[Due Date: April, 30, noon]

CS 381, Spring 2013 — Homework 2 (Semantics)

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 in the file.)
2. Submit ***one*** solution per team (each team can have 2 or 3 members), through the COE TEACH web site. Put the name of all team members as a comment in the file.
3. Hand in a ***printed*** copy of your solution (again, one copy per team) before or after class on April, 30, or drop it off at my office before noon. Make sure that all lines are readable on the printout.

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

Exercise 1. A Stack Language

Consider the stack language *S* defined by the following grammar.

$$\begin{aligned} S &::= C \mid C\texttt{;}S \\ C &::= \texttt{LD}\ \mathit{Int} \mid \texttt{ADD} \mid \texttt{MULT} \mid \texttt{DUP} \end{aligned}$$

An *S*-program essentially consists of a (non-empty) sequence of commands/operations C. The meaning of an *S*-program is to start with an empty stack and to perform its first operation on it, which results in a new stack to which the second operation in *S* (if it exists) is applied, and so on. The stack that results from the application of the last command is the result of the program.

The operation `LD` loads its integer parameter onto the stack. The operation `ADD` removes the two topmost integers from the stack and puts their sum onto the stack. If the stack contains fewer than two elements, `ADD` produces an error. Similarly, the operation `MULT` takes the two topmost integers from the stack and puts their product on top of the stack. It also produces an error if the stack contains fewer than two elements. Finally, the operation `DUP` places a second copy of the stack's topmost element on the stack. (You can find out the error condition for `DUP` yourself.) Here is a definition of the abstract syntax that you should use.

```
type Prog = [Cmd]

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
```

Integer stacks should be represented by the type `[Int]`, that is, lists of integers, that is, your program should contain and use the following definition.

```
type Stack = [Int]
```

Define the semantics for the stack language as a Haskell function `sem` that yields the semantics of a program. Please note that the semantic domain has to be defined as a function domain (since the meaning of a stack program is a transformation of stacks) *and* as an error domain (since operations can fail). Therefore, `sem` has to have the following type where you have to find an appropriate type defintition for `D`.

```
sem :: Prog -> D
```

To define `sem` you probably want to define an auxiliary function `semCmd` for the semantics of individual operations, which has the following type.

```
semCmd :: Cmd -> D
```

Hint. Test your functions with the programs `[LD 3,DUP,ADD,DUP,MULT]` and `[LD 3,ADD]` and the empty stack `[]` as inputs.

Exercise 2. Extending the Stack Language by Macros

Suppose we want to add a simple macro facility to the stack language that allows us to define parameterless macros like `SQR = DUP; MULT`. The definition C would change as follows.

$$C ::= \texttt{LD}\ \textit{Int} \mid \texttt{ADD} \mid \texttt{MULT} \mid \texttt{DUP} \mid \texttt{DEF}\ \textit{String}\ \texttt{(}S\texttt{)} \mid \texttt{CALL}\ \textit{String}$$

The operation `DEF` n `(`C_1`;` ... `;`C_k`)` defines a macro named n that is available in the rest of the program and, when called, causes the execution of C_1`;` ... `;`C_k. `CALL` n calls the macro named n if it has been defined earlier in the program. If n is not a defined macro, `CALL` n yields an error. You can choose how to deal with nested macro definitions (allowing nested macro definitions might lead to a simpler semantics definition).

(a) Extend the abstract syntax to represent macro definitions and calls, that is, give a correspondingly changed data definition for `Cmd`.

(b) Define a new type `State` to represent the state for the new language. The state includes the macro definitions and the stack. Please note that a macro definition can be represented by a pair whose first component is the macro name and the second component is the sequence of commands. Multiple macro definitions can be stored in a list. A type to represent macro definitions could thus be defined as follows.

```
type Macros = [(String,Prog)]
```

(c) Define the semantics for the extended language as a function `sem2`. As in exercise 1, you probably want to define an auxiliary function `semCmd2` for the semantics of individual operations.

Exercise 3. Mini Logo

Consider the simplified version of Mini Logo (without macros), defined by the following abstract syntax.

```
data Cmd = Pen Mode
         | MoveTo Int Int
         | Seq Cmd Cmd

data Mode = Up | Down
```

The semantics of a Mini Logo program is ultimately a set of drawn lines. However, for the definition of the semantics a "drawing state" must be maintained that keeps track of the current position of the pen and the pen's status (`Up` or `Down`). This state should be represented by values of the following type.

```
type State = (Mode,Int,Int)
```

The semantic domain representing a set of drawn lines is represented by the type `Lines`.

```
type Line  = (Int,Int,Int,Int)
type Lines = [Line]
```

Define the semantics of Mini Logo by giving two function definitions. First, define a function `semS` that has the following type.

```
semS :: Cmd -> State -> (State,Lines)
```

This function defines for each `Cmd` how it modifies the current drawing state and what lines it produces.

After that define the semantic function `sem'` of the following type.

```
sem' :: Cmd -> Lines
```

The function `sem'` should call `semS`. The initial state is defined to have the pen up and the current drawing position at $(0, 0)$.

Note. To test your semantics you can use the function `ppLines` defined in the Haskell file provided on the class web site. This function converts a list of lines into an svg file that can be rendered by most browsers into a picture.