sue).  % fact saying that sue is a child of john.
child(john,sam).  % fact saying that sam is a child of john.
child(jane,sue).  % fact saying that sue is a child of jane.
male(john).       % fact saying that john is a male.
male(robert).     % fact saying that robert is male
% some rules
parent(Y,X) :- child(X,Y).            % if Y is a child of X, then X is a parent of Y
father(Y,X) :- child(X,Y), male(X).   % if Y is a child of X and X is a male then X is the father of Y.
% some queries
?- child(john,sue).
?- child(john,john).
?- child(john,pete).
?- male(john).
?- male(robert).
?- male(jane).
?- male(eve).
?- male(X).
?- child(john,X).
?- parent(sue,X).
?- father(X,Y).
?- child(X,X).
```

Once the uProlog interpreter is completed, you should be able to run the program. The output will be:

```
child(john,sue): yes
child(john,john): no
child(john,pete): no
male(john): yes
male(robert): yes
male(jane): no
male(eve): no
male(X): X <- [john,robert]
child(john,X): X <- [sam,sue]
parent(sue,X): X <- [jane,john]
father(X,Y): (X,Y) <- [(sam,john),(sue,john)]
child(X,X): X <- []
```

Note that the output format produced by the `uProlog` interpreter differs from the output format that is produced by a standard Prolog interpreter.

In `uProlog` identifiers start with a letter, followed by zero or more characters or digits (e.g. `pi314` is a valid identifier). The name of relations and constant objects are identifiers that start with a lower case letter. If the first letter of an identifier is an upper case letter, then the identifier is a variable (i.e. a formal parameter of a relation).
A statement in `uProlog` can be a *fact*, a *rule*, or a *query*. Each statement is terminated by a full stop (`'.'`).

- Fact: A *fact* defines a relation with contant arguments. The name of the relation is followed by one or more constant arguments which are enclosed by parentheses. Arguments are separated by commas.
- Rule: A *rule* consists of two parts. The first part is similar to a fact, however it is a relation with non-constant arguments (i.e. the relation has formal parameters). The second part consists of other facts or rules which are separated by commas which must all be true for the rule itself to be true, so the comma between two conditions can be considered as a logical-AND operator. The two parts of a rule are separated by "`:-`". You may interpret this operator as "if" in English.
- Query: A *query* starts with "`?-`". It is a statement starting with the name of a predicate (a fact, or a rule) followed by its arguments, some of which may be variables.

In a `uProlog` program, a fact indicates a true statement. An absence of a fact indicates a statement that is not true. This explains why, in the example program, the query `?- child(john,sue)` yields the output `yes`, since this was given as a fact in the first line of the program. The query `?- child(john,john)` yields the answer `no` because `child(john,john)` is not specified as a fact, nor can it be deduced using any of the rules. The same holds for queries like `?- male(eve)` since `eve` does not occur anywhere.

A more interesting query is `?- male(X)` which asks to output all `X` for which `male(X)` is true. This yields the answer `X <-[john,robert]` which should be read as `male(X)` is true for all `X`'s that are taken from the list `[john,robert]`. If a query has multiple non-constant arguments, then lists of tuples are generated. An example of this is the query `?- father(X,Y)` which yields the answer `(X,Y) <- [(sue,john),(sam,john)]`. If a query has no satisfying answer, then the empty list is returned. This is the case for the query `?- child(X,X)`.

The syntax of `uProlog` is given by the following grammar (which is in LL(1) format):

```
Prolog          -> Statement Prolog
Prolog          -> <empty string>

Statement       -> '?-' Relation '.'
Statement       -> Relation Statement'
Statement'      -> ':-' RelationList '.'
Statement'      -> '.'

RelationList    -> Relation RelationList'
RelationList'   -> ',' Relation RelationList'
RelationList'   -> <empty string>

Relation        -> <relation identifier> Args

Args            -> '(' ArgList ')'
ArgList         -> Argument ArgList'
ArgList'        -> ',' Argument ArgList'
ArgList'        -> <empty string>

Argument        -> <variable> | <constant>
```

Exercise 1: Lexer for uProlog

We start with writing a lexer for uProlog. In the zip archive, you can find an incomplete module Lexer.hs containing the following code:

```
module Lexer(LexToken(..),lexer) where
import Data.Char
import Error

data LexToken = DotTok   | CommaTok | FollowsTok
              | QueryTok | LparTok  | RparTok
              | IdentTok String
              | VarTok String
  deriving Eq

instance Show LexToken where
  show DotTok             = "."
  show CommaTok           = ","
  show FollowsTok         = ":-"
  show QueryTok           = "?-"
  show LparTok            = "("
  show RparTok            = ")"
  show (IdentTok name)    = "<id:" ++ name ++ ">"
  show (VarTok name)      = "<var:" ++ name ++ ">"

lexer :: String -> [(LexToken,Int)]    -- The Int is the line number in the source code
-- Please implement this function yourself
```

The data type LexToken consists of 8 tokens. The relation between tokens and their lexical representation is clear from the function show. The tokens IdentTok and VarTok represent identifiers and have a String field containing the name of the identifier. For VarTok this identifier starts with an uppercase letter, while for any other identifier we return an IdentTok (which start with a lower case letter). Identifiers in uProlog are any string that start with a letter followed by zero or more letters and digits.

You must implement the function lexer which gets as its input a uProlog program as a String. The lexer should skip white space (newlines, tabs, spaces) and also comments. In uProlog a comment starts with the character '%' and spans the rest of the line. The output of the lexer should be a list of (LexToken,Int) pairs, where the Int represents the line number where a token is found. These line number are used later for proper error reporting in the parser.

If an invalid character is encountered, the lexer should print an error message and abort the program. The module Error.hs (also available in the zip archive) supplies a couple of error reporting functions. One of these functions is the function lexError that takes two arguments. The first is an Int which is a line number in the uProlog program, and the second is a Character. For example, the call lexError 42 '#' will produce the error mesage "Lexical error in line 42: unexpected character '#'.".

A demo module testLexer.hs is available to test your lexer. Once you have completed the lexer module, you can compile your program using the command: ghc testLexer. The output should be an executable testLexer. In the zip archive you can find the file sorates.upl which contains the following content:

```
mortal(X) :- man(X).
man(socrates).
?- mortal(socrates).
```

You can test your lexer by typing ./testLexer socrates.upl on the command line of your shell. It should produce the following output:

```
[(<id:mortal>,1),((,1),(<var:X>,1),(),1),(:-,1),(<id:man>,1),((,1),(<var:X>,1),(),1),(.,1),
(<id:man>,2),((,2),(<id:socrates>,2),(),2),(.,2),(?-,3),(<id:mortal>,3),((,3),(<id:socrates>,3),
(),3),(.,3)]
```

In the zip archive you can also find the file lexerror.upl which contains (deliberately) a lexical error:

```
mortal(X) :- man(X).
man(socrates).
!- mortal(socrates).
```

Executing `./testLexer lexerror.upl` should produce the following output:

```
Lexical error in line 3: unexpected character '!'.
```

You should submit only the completed module `Lexer.hs` to Themis.

Exercise 2: Making a Parser for `uProlog`

The second step of the front-end is to make a parser for the grammar that was given in the introduction. The parser must check the syntax of the input, and should perform proper error reporting. Moreover, for syntactically correct input, the parser should return a value of the type `Program` which is a representation of the input. The type `Program` is defined, together with some other types, in the module `Types.hs` (available in the zip archive). This module exports the following types:

```
data Argument = Const String
              | Arg String

data FuncApplication = FuncApp String [Argument]

data Statement = Fact FuncApplication
               | Rule FuncApplication [FuncApplication]
               | Query FuncApplication

data Program = Program [(Statement,Int)]
```

An `Argument` can be a constant identifier or an argument. A constant is an identifier that starts with a lower case letter, while an argument starts with an upper case letter. For example, `Arg "X"` represents the argument `X` of some relation, while `Const "obj"` represent the name of the constant identifier `obj`, which can be a relation name (e.g. `child`) or the name of a constant object (e.g. `john`).

The type `FuncApplication` represents the application of a function (relation) to its arguments. The data type has two fields: a `String` representing the relation name, and a list of arguments. For example, `child(X,sue)` is represented by `FuncApp "child" [Arg "X",Const "sue"]`.

A `Statement` is either a `Fact`, a `Rule`, or a `Query`. The fact `child(john,sue)` is represented by the expression `Fact (FuncApp "child" [Const "john",Const "sue"])`. Similarly, the Query `?- child(X,sue)` is represented by the expression `Query (FuncApp "child" [Arg "X",Const "sue"])`. A `Rule` has two fields. The first field is the conclusion (the head) of the rule, while the second is a list of premises (i.e. a list of function appliations). For example, a `uProlog` rule like `father(Y,X) :- male(X), child(X,Y)` is repsented by the following `Rule` expression:
`Rule (FuncApp "father" [Arg "Y",arg "X"]) [FuncApp "male" [Arg "X"],FuncApp "child" [Arg "X",Arg "Y"]]`

A `Program` is a list of pairs `(Statement,Int)`. The first element of this pair is clearly a statement, and the second is the (starting) line number of the statement in the input. These line numbers are used for error reporting in later stages of the interpreter.

In the zip archive, an incomplete module `Parser.hs` is available. Its body consists of:

```
parseProgram :: String -> Program
parseProgram input = Program (parseProlog [] (lexer input))

parseProlog :: [(Statement,Int)] -> [(LexToken,Int)] -> [(Statement,Int)]
```

You must complete this module yourself by implementing the function `parseProlog`. The first argument of this function is what has been accepted thus far, the second argument are the tokens (and corresponding line numbers) of the remaining input, and the result is a representation of a correctly parsed program. The parser must be written in the same fashion as the parsers that you made in lab 2 (and the one discussed in the lecture).

Note that the parser should perform proper syntax error reporting. On the detection of an error the program should abort. There are suitable error reporting functions available in the module `Error.hs`. In Themis all input/output tests are visible, so you can infer from these which errors (the exact text!) must be reported.

In the zip archive there is a module `testParser.hs` available which you can use to test your parser. Once you have completed the parser module, you can compile a test program using the command: `ghc testParser`. The output should be an executable `testParser`. If you run this program with the file `socrates.upl` as its input, the output will look like:

```
1:mortal(X) :- man(X)
2:man(socrates)
3:?- mortal(socrates)
```

You must submit only the completed module `Parser.hs` to Themis.

A proper front end of an interpreter (or a compiler) should also perform *semantic checking*. For example, it should test whether the number of arguments of a function call matches the number of arguments in the definition of the function. Writing a semantic analayser is quite a bit of work, and has been done for you (after all, this lab should not become a compiler construction lab). Once your parser has been accepted by Themis, you can use the files `uProlog.hs` and `Analysis.hs` from the zip archive. The file `uProlog.hs` contains the main program which in the end will be the full `uProlog` interpreter. The file `Analysis.hs` is a module that exports only one function:

```
analyse :: Program -> Program
```

The main program calls the parser. If the parser returns a successful representation of the input program (i.e. no syntax errors have been encountered) then the resulting `Program` is passed to the function `analyse`, which performs semantic checking. If this checking fails, then the function prints a proper error message and the program aborts. In case of a successful semantic analysis (i.e. no errors were found), then the function simply returns the input as its output (i.e. it behaves as the identity function on `Program`s).

Introduction: Logical Inference

The remaining exercises of this lab session focus on the implementation of the back-end. But, before we start building the back-end of the `uProlog` interpreter it is important to understand some theoretical aspects about logical reasoning using an automatic inference technique called *resolution*. You may have encountered this topic before in some other (logic) course, but it won't hurt to recap the topic.

Most general Unifiers (MGU)

We start by introducing the notion of *unification*. The unification process in the context of First Order Logic (FOL) is more general than explained here, but this stripped explanation is sufficient to make this lab exercise.

We consider (as an example) two predicates (i.e. logical expressions with arguments) $\alpha =$ `child(X,Y)` and $\beta =$ `child(john,Y)`.
A *subsitution* is a list of pairs. Each pair consists of a variable (starts with an upper case letter) and a literal (variable name that starts with an upper case letters or an object name that starts with a lower case letter). Three examples of substitutions are $\theta_0 = [(\texttt{X}, \texttt{john}), (\texttt{Y}, \texttt{sue})]$, $\theta_1 = [(\texttt{Y}, \texttt{sue})]$, and $\theta_2 = [\texttt{(Y,B)}]$. The function `subst` takes two arguments: a substitution and a predicate. The result is a new predicate that is obtained by replacing each variable in the original predicate by the corresponding literal from the substitution. Hence, applying `subst` to a predicate is a purely syntactic operation. It is defined as follows:

$$\begin{aligned} \texttt{subst []}\ \alpha &= \alpha \\ \texttt{subst ((X,E):}\theta\texttt{)}\ \alpha &= \texttt{subst}\ \theta\ \alpha_E^X \end{aligned}$$

Here, the notation α_E^X means "replace each occurrence of `X` in α by `E`". As a result:

$$\begin{aligned} \texttt{subst}(\theta_0, \alpha) &= \texttt{child(john,sue)} \\ \texttt{subst}(\theta_1, \alpha) &= \texttt{child(X,sue)} \\ \texttt{subst}(\theta_2, \alpha) &= \texttt{child(X,B)} \\ \texttt{subst}(\theta_0, \beta) &= \texttt{child(john,sue)} \\ \texttt{subst}(\theta_1, \beta) &= \texttt{child(john,sue)} \\ \texttt{subst}(\theta_2, \beta) &= \texttt{child(john,B)} \end{aligned}$$

A *unifier* is a substitution that *unifies* two predicates (i.e. makes them equal). From the above examples, it is clear that θ_0 is a unifier for the predicates α and β because $\texttt{subst}(\theta_0, \alpha) = \texttt{subst}(\theta_0, \beta)$. Here 'equality' means syntactically the same expression.

Given two predicates, there might exist several unifiers. For example, `f(A,B,c,D,E)` and `f(B,a,C,c,F)` are unified with the unifiers $\theta_0 = [(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})]$ and $\theta_1 = [(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{a}), (\texttt{F}, \texttt{a})]$. In fact, there exist more unifiers for this example. However, there is only one single unifier that we call the *most general unifier (MGU)*, which is in this case θ_0.

A *most general unifier* is a unifier that consists of a minimal number of subtitutions. Hence, it transforms the two predicates in the most general unified expressions.

$$\begin{aligned} \texttt{f(a,a,c,c,F)} &= \texttt{subst}([(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})],\ \texttt{f(A,B,c,D,E)}) \\ &= \texttt{subst}([(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})],\ \texttt{f(B,a,C,c,F)}) \end{aligned}$$

The unifier θ_1 would unify the predicates to predicate `f(a,a,c,c,a)` which is clearly less general, because an explicit value for the last argument `F` of `f` was substituted.

It is well known that the most general unifier of two predicates is unique upto renaming of (some) variables. For example, in the MGU θ_0 we rename the variable E to F, but we could just as well decide to rename F to E.
Note that we could have expressed the unifier θ_0 as:

$$\theta_0' = [(\texttt{A}, \texttt{B}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})]$$

We consider θ_0' to be the same unifier as θ_0, since $\texttt{subst}\ \theta_0\ \phi = \texttt{subst}\ \theta_0'\ \phi$ for any predicate ϕ.

Clausal Form

A *clause* is a disjuntion of predicates. For example, the predicate $f(X, Y) \vee g(X) \vee \neg h(Y)$ is a clause. Because there is only one connective ($\vee$), clauses can be represented as a list of predicates. Actually, the list is a set since the order of the elements in a clause is irrelevant, and there are no duplicates in a clause. For the given example this list would be the list $[f(X, Y), g(X), \neg h(Y)]$.

Because a `uProlog` program is a collection of facts and implications (rules), we can easily transform the program into a set of clauses: facts are transformed into singleton clauses, while rules are transformed using the rule that $\alpha \Rightarrow \beta$ is equivalent with $\neg\alpha \vee \beta$.

For example, in the following `uProlog` program fragment this translation in clausal form is given in a comment for each program line (where `~` denotes negation):

```
child(sue,john).                  % [child(sue,john)]
child(sam,john).                  % [child(sam,john)]
child(sue,jane).                  % [child(sue,jane)]
male(john).                       % [male(john)]
parent(Y,X) :- child(X,Y).        % [~child(X,Y),parent(Y,X)]
father(B,A) :- male(A),child(A,B). % [~male(A),~child(A,B),father(B,A)]
```

Inference by Resolution

The logical inferences that the `uProlog` interpreter will make is based on one single inference rule, called the *resolution rule*. Let α, β, γ, δ be predicates. Also, let θ be a most general unifier for α and β. Then the resolution rule states:

$$\frac{\alpha \vee \gamma \quad \neg\beta \vee \delta}{\texttt{subst}\ \theta\ (\gamma \vee \delta)}$$

The conclusion $\texttt{subst}\ \theta\ (\gamma \vee \delta)$ is called the *resolvent*.

The resolution rule is sound because if we apply the unifier θ to $\alpha \vee \gamma$, then we obtain $\texttt{subst}\ \theta\ \alpha \vee \texttt{subst}\ \theta\ \gamma$, which is logically equivalent to $\neg\texttt{subst}\ \theta\ \alpha \Rightarrow \texttt{subst}\ \theta\ \gamma$. Similary, we can apply the unifier to $\neg\beta \vee \gamma$ to obtain $\texttt{subst}\ \theta\ \beta \Rightarrow \texttt{subst}\ \theta\ \delta$. Because θ is a unifier for α and β, we can replace $\texttt{subst}\ \theta\ \beta$ by $\texttt{subst}\ \theta\ \alpha$ to obtain $\texttt{subst}\ \theta\ \alpha \Rightarrow \texttt{subst}\ \theta\ \delta$. Since $\texttt{subst}\ \theta\ \alpha$ is either true or false, we conclude $\texttt{subst}\ \theta\ \gamma \vee \texttt{subst}\ \theta\ \delta$ which equals $\texttt{subst}\ \theta\ (\gamma \vee \delta)$.

The conclusion is that we can use the resolution rule to make inferences in `uProlog` programs as follows. Consider, the clauses `[male(john)]` and `[~male(A),~child(B,A),father(A,B)]`. We can unify the complementary terms `male(john)` and `male(A)` using the most (and only) general unifier `[(A,john)]`. Hence, using the resolution rule we

infer the resolvent (i.e. a new clause) `[~child(B,john),father(john,B)]`. Next, we apply resolution again on this new clause together with the clause `[child(sue,john)]` (using the mgu `[(B,sue)`) to infer `[father(john,sue)]`.

An imperative style pseudo-code of an algorithm that infers all possible clauses by repeated application of the resolution rule is given below. It is the driving heart of the `uProlog` interpreter.

```
clauses := list of clauses representing the uProlog program
continue := true
while continue do {
  inferences := empty set
  for each pair C0,C1 from clauses do {
    inferences := union(inferences,
                        clauses that can be inferred by applying the resolution rule to C0 and C1)
  }
  if inferences is a subset of clauses
  then continue := false
  else clauses := union(clauses,inferences)
}
```

Exercise 3: Conversion into clauses

To get started, you must extend the file `Types.hs` with a few types:

```
type Substitution = (String,Argument)
type Unifier = [Substitution]
type Clause = [(FuncApplication,Bool)]
type Clauses = [Clause]
```

The type `Substitution` represents a substitution of a variable (first element of the pair) by a literal (a constant object or variable). For example, the substitution `(X,john)` is represented by `("X",Const "john")` (which has the type `Substitution`). A renaming of a variable like `(F,E)` is represented by `("F",Arg "E")`.
A unifier is a `Substitution`s list: the unifier `[(X,john),(F,E)]` is stored as `[("X",Const "john"),("F",Arg "E")]`.

A clause is represented by the data type `Clause`, which is a list of pairs. Of each pair, the first element is a function application (i.e. a predicate), while the second is a boolean value. This boolean value is `True` for a positive predicate and `False` for a negated predicate. For example, the clause `[~male(A),~child(B,A),father(A,B)]` is represented by `[(FuncApp "male" [Arg "A"],False),(FuncApp "child" [Arg "B",Arg "A"],False), (FuncApp "father" [Arg "A",Arg "B"],True)]`. On its turn, a set of clauses is represented by the type `Clauses`, which is implemented as a list.

Make a module `Clause.hs` which exports only the function `programToClauses`. The type of this function is

```
programToClauses :: Program -> Clauses
```

The function accepts as its input a `uProlog` program (parsed and checked by the front-end of the interpreter) and outputs the clausal form of this program. Note that in this conversion queries must be skipped. Only facts and rules must be converted to clausal form.

Change the file `uProlog.hs` such that it also imports `Clause.hs` and change the `process` function in this file into:

```
process :: String -> Clauses
process str = (programToClauses.analyse.parseProgram) str
```

You can compile the program using `ghc uProlog.hs`. The resulting executable will be named `uProlog`. You can see the result of the conversion of `socrates.upl` into clauses using the command `./uProlog socrates.upl`. The output should look like:

```
[[(man(Arg X),False),(mortal(Arg X),True)],[(man(Const socrates),True)]]
```

For this exercise, you should submit only the module `Clause.hs` to Themis. Note that the order in the generated list of clauses must be the same as in the input `uProlog` program in order to pass tests in Themis.

Exercise 4: Unification of Predicates

Make a module `Unify.hs` which exports two functions: `mgu`, and `applyUnifier`. Their types are

```
mgu :: FuncApplication -> FuncApplication -> Maybe Unifier
applyUnifier :: Unifier -> FuncApplication -> FuncApplication
```

The function `mgu` accepts two predicates (in the form of a `FuncApplication)`. If these predicates cannot be unified, then the function returns `Nothing`. Otherwise, it returns `Just theta` where `theta` is the most general unifier of the input predicates.

For example, if we wat to unify the predicate `male(john)` and `male(robert)` then this function should return `Nothing` since `john` and `robert` cannot be unified. If we want to unify `male(john)` and `male(X)` then the function should return `Just [("X",Const "john")]`.

You should submit only the module `Unify.hs` to Themis. Note that most general unifiers are not unique in the sense that renaming variables is arbitrary. For example, `f(X)` and `f(Y)` can be unified by substituting `Y` for `X` or by substituting `X` for `Y`. Themis will accept either unifier. This means that Themis is not comparing directly the output of your function with the output of a reference implementation (as it normally does), but the judge for this exercise is a quite complicated process that simply prints `ACCEPTED` in case your program produces a correct MGU. Otherwise, it's output is an error message that (hopefully) gives a hint what went wrong. All test cases in Themis are small files that consist of three lines: the first line is a comment line with the correct (expected) MGU, and the other two lines contain two function applications that need to be unified. For example, test case 15 in Themis is:

```
%mgu should be: Just [("A",f),("C",f),("E",f),("D",f),("B",f)]
f(A,C,E,E,C).
f(B,D,D,f,B).
```

Your program should produce the unifier that is in the first (comment) line. All test cases are visible, so you can always see the input and the output produced during the execution of your function in Themis.

Exercise 5: Resolution

Make a module `Resolution.hs` which exports only one function:

```
resolveClauses :: Clauses -> Clauses
```

The function gets as its input a list of clauses, and outputs an extension of all the possible clauses that can be inferred using the resolution algorithm (of which the pseudo code is given in the introduction). Note that you have to solve a tricky detail, because predicates may have overlapping variable names. Therefore, you may need to rename some variables (in the literature on resolution and unification, this is called *standardizing apart*) before you can apply the `mgu` function.

As an example, the output of

```
(programToClauses.parseProgram) "mortal(X) :- man(X). man(socrates). mortal(pyhtagoras)."
```

should be the following list with 3 clauses:

```
[[(man(X),False),(mortal(X),True)],[(man(socrates),True)],[(mortal(pyhtagoras),True)]]
```

We can apply resolution on this set of clauses using:

```
(resolveClauses.programToClauses.parseProgram) "mortal(X) :- man(X). man(socrates). mortal(pyhtagoras)."
```

The resulting output should look like (note that the naming of variables may differ for your implementation):

```
[[(man(X),False),(mortal(X),True)],[(man(socrates),True)],[(mortal(pyhtagoras),True)],[(mortal(socrates),True)]]
```

As you can see, the resolution process 'inferred' that Socrates is mortal.

Change the file `uProlog.hs` such that it imports `Resolution.hs` and change the `process` function in this file into:

```
process :: String -> Clauses
process str = (resolveClauses.programToClauses.analyse.parseProgram) str
```

You can compile the program using `ghc uProlog.hs`. The resulting executable will be named `uProlog`. You can see the result of applying the resolution process to `socrates.upl` using the command `./uProlog socrates.upl`.

You should submit only the module `Resolution.hs` to Themis.

Exercise 6: Answering Queries

Make a module `Query.hs` that exports only one function:

```
answerQueries :: Program -> String
```

The input of this function is the representation of a `uProlog` program, and the output should be a string representation of the answers produced by the `uProlog` interpreter as a complete string (including newlines).

For example, if we apply this function to the `family.upl` program that is given in the introduction of this lab (page 1), then the output of `answerQueries` should be:

```
"child(john,sue): yes\nchild(john,john): no\nchild(john,pete): no\nmale(john): yes\nmale(robert): yes\n
male(jane): no\nmale(eve): no\nmale(X): X <- [john,robert]\nchild(john,X): X <- [sam,sue]\n
parent(sue,X): X <- [jane,john]\nfather(X,Y): (X,Y) <- [(sam,john),(sue,john)]\nchild(X,X): X <- []"
```

Note that lists in this output string must be (lexicographically) sorted to pass the tests in Themis. You should submit only the module `Query.hs` to Themis.

To wrap up this last lab session, modify the file `uProlog.hs` such that it contains:

```
import System.Environment
import Parser
import Analysis
import Query

process :: String -> String
process str = (answerQueries.analyse.parseProgram) str

main = do
  args <- getArgs
  let reader = if args == [] then getContents else readFile (head args)
  text <- reader
  putStr (process text ++ "\n")
```

You can compile this file using `ghc uProlog.hs`. This will produce a complete interpreter, called `uProlog`. Congratulations, you implemented a (small) programming language!