Homework #3

Due: Thur, Feb 6, 10:59pm

How to Submit

- Submit one solution *per team* (each team can have 1–3 members), through **TEACH**. Put the names and ONID username of all team members as a comment at the top of the file.
- Your submission should consist of one file named `HW3.<your-username>.hs`, where `<your-username>` is the ONID username of the team member who submitted the file.
- This file *must compile without errors or warnings* in GHCi. Put all non-working parts of your solution in comments! If your file does not compile, the TAs will not evaluate it.
- If you can't solve a problem, you can still get *full credit* by describing in comments what you tried and where you got stuck.

Description

MiniLogo is toy version of the Logo language for programming simple 2D graphics. A MiniLogo program describes a graphic by a sequence of `move` commands that move a pen from one position to another on a Cartesian plane, drawing lines as it goes. For example, here is a MiniLogo program that draws a 2x2 square with its bottom-left corner at the origin.

```
pen up; move (0,0);
pen down; move (2,0); move (2,2);
          move (0,2); move (0,0);
```

Conceptually, the MiniLogo execution environment consists of two parts:

- a *canvas* rooted at position `(0,0)` and extending infinitely upward and to the right
- a *pen* which is always located at a certain position on the canvas, and which can be in one of two states, either `up` or `down`

The `move` command moves the position of the pen from one position to another. If the pen is `down` when it moves, it draws a straight line connecting the two positions. If the pen is `up`, it just moves to the new position but does not draw a line. The state of the pen can be changed using the `pen` command as illustrated in the example program above.

In addition to basic `pen` and `move` commands, a MiniLogo program can define and invoke *macros*. A macro is a procedure that takes some coordinate values as inputs and performs some commands. Within the body of a macro, commands can refer to input values by name. For example, here is the definition of a macro that draws a 2x2 square starting from an arbitrary origin.

```
define square (x,y) {
  pen up; move (x,y);
  pen down; move (x+2,y); move (x+2,y+2);
            move (x,y+2); move (x,y);
}
```

Now, if I want to draw two squares in two different places, later in my program I can `call` the macro with two different sets of arguments. The following will draw two 2x2 squares, one anchored at position `(3,5)`, the other anchored at `(13,42)`.

```
call square (3,5); call square (13,42);
```

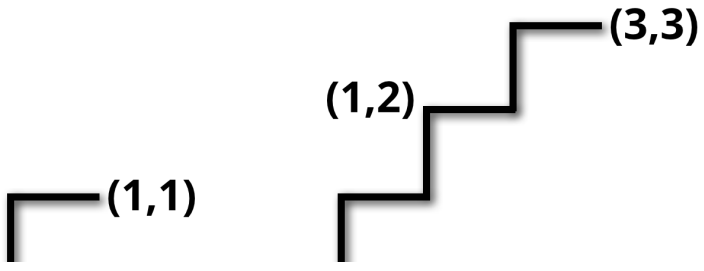
Notice in the definition of the `square` macro, that we can also perform addition on coordinate values.

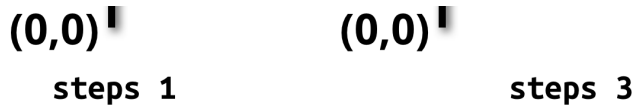
The concrete syntax of the MiniLogo language is defined by the following grammar:

num	::=	(any natural number)	
var	::=	(any variable name)	
macro	::=	(any macro name)	
prog	::=	ε | *cmd* ; *prog*	sequence of commands
mode	::=	`down` | `up`	pen status
expr	::=	*var*	variable reference
	|	*num*	literal number
	|	*expr* + *expr*	addition expression
cmd	::=	`pen` *mode*	change pen mode
	|	`move (` *expr* `,` *expr* `)`	move pen to a new position
	|	`define` *macro* `(` *var** `) {` *prog* `}`	define a macro
	|	`call` *macro* `(` *expr** `)`	invoke a macro

Tasks

1. Define the abstract syntax of MiniLogo as a set of Haskell data types. You should use built-in types for *num*, *var*, and *macro*. (If you want to define a type `Num`, you will have to hide that name from the Prelude).
2. Define a MiniLogo macro `line (x1,y1,x2,y2)` that (starting from anywhere on the canvas) draws a line segment from `(x1,y1)` to `(x2,y2)`.
    - First, write the macro in MiniLogo *concrete syntax* (i.e. the notation defined by the grammar and used in the example programs above). Include this definition in a comment in your submission.
    - Second, encode the macro definition as a Haskell value using the data types defined in Task 1. This corresponds to the *abstract syntax* of MiniLogo. Your Haskell definition should start with something like `line = Define "line" ...`
3. Use the `line` macro you just defined to define a new MiniLogo macro `nix (x,y,w,h)` that draws a big "X" of width `w` and height `h`, starting from position `(x,y)`. Your definition should not contain any `move` commands.
    - First, write the macro in MiniLogo *concrete syntax* and include this definition in a comment in your submission.
    - Second, encode the macro definition as a Haskell value, representing the *abstract syntax* of the definition.
4. Define a Haskell function `steps :: Int -> Prog` that constructs a MiniLogo program that draws a staircase of *n* steps starting from `(0,0)`. Below is a visual illustration of what the generated program should draw for a couple different applications of `steps`. You may assume that $n \geq 0$.





5. Define a Haskell function `macros :: Prog -> [Macro]` that returns a *list of the names* of all of the macros that are defined anywhere in a given MiniLogo program. Don't worry about duplicates—if a macro is defined more than once, the resulting list may include multiple copies of its name.

6. Define a Haskell function `pretty :: Prog -> String` that pretty-prints a MiniLogo program. That is, it transforms the abstract syntax (a Haskell value) into nicely formatted concrete syntax (a string of characters). Your pretty-printed program should look similar to the example programs given above; however, for simplicity you will probably want to print just one command per line.

   In GHCi, you can render a string with newlines by applying the function `putStrLn`. So, to pretty-print a program `p` use: `putStrLn (pretty p)`.

For all of these tasks, you are free to define whatever helper functions you need. You may also use functions from the Prelude and Data.List. You may find the functions `intersperse` or `intercalate` in Data.List useful for inserting commas in your implementation of `pretty`.

Bonus Problems

These problems are not any harder than the other problems in the assignment. They are included mainly to give you a bit more practice writing Haskell functions that manipulate syntax, if you want that. However, as a little external incentive, you will earn a small amount of extra credit if you complete them both.

7. Define a Haskell function `optE :: Expr -> Expr` that partially evaluates expressions by replacing any additions of literals with the result. For example, given the expression `(2+3)+x`, `optE` should return the expression `5+x`.

8. Define a Haskell function `optP :: Prog -> Prog` that optimizes all of the expressions contained in a given program using `optE`.

Back to course home page