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 italics, and terminals are typeset in typewriter font).

\[
\begin{align*}
\text{cmd} &::= \text{pen mode} \\
&| \text{moveto } (\text{pos, pos}) \\
&| \text{def name } (\text{pars}) \text{ cmd} \\
&| \text{call name } (\text{vals}) \\
&| \text{cmd;} \text{ cmd}
\end{align*}
\]

\[
\begin{align*}
\text{mode} &::= \text{up } | \text{down}
\end{align*}
\]

\[
\begin{align*}
\text{pos} &::= \text{num } | \text{name}
\end{align*}
\]

\[
\begin{align*}
\text{pars} &::= \text{name, pars } | \text{name}
\end{align*}
\]

\[
\begin{align*}
\text{vals} &::= \text{num, vals } | \text{num}
\end{align*}
\]

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

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

(b) Write a Mini Logo macro \text{vector} that draws a line from a given position \((x_1, y_1)\) to a given position \((x_2, y_2)\) 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.

\[
\begin{align*}
\text{def vector } (...) \ldots
\end{align*}
\]
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 \textbf{Def} is the constructor name representing the \texttt{def} production of the Haskell data type).

\begin{verbatim}
vector = Def "vector" ... ...
\end{verbatim}

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

\begin{center}
\begin{tikzpicture}
  \node at (0,0) (A) {vector};
  \node at (2,0) (B) {vector};
\end{tikzpicture}
\end{center}

Results of the Mini Logo programs produced by \texttt{steps 1} and \texttt{steps 3}.

**Exercise 2. Digital Circuit Design Language**

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

```
AND
1
2
0

OR
1
2
0

XOR
1
2
0
0
0
0

NOT
1
```

The “Digital Circuits Design Language” whose syntax is shown below can be used to describe circuits built from these gates.

```
circuit ::= gates links

gates ::= num:gateFn ; gates \mid \epsilon

gateFn ::= and \mid or \mid xor \mid not

links ::= from num.num to num.num ; links \mid \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.

```
\begin{center}
\begin{tikzpicture}
  \node at (0,0) (A) {AND};
  \node at (2,0) (B) {OR};
  \node at (4,0) (C) {XOR};
  \node at (6,0) (D) {NOT};
\end{tikzpicture}
\end{center}
```

This circuit can be defined by the following DiCiDL program.
Exercise 3. Designing Abstract Syntax

Consider the following abstract syntax for arithmetic expressions.

```haskell
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).)

```haskell
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.