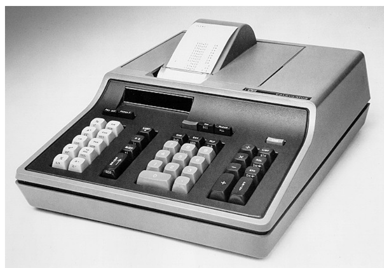
CSCI 490 problem set 10

Due Tuesday, April 5

Revision 1: compiled Wednesday 30th March, 2016 at 05:42

- Files you should submit: `Calc.hs`, containing a module of the same name.

As we have seen in class, Haskell's *type classes* provide *ad-hoc polymorphism*, that is, the ability to decide what to do based on the type of an input. This homework explores one interesting use of type classes in constructing *domain-specific languages*.



Expressions

On day one of your new job as a software engineer, you've been asked to program the brains of the company's new blockbuster product: a calculator. But this isn't just any calculator! Extensive focus group analysis has revealed that what people really want out of their calculator is something that can add and multiply integers. Anything more just clutters the interface.

Your boss has already started by modeling the domain with the following data type of arithmetic expressions:

```
data ExprT where
  Lit :: Integer -> ExprT
  Add :: ExprT -> ExprT -> ExprT
  Mul :: ExprT -> ExprT -> ExprT
  deriving (Show, Eq)
```

This type is capable of representing expressions involving integer constants, addition, and multiplication. For example, the expression $(2+3) \times 4$ would be represented by the value

```
Mul (Add (Lit 2) (Lit 3)) (Lit 4).
```

Your boss has already provided the definition of `ExprT` in `ExprT.hs`, so as usual you just need to add `import ExprT` to the top of your file. However, this is where your boss got stuck.

Exercise 1

Write Version 1 of the calculator: an evaluator for `ExprT`, with the signature

```
eval :: ExprT -> Integer
```

For extra brownie points, first write a fold for `ExprT` and use it to implement `eval`.

For example, `eval (Mul (Add (Lit 2) (Lit 3)) (Lit 4)) == 20`.

Exercise 2

The UI department has internalized the focus group data and is ready to synergize with you. They have developed the front-facing user-interface: a parser that handles the textual representation of the selected language. They have sent you the module `Parser.hs`, which exports `parseExp`, a parser for arithmetic expressions. If you pass the constructors of `ExprT` to it as arguments, it will convert `Strings` representing arithmetic expressions into values of type `ExprT`. For example:

```
*Calc> parseExp Lit Add Mul "(2+3)*4"
Just (Mul (Add (Lit 2) (Lit 3)) (Lit 4))
*Calc> parseExp Lit Add Mul "2+3*4"
Just (Add (Lit 2) (Mul (Lit 3) (Lit 4)))
*Calc> parseExp Lit Add Mul "2+3*"
Nothing
```

Leverage the assets of the UI team to implement the value-added function

```
evalStr :: String -> Maybe Integer
```

which evaluates arithmetic expressions given as a `String`, producing `Nothing` for inputs which are not well-formed expressions, and `Just n` for well-formed inputs that evaluate to *n*.

Exercise 3

Good news! Early customer feedback indicates that people really do love the interface! Unfortunately, there seems to be some disagreement over exactly how the calculator should go about its calculating business. The problem the software department (*i.e.* you) has is that while `ExprT` is nice, it is also rather inflexible, which makes catering to diverse demographics a bit clumsy. You decide to abstract away the properties of `ExprT` with a type class.

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

Create a type class called `Expr` with three methods called `lit`, `add`, and `mul` which parallel the constructors of `ExprT`. Make an instance of `Expr` for the `ExprT` type, in such a way that

```
mul (add (lit 2) (lit 3)) (lit 4) :: ExprT
  == Mul (Add (Lit 2) (Lit 3)) (Lit 4)
```

Think carefully about what types `lit`, `add`, and `mul` should have.

It may be helpful to consider the types of the `ExprT` constructors.

Remark. Take a look at the type of the foregoing example expression:

```
*Calc> :t mul (add (lit 2) (lit 3)) (lit 4)
Expr a => a
```

What does this mean? The expression `mul (add (lit 2) (lit 3)) (lit 4)` has *any type* which is an instance of the `Expr` type class. So writing it by itself is ambiguous: GHC doesn't know what concrete type you want to use, so it doesn't know which implementations of `mul`, `add`, and `lit` to pick.

One way to resolve the ambiguity is by giving an explicit type signature, as in the above example. Another way is by using such an expression as part of some larger expression so that the context in which it is used determines the type. For example, we may write a function `reify` as follows:

```
reify :: ExprT -> ExprT
reify = id
```

To the untrained eye it may look like `reify` does no actual work! But its real purpose is to constrain the type of its argument to `ExprT`. Now we can write things like

```
reify $ mul (add (lit 2) (lit 3)) (lit 4)
```

at the `ghci` prompt.

Exercise 4

The marketing department has gotten wind of just how flexible the calculator project is and has promised custom calculators to some big clients. As you noticed after the initial roll-out, everyone loves the interface, but everyone seems to have their own opinion on what the *semantics* should be. Remember when we wrote `ExprT` and thought that addition and multiplication of integers was pretty cut and dried? Well, it turns out that some big clients want customized calculators with behaviors that they have decided are right for them.

The point of our `Expr` type class is that we can now write down arithmetic expressions *once* and have them interpreted in various ways just by using them at various types.

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

Make instances of `Expr` for each of the following types:

- `Integer` — works like the original calculator
- `Bool` — every literal value less than or equal to 0 is interpreted as `False`, and all positive `Integers` are interpreted as `True`; "addition" is logical or, "multiplication" is logical and
- `MinMax` — "addition" is taken to be the `max` function, while "multiplication" is the `min` function
- `Mod7` — all values should be in the ranage $0 \ldots 6$, and all arithmetic is done modulo 7; for example, $5 + 3 = 1$.

The last two variants work with `Integers` internally, but in order to provide different instances, we wrap those `Integers` in `newtype` wrappers. These are used just like the `data` constructors we've seen before.

```
newtype MinMax  = MinMax Integer deriving (Eq, Show)
newtype Mod7    = Mod7 Integer deriving (Eq, Show)
```

Once done, the following code should demonstrate our family of calculators:

```
testExp :: Expr a => Maybe a
testExp = parseExp lit add mul "(3 * -4) + 5"

testInteger  = testExp :: Maybe Integer
testBool     = testExp :: Maybe Bool
testMM       = testExp :: Maybe MinMax
testSat      = testExp :: Maybe Mod7
```

Try printing out each of those tests in `ghci` to see if things are working. It's great how easy it is for us to swap in new semantics for the same syntactic expression!

Exercise 5

The folks down in hardware have finished our new custom CPU, so we'd like to target that from now on. The catch is that a stack-based architecture was chosen to save money. You need to write a version of your calculator that will emit assembly language for the new processor.

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

The hardware group has provided you with `StackVM.hs`, which is a software simulation of the custom CPU. The CPU supports six operations, as embodied in the `StackExp` data type:

```
data StackExp where
  PushI :: Integer -> StackExp
  PushB :: Bool    -> StackExp
  AddOp ::            StackExp
  MulOp ::            StackExp
  AndOp ::            StackExp
  OrOp  ::            StackExp
  deriving Show

type Program = [StackExp]
```

`PushI` and `PushB` push values onto the top of the stack, which can store both `Integer` and `Bool` values. `AddOp`, `MulOp`, `AndOp`, and `OrOp` each pop the top two items off the top of the stack, perform the appropriate operation, and push the result back onto the top of the stack. For example, executing the program

```
[PushB True, PushI 3, PushI 6, Mul]
```

will result in a stack holding `True` on the bottom, and `18` on top of that.

If there are not enough operands on top of the stack, or if an operation is performed on operands of the wrong type, the processor will melt into a puddle of silicon goo. For a more precise specification of the capabilities and behavior of the custom CPU, consult the reference implementation provided in `StackVM.hs`.

Your task is to implement a compiler for arithmetic expressions. Simply create an instance of the `Expr` type class for `Program`, so that arithmetic expressions can be interpreted as compiled programs. For any arithmetic expression `exp :: Expr a => a` it should be the case that

```
stackVM exp == Right [IVal exp]
```

Note that in order to make an instance for `Program` (which is a type synonym) you will need to enable the `TypeSynonymInstances` language extension, which you can do by adding

```
{-# LANGUAGE TypeSynonymInstances #-}
```

at the top of your file.

Finally, put together the pieces you have to create a function

```
compile :: String -> Maybe Program
```

which takes `String`s representing arithmetic expressions and compiles them into programs that can be run on the custom CPU.

Exercise 6

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

Some users of your calculator have requested the ability to give names to intermediate values and then reuse these stored values later.

To enable this, you first need to give arithmetic expressions the ability to contain variables. Create a new type class `HasVars a` which contains a single method `var :: String -> a`. Thus, types which are instances of `HasVars` have some notion of named variables.

Start out by creating a new data type `VarExprT` which is the same as `ExprT` but with an extra constructor for variables. Make `VarExprT` an instance of both `Expr` and `HasVars`. You should now be able to write things like

```
*Calc> add (lit 3) (var "x") :: VarExprT
```

But we can't stop there: we want to be able to interpret expressions containing variables, given a suitable mapping from variables to values. For storing mappings from variables to values, you should use the `Data.Map` module. Add

```
import qualified Data.Map as M
```

at the top of your file. The `qualified` import means that you must prefix `M.` whenever you refer to things from `Data.Map`. This is standard practice, since `Data.Map` exports quite a few functions with names that overlap with names from the `Prelude`. Consult the `Data.Map` documentation to read about the operations that are supported on `Maps`.

http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Map.html

Implement the following instances:

```
instance HasVars (M.Map String Integer -> Maybe Integer)

instance Expr (M.Map String Integer -> Maybe Integer)
```

The first instance says that variables can be interpreted as functions from a mapping of variables to `Integer` values to (possibly) `Integer` values. It should work by looking up the variable in the mapping.

To write these instances you will need to enable the `FlexibleInstances` language extension by putting

```
{-# LANGUAGE FlexibleInstances #-}
```

at the top of your file.

The second instance says that these same functions can be interpreted as expressions (by passing along the mapping to subexpressions and combining results appropriately).

Once you have created these instances, you should be able to test them as follows:

```
withVars :: [(String, Integer)]
         -> (M.Map String Integer -> Maybe Integer)
         -> Maybe Integer
withVars vs exp = exp $ M.fromList vs
```

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

```
*Calc> :t add (lit 3) (var "x")
add (lit 3) (var "x") :: (Expr a, HasVars a) => a
*Calc> withVars [("x", 6)] $ add (lit 3) (var "x")
Just 9
*Expr> withVars [("x", 6)] $ add (lit 3) (var "y")
Nothing
*Calc> withVars [("x", 6), ("y", 3)]
         $ mul (var "x") (add (var "y") (var "x"))
Just 54
```

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.