Functional Programming
lab session 3a

Lab session 3 has been split in two parts: lab 3a and lab 3b. The objective of lab 3 is to implement an interpreter for a (very) small subset of the Prolog Programming language. We will call this language `uProlog` (micro-Prolog).

The interpreter consists of two major parts. The first part, the so-called *front-end* of the interpreter parses the input (a program), while the second part (called the *back-end*) performs the actual execution. In lab 3a we will focus solely on the front-end. In lab 3b we will implement the back-end.

Lab 3a consists of only two exercises. The first exercise is worth 1 grade point. The second exercise is worth 2 grade points.

Note that the deadline for lab 3a is very strict! The reason is that we will publish a reference solution for lab 3a after this deadline, which can be used as the basis for lab 3b.

1 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):

```
% 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(X,Y) :- 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).
```

If we execute this `uProlog` 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 <- [sue,sam]
parent(sue,X): X <- [john,sue]
father(X,Y): (X,Y) <- [(john,sue),(john,sam)]
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 fullstop.

- 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 statement that is true. 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 `[(john,sue),(john,sam)]`. 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>

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

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

Relation       -> Identifier Args

Args           -> '(' ArgList ')'
```

```
ArgList        -> Argument ArgList'
ArgList'       -> ',' Argument ArgList'
ArgList'       -> <empty>

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

Exercise 1: Making a Lexer for `uProlog`

We start with writing a lexer for `uProlog`. On Nestor, 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 on Nestor) 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`. On Nestor you can find the file `sorates.pl` which contains the following content:

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

You can test your lexer by typing `testLexer socrates.pl` 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)]
```

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`. 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(X,Y) :- male(X), child(X,Y)` is repsented by the following `Rule` expression:
`Rule (FuncApp "father" [Arg "X",arg "Y"]) [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 (lab 3b).

On Nestor, an incomplete module `Parser.hs` is available. You must complete this module yourself. Note that the parser should also perform proper 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 available, so you can infer from these which errors (the exact text!) must be reported.

On Nestor, 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.pl` as its input, the output will look like:

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

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

[Note: 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. You will be supplied a module at the start of lab 3b that percforms all these tasks. You do not have to write that yourself.]