**[Due Date: April 25, 2pm]**

# CS 381, Spring 2019 — Homework 1 (Abstract Syntax)

Please follow carefully ***all*** of the following steps:

1. Prepare a Haskell (or literate Haskell) file (ending in `.hs` or `.lhs`, respectively) that compiles without errors in GHCi. (Put all non-working parts and all non-code answers in comments.)
2. Submit ***only one*** solution per team (each team can have up to 5 members), through the COE TEACH web site. List the names of all team members as a comment in the file.
3. Hand in ***one printed*** copy of your (team) solution before class on April 25, or make otherwise sure that the TAs receive the copy before the deadline. Make sure that all lines are readable on the printout.

Late submissions will ***not*** be accepted. Do ***not*** send solutions by email.

## Exercise 1. Mini Logo

Mini Logo is an extremely simplified version of the Logo language for programming 2D graphics. The idea behind Logo and Mini Logo is to describe simple line graphics through commands to move a pen from one position to another. The pen can either be "up" or "down". Positions are given by pairs of integers. Macros can be defined to reuse groups of commands. The syntax of Mini Logo is as follows (nonterminals are typeset in intalics, and terminals are typeset in typewriter font).

*cmd* ::= `pen` *mode*
| `moveto (`*pos*`,`*pos*`)`
| `def` *name* `(` *pars* `)` *cmd*
| `call` *name* `(` *vals* `)`
| *cmd*`;` *cmd*

*mode* ::= `up` | `down`

*pos* ::= *num* | *name*

*pars* ::= *name*`,` *pars* | *name*

*vals* ::= *num*`,` *vals* | *num*

**Note:** Please remember that unspecified nonterminals, such as *num* and *name*, should be represented by corresponding predefined Haskell types, such as `Int` and `String`.

(a) Define the abstract syntax for Mini Logo as a Haskell data type.

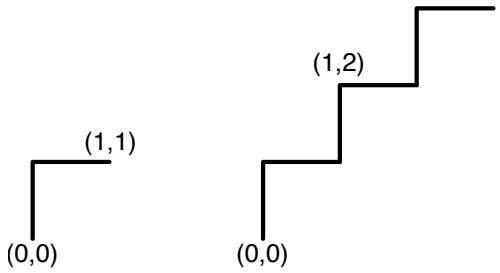
(b) Write a Mini Logo macro `vector` that draws a line from a given position `(x1,y1)` to a given position `(x2,y2)` and represent the macro in abstract syntax, that is, as a Haskell data type value.

> **Note.** What you should actually do is write a Mini Logo program that defines a vector macro. So the answer should have the following form.
>
> ```
> def vector (...) ...
> ```

This is the *textual* representation in *concrete syntax*. Then you should write the same Mini Logo program in abstract syntax, that is, give a Haskell data type value in the following form (assuming `Def` is the constructor name representing the `def` production of the Haskell data type).

```
vector = Def "vector" ... ...
```

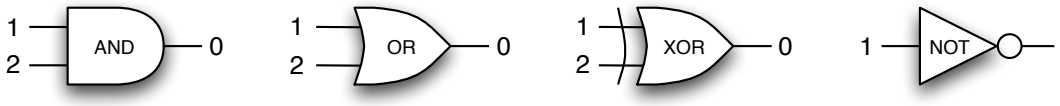
(c) Define a Haskell function `steps :: Int -> Cmd` that constructs a Mini Logo program which draws a stair of $n$ steps. Your solution should ***not*** use the macro `vector`.




Results of the Mini Logo programs produced by `steps 1` and `steps 3`.

## Exercise 2. Digital Circuit Design Language

Digital circuits can be built using the following four basic types of logical gates.




The "**Di**gital **C**ircuits **D**esign **L**anguage" whose syntax is shown below can be used to describe circuits built from these gates.

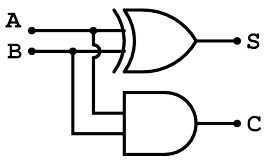
*circuit* ::= *gates links*

*gates* ::= *num*`:`*gateFn* `;` *gates* | $\epsilon$

*gateFn* ::= `and` | `or` | `xor` | `not`

*links* ::= `from` *num*`.`*num* `to` *num*`.`*num*`;` *links* | $\epsilon$

Please note that logical gates as well as their input/output ports are identified by numbers. The inputs of a gate are numbered from top to bottom, starting from 1. The output is always numbered 0. Consider the following circuit, a half adder.




This circuit can be defined by the following DiCiDL program.

```
1:xor;
2:and;
from 1.1 to 2.1;
from 1.2 to 2.2;
```

(a) Define the abstract syntax for the above language as a Haskell data type.

(b) Represent the half adder circuit in abstract syntax, that is, as a Haskell data type value.

(c) Define a Haskell function that implements a pretty printer for the abstract syntax.

## Exercise 3. Designing Abstract Syntax

Consider the following abstract syntax for arithmetic expressions.

```
data Expr = N Int
          | Plus Expr Expr
          | Times Expr Expr
          | Neg Expr
```

Now consider the following alternative abstract syntax. (Note: The different names have been chosen only to allow both definitions to occur within one Haskell module and have otherwise no significance. In particular, the names are irrelevant for exercise part (b).)

```
data Op = Add | Multiply | Negate

data Exp = Num Int
         | Apply Op [Exp]
```

(a) Represent the expression `-(3+4)*7` in the alternative abstract syntax.

(b) What are the advantages or disadvantages of either representation?

(c) Define a function `translate :: Expr -> Exp` that translates expressions given in the first abstract syntax into equivalent expressions in the second abstract syntax.