Weekly Assignments 3

To be submitted: Friday, 2 February 2018

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious. Please also pay attention to stylistic issues. The goal is always to submit code that does not just correctly do what was asked for, but also could be committed without further changes to an imaginary company codebase.

W3.1 Packaging

Prepare a Cabal package to contain all your solutions so that it can easily be built using cabal-install or stack.

Note that one package can contain one library (with arbitrarily many modules) and possibly several executables and test suites.

Please include a `README` file in the end explaining clearly where within the package the solutions to the individual subtasks are located.

Please do *NOT* try to upload your package to Hackage.

W3.2 Number of distinguishable values of datatypes

In the presence of laziness, datatypes contain more distinguishable values than one might expect.

Intuitively, values `a1` and `a2` of type `A` are distinguishable if there is a context in which `a1` and `a2` behave differently. For the purposes, of this exercise, we are going to consider functions

```
f :: A -> Int
```

and say `a1` and `a2` are distinguishable if `f a1` and `f a2` have different results. Note furthermore that for the purposes of this exercise, we treat all forms of nontermination and crashing as equivalent.

How many dinstinguishable Haskell values are there of the following types:

- `Bool`
- `Either Bool Bool`
- `(Bool, Bool)`
- `Maybe Bool`

- `Bool -> Bool`

For `(Bool, Bool)` only, write functions that distinguish all of the different values.

W3.3 Fixed points

Recall that a combinator `F` is called a fixed point combinator if `F f` is a fixed point of `f`, i.e., if

```
F f = f (F f)
```

The actual fixed point combinators from untyped lambda calculus such as

```
y = \ f -> (\ x -> f (x x)) (\ x -> f (x x))
```

do not type-check in Haskell. This is because they rely on self-application (such as seen in `x x`). In order for `x` to be applicable at all, it must be a function, i.e., of type `a -> a` for some `a`. But now the type of the argument must be the same as the type of the function, so `a = a -> a`. There is no type in Haskell that fulfills this equation.

However, since Haskell has general recursion built into the language, we can just use the fixed point property to define a fixed point combinator ourselves:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Subtask 3.3.1

Rewrite the function `map` on lists so that it does not make use of any recursion except for calling `fix`.

Subtask 3.3.2

What Prelude function does

```
mystery f x = fix $ (x :) . map f
```

correspond to?

Subtask 3.3.3

More useful in practice than `fix` on terms is `Fix` on types. We can define a fixed point combinator on types like this:

```
newtype Fix f = In { out :: f (Fix f) }
```

Observe that

```
In   ::  f (Fix f) -> Fix f
out  ::  Fix f -> f (Fix f)
```

now witness an isomorphism between `Fix f` and `f (Fix f)` (which is a bit weaker than the two types being equal, but good enough).

Given the datatype

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

define a datatype `TreeF` (not making use of `Tree` in the definition) such that `Fix (TreeF a)` and `Tree a` are isomorphic. Also define functions

```
fromTree :: Tree a -> Fix (TreeF a)
toTree   :: Fix (TreeF a) -> Tree a
```

that convert between the two representations of trees in such a way that they are mutual inverses.

W3.4 A simple stack language

Cryptocurrencies typically come with scripting languages. These scripting languages vary in power quite a bit. Ethereum has a Turing complete and complicated scripting language that is supposed to enable sophisticated smart contracts. Bitcoin still has a scripting language, but it is much more restricted. Bitcoin's language is actually a stack-based language. In this exercise, we are going to implement an extremely simple stack-based language and an evaluator for that language.

The abstract syntax of the language is given by

```
data Instructions =
    Push Int Instructions
  | Add Instructions
  | Mul Instructions
  | Dup Instructions
  | Swap Instructions
  | Neg Instructions
  | Pop Instructions
  | Over Instructions
  | IfZero Instructions Instructions
  | Loop (Instructions -> Instructions)
  | Halt
```

The description of each of the instructions is as follows:

- `Push` pushes the given integer on the top of the stack,
- `Add` removes the top two elements from the stack and pushes their sum,

- `Mul` removes the top two elements from the stack and pushes their product,
- `Dup` duplicates the top element of the stack,
- `Swap` swaps the top two elements of the stack,
- `Neg` negates the top element of the stack,
- `Pop` removes the top element from the stack,
- `Over` pushes a copy of the element just beyond the top on top of the stack (e.g., if the top element is `1` and the next element is `2`, then the new stack has top element `2`, followed by `1` and `2`)
- `IfZero` removes the top element of the stack, and if that element is `0`, executes the first set of instructions, otherwise it executes the second set of instructions,
- `Loop` executes the function, passing itself as an argument
- `Halt` stops execution.

The goal is to implement a function

```
run :: Instructions -> Maybe [Int]
```

that runs a set of instructions on an initially empty stack and returns the final stack. It should return `Nothing` if at any point in time, there are not sufficiently many elements on the stack.

You may want to use a suitable monad, but this is not a requirement.

Here is the factorial function as an example program:

```
fact5 :: Instructions
fact5 =
  Push 5 $
  Push 1 $
  Swap $
  Loop $ \ loop ->
  Dup $
  IfZero (Pop $ Halt) $
  Swap $
  Over $
  Mul $
  Swap $
  Push 1 $
  Neg $
  Add $
  loop
```

The very first `Push` is the argument. So this program should evaluate to the stack containing just `120`.

W3.5 A parser for stack programs

Define a parsec parser so that you have a concrete syntax for the abstract syntax given in exercise W3.4.

The goal is to be able to write the example program as follows:

```
{
  push 5;
  push 1;
  swap;
  loop {
    dup;
    ifzero {
      pop;
      halt
    } {
      swap;
      over;
      mul;
      swap;
      push 1;
      neg;
      add;
      ret
    }
  }
}
```

The grammar is

```
block  -> "{" instrs "}"
instrs -> simple ";" instrs | ctrl
simple -> "push" int | "add" | "mul" | "dup" | "swap" | "neg" | "pop" | "over"
ctrl   -> "ifzero" block block | "loop" block | "halt" | "ret"
int denotes an optionally signed integer number
```

A `ret` indicates that we want to return to the beginning of the loop. If a `ret` occurs outside of a loop, it is interpreted as `halt`.

Layout is not important. Arbitrary whitespace is allowed between any two tokens. As tokens, we consider all the literal strings that occur in the grammar.

Try to find a disciplined handling of whitespace (i.e., either write a separate lexer, or find a way to isolate whitespace handling so that it does not occur all over the place).

Write a function that reads a source file from disk, parses it, and if parsing is successful, runs it through the stack evaluator from the previous assignment.