This assignment tests your understanding of the content of Module 5. You will implement a functional language in GNU Smalltalk. As Assignment 2 was also the implementation of a functional language, you will find it to be a useful starting place. This assignment consists of two parts: The implementation of a functional language, and the implementation of a monad.

Submit all code via UWaterloo submit, e.g.:

`submit cs442 a3`.

This assignment is due on Wednesday, March 10th, by 12PM NOON, NOT MIDNIGHT, Eastern time.

Build You a Haskell

You are provided with `haskelltl.st`, an implementation of the types for the syntax tree of a (severely) reduced version of Haskell, called Haskell Total Landscaping, or Haskell TL for short. Haskell Total Landscaping is defined by the syntax and semantics in the appendix to Module 5, with the following syntactic sugar, simplifications, and additions:

- I/O is supported through the IO monad. IO monad objects are implemented as Smalltalk objects. Abstractions which evaluate to IO monad objects when applied are also implemented as Smalltalk objects.
- Let bindings are not part of the parsed syntax. Instead, they are simply replaced by abstractions and applications. For instance, `let x = y in z` is rewritten as `\x.z y`.
- Algebraic datatypes and matching are not implemented.
- A simple sort of tuple is supported as syntactic sugar, by replacing an expression such as `[3 1 4 1]` with a function which, given a value `n`, evaluates to the `n`th element of the tuple. For instance, `[3 1 4 1] 1` is 3, `[3 1 4] 2` is 1. If `n = 0`, the length of the tuple is returned, so `[3 1] 0` is 2. If `n` is too large, `error` is returned.
- Division is removed and replaced with an equality operator, `=`.
- A subtraction `x - y` where `x < y` should evaluate to 0, rather than getting stuck. This was our alternate definition of subtraction over natural numbers.

`haskelltl.st` implements the following classes (and extension to the `String` class):

```lisp
Object subclass: HaskellTLExp [ isVariable.
  isAbstraction.
  isApplication.
  isBoolLiteral.
  isTrue.
```
HaskellTLExp subclass: HaskellTLVar |
  HaskellTLVar class >> withName: name.
  dup.
  isVariable.
  name.
  freeVars: map.
  displayString.
|
HaskellTLExp subclass: HaskellTLAbs |
  HaskellTLAbs class >> withVar: var body: body.
  dup.
  isAbstraction.
  var.
  body.
  freeVars: map.
  displayString.
|
HaskellTLExp subclass: HaskellTLApp |
  HaskellTLApp class >> withRator: rator rand: rand.
  dup.
  isApplication.
  rator.
  rand.
  freeVars: map.
  displayString.
|
HaskellTLExp subclass: HaskellTLBoolLiteral |
  isBoolLiteral.
  isValue.
|
HaskellTLBoolLiteral subclass: HaskellTLLTrue |
  value.
  dup.
  isTrue.
  displayString.
|
HaskellTLBoolLiteral subclass: HaskellTLLFalse |
  value.
  dup.
  isFalse.
  displayString.
|
HaskellTLExp subclass: HaskellTLIf |
  HaskellTLIf class >> withCondition: scond thenExp: sthene elseExp: selsee.
  dup.
  isIf.
  condition.
  thenExp.
  elseExp.
  freeVars: map.
  displayString.
HaskellTLExp subclass: HaskellTLNum [  
  HaskellTLNum class >> withValue: value.  
    dup.  
    isNum.  
    isValue.  
    value.  
    displayString.  
]

HaskellTLExp subclass: HaskellTLNumExp [  
  HaskellTLNumExp class >> withOp: sop left: sleft right: sright.  
    dup.  
    isNumExp.  
    op.  
    left.  
    right.  
    displayString.  
]

HaskellTLExp subclass: HaskellTLError [  
    dup.  
    isError.  
    isValue.  
    displayString.  
]

HaskellTLExp subclass: HaskellTLOAbs [  
  HaskellTLOAbs class >> withConstructor: scons.  
    dup.  
    isIOAbstraction.  
    apply: exp.  
    displayString.  
]

HaskellTLExp subclass: HaskellTLIO [  
    isIO.  
    displayString.  
]

HaskellTLIO subclass: HaskellTLReadNum [  
    dup.  
    performIO: globals heap: heap.  
]

HaskellTLIO subclass: HaskellTLWriteNum [  
  HaskellTLWriteNum class >> new: sexp.  
    dup.  
    performIO: globals heap: heap.  
]

HaskellTLOAbs subclass: HaskellTLCurryBinding [  
  HaskellTLCurryBinding class >> new: sleft.  
    dup.  
    apply: exp.  
]

HaskellTLIO subclass: HaskellTLBinding [  
  HaskellTLBinding class >> withLeft: sleft right: sright.  
    dup.  
    performIO: globals heap: heap.  
]

Object subclass: HaskellTLParse [  
  HaskellTLParse class >> new.  
  HaskellTLParse class >> new: text.  
  HaskellTLParse class >> parse: text.  
  HaskellTLParse class >> parseFile: text withGlobals: globals.  
  parseFile: globals.  
  parse.
Object subclass: HaskellTLPrelude
    HaskellTLPrelude class >> globals.
]

String extend [
    runHaskellTL.
]

The HaskellTLExp, HaskellTLVar, HaskellTLAbs, and HaskellTLApp classes act exactly like the LambdaExp family of classes from Assignment 2. The other subclasses of HaskellTLExp are new types of expressions in Haskell Total Landscaping. New is* methods have been added for all the new subclasses, and the HaskellTLExp>>reduceWith: method has had its steps: argument removed, as we will only be performing full reductions in this assignment. In addition, the isValue method has been added, which returns true in all classes representing terminal values.

The HaskellTLLiteral class represents boolean literals, with its two subclasses HaskellTLLiteral and HaskellTLLiteralFalse.

The HaskellTLIf class represents an if-then-else expression. Its components can be extracted by the condition, thenExp, and elseExp methods.

The HaskellTLLNum class represents a numeric value. Its value can be extracted with value.

The HaskellTLLNumExp class represents a binary numeric expression. Its operator can be extracted as a string with op, and will always be one of '+', '-', '*', or '='. Its operands can be extracted with left and right. Remember that numbers, per the semantics, are all natural numbers (integers greater than or equal to zero).

The HaskellTLError class represents an error. Per the semantics, an error has no error message, and so nothing can be extracted from a HaskellTLError, except of course for the fact that it is an error.

The HaskellTLOAbs class represents an IO abstraction, such as Haskell's putStrLn. This has to be a separate kind of abstraction from HaskellTLAbs because the IO monad is a black box, in which you can perform no substitution. The apply: method is used to perform application on the HaskellTLOAbs with the given expression, returning a HaskellTLIO.

The HaskellTLIO class represents an IO monad. It is essentially a black box, but its performIO:heap: method performs the I/O, given a dictionary of globals (σ) and a heap dictionary (Σ) as arguments. Note that none of the built-in IO monads use the heap argument; you will use it in your own monad.

The HaskellTLLReadNum and HaskellTLLWriteNum classes represent our two IO monads, which read and write numbers as lines.

The HaskellTLCurryBinding and HaskellTLLBinding methods represent the single-argument curried and complete forms of a bound pair of IO monads.

The HaskellTLParser class is like LambdaParser, but with extended syntax for the new features of Haskell Total Landscaping. As well as all the methods that LambdaParser had, because Haskell TL has a file syntax with declarations, it additionally implements HaskellTLParser class>>parseFile:withGlobals: and parseFile:, both of which take a dictionary of pre-declared declarations (the standard library) and return the same dictionary with the declarations in the given file added. For instance, you can parse the file 'main = factorial 2; factorial = \x. if (= x 0) then 1 else * x (factorial (- x 1))' in either of these two ways, given a pre-existing variable globals:

f := 'main = factorial 2; factorial = \x. if (= x 0) then 1 else * x (factorial (- x 1))'.
p := HaskellTLParser new: f.
e := p parseFile: globals.

The HaskellTLPrelude class represents the standard library, i.e., Prelude. The HaskellTLPrelude class>>globals method returns a dictionary of pre-defined global variables; read its implementation to see what they are. This is a class method, so it’s called simply with HaskellTLPrelude globals.

Finally, the String class itself is extended with a runHaskellTL method, such that (once your assignment is
complete), you can run Haskell TL code by simply calling that method on a string containing Haskell TL code.

Several examples are included in the “demonstration” section of this document.

Note that we will only be testing code which evaluates to a number, boolean, error, or IO monad, or which gets stuck, not an abstraction. This is so that you can internally use de Bruijn indices or not, or rename variables as you please, or, frankly, perform reduction in any (correct) way you wish.

1 NOE reduction

Create a file, a3.st, which implements at least the following class and method:

```small
Object subclass: HaskellTL [  
   HaskellTL class >> new: exp withGlobals: dict.  
   eval.  
]
```

You can (and should!) start this by using the Lambda class you implemented in Assignment 2. Haskell TL’s reductions are based on NOR’s simplified form NOE, so you can use the nor and nor: methods as a starting point.

The eval method fully evaluates the expression stored in the HaskellTL with the global variables (σ) given by dict. If the expression reduces forever, eval should never return (i.e., don’t try to solve the halting problem). Note that since Haskell Total Landscaping does not include let bindings, σ never changes. Make sure you do not reduce inside an abstraction, or reduce the rand of an application, as those are standard in NOR but not done in Haskell. It is invalid to call eval twice on the same HaskellTL or expression, so you may update the expression in any way you please.

Be careful to use dup when appropriate. Just like in Assignment 2, if you mutate the expression, you will need to be careful to dup during substitution so that you don’t have multiple references to the same mutable expression. In this case, as extracting variables from the store is similar to substitution, you will have to be careful to dup there as well if you mutate expressions.

Since there is no method to take a single step of reduction, you cannot and will not be judged on whether your individual reduction steps work as expected. If you wish to implement reduction in a very different way, you are free to. Just make sure you use lazy evaluation, as expressions which terminate under NOR but reduce forever under AOR or AOE will absolutely be tested. In addition, “getting stuck” behavior will be tested, including but not limited to:

- If the condition of an if-then-else does not evaluate to a boolean, then the reduction should get stuck.
- If a numeric binary expression is used and either of the operands does not reduce to a number, then the reduction should get stuck. Note that this is also true of the added = operator, which should only check equality of numbers, not arbitrary expressions.
- If the rator of an application does not reduce to a HaskellTLAbs or a HaskellTLIOAbs, then the reduction should get stuck.

Remember to implement semantics for every non-terminal HaskellTL* class. In an HaskellTLApp in which the rator is a HaskellTLIOAbs (i.e., its isIOAbstraction returns true), the correct semantics is to reduce to the result of rator apply: rand. Just like any other application, do not reduce the rand before calling apply:. Subclasses of HaskellTLIO are terminal values, and cannot be reduced; do not try to use performIO: as reduction!

Be careful about types in your reduction. Reduction should never result in a value that is not a HaskellTLExp. For instance, when adding two numbers, you should create a HaskellTINum with the actual result as its value, not just a number. When performing an equality comparison, you should create a HaskellTLTrue or HaskellTLFalse, not just true or false.
2 State Monad

Extend a3.st, adding the following class:

```small
Object subclass: HaskellTLRefLib |
    HaskellTLRefLib class >> globals: dict.
|
```

The HaskellTLRefLib class>>globals: method should extend the dictionary given as an argument with entries 'ref', 'get', and 'put', which implement the state monad as IO monads. 'ref' and 'get' should be abstractions (presumably HaskellTLIOAbs values) which evaluate to IO monads of your own creation; you may want to look at the implementation of HaskellTLWriteNum and 'writeNum' for reference. 'put' should be an abstraction which, when given two arguments, evaluates to an IO monad of your own creation; you may want to look at the implementation of HaskellTLCurryBinding, HaskellTLBinding, and 'bind' for reference.

Note that since Haskell TL is lazily evaluated, the arguments to each of these will be stored in the IO monad unevaluated. It is up to each monad implementation to actually evaluate them.

The 'ref' monad's performIO:heap: method should fully evaluate its argument, then create a new entry in the heap dictionary, mapping a fresh label to that value. Its return value should be a Haskell TL value associated with the label. Exactly how you implement labels is up to you; perhaps labels are simply numbers, perhaps they are variables, perhaps they are a new subclass of HaskellTLExp. The important thing is that labels are unique. In addition, as you are writing the only component that cares about the heap, you are free to use other keys in the heap to store information needed to create fresh labels.

The 'get' monad's performIO:heap: method should fully evaluate its argument, interpret it as a label, and return the value associated with that label in the heap. No error checking is required, so any behavior is allowed if the argument is not a label, or the label does not exist in the heap.

The 'put' monad's performIO:heap: method should fully evaluate both of its arguments, and interpret the first as a label and the second as a value. The heap should be updated such that the value associated with the label is replaced with the value given as a second argument. The return is the Haskell TL value associated with the label, not the value.

Hints

To implement this, you will need to implement several other classes. Those classes will not be tested directly, only via their behavior in the globals created by HaskellTLRefLib. A suggested, but not required, design for those classes is:

```small
HaskellTLIO subclass: HaskellTLRef |
    HaskellTLRef class >> new: exp.
    dup.
    performIO: globals heap: heap.
|
HaskellTLIO subclass: HaskellTLGet |
    HaskellTLGet class >> new: exp.
    dup.
    performIO: globals heap: heap.
|
HaskellTLIOAbs subclass: HaskellTLCurryPut |
    HaskellTLCurryPut class >> new: sleft.
    dup.
    apply: exp.
|
HaskellTLIO subclass: HaskellTLPut |
    HaskellTLPut class >> withLeft: sleft right: sright.
    dup.
    performIO: globals heap: heap.
```

Demonstration

We will demonstrate the various features step-by-step. If you’ve partially implemented HaskellTL>>eval but not HaskellTLRefLib class>>globals, you can stub out the latter with a trivial non-implementation:

```small
Object subclass: HaskellTLRefLib [ HaskellTLRefLib class >> globals: dict [ ^dict ] ]
```

which of course won’t work, but will allow String>>runHaskellTL to operate.

Basics

Haskell Total Landscaping is the λ-calculus at its heart, and so anything you could do in the λ-calculus, you can do in Haskell TL, as well as recursion without the Y combinator. Note that we will not be testing examples of this type, as we won’t be testing abstractions for equality (so that you can continue to use de Bruijn indices or any form of substitution); still, λ-calculus expressions should behave in a predictable way, albeit with less reduction than nor:

```small
st> 'two = \f. \x. f (f x); three = \f. \x. f (f (f x)); mul = \m. \n. \f. m (n f); main = mul two three' runHaskellTL
(\f.(two (three f)))
```

None of this is interestingly changed from Assignment 2, so we will focus on other features.

Numbers and Math

Haskell TL should implement natural numbers and several operators. Just like Haskell, we can use this to convert Church numerals into natural numbers:

```small
st> 'two = \f. \x. f (f x); three = \f. \x. f (f (f x)); mul = \m. \n. \f. m (n f); main = mul two three (\x. (+ x 1)) 0' runHaskellTL
6
```

Of course, this could be done without Church numerals:

```small
st> 'main = * 2 3' runHaskellTL
6
```

Subtraction operates within the natural numbers:

```small
st> 'main = - 2 3' runHaskellTL
0
st> 'main = - 3 2' runHaskellTL
1
```

And of course, all of it uses λ-calculus-like application syntax, rather than Haskell-like infix operator syntax (i.e., the operator doesn’t go in between its operands).

Booleans, Conditions, and Recursion

While it’s possible to implement λ-calculus-style booleans, Haskell TL also supports native booleans:

```small
st> 'main = true' runHaskellTL
true
st> 'main = false' runHaskellTL
false
```
As well as numeric comparison:

```haskell
st> 'main = 1 1' runHaskellTL
true
st> 'main = 1 2' runHaskellTL
false
st> 'main = 1 (- 2 1)' runHaskellTL
true
```

With the addition of `if-then-else`, it’s easy to write recursive functions:

```haskell
st> 'factorial = \x. if (= x 0) then 1 else * x (factorial (- x 1)); main = factorial 12' runHaskellTL
479001600
```

You can store your numbers as Smalltalk integers, which are 64-bit on 64-bit systems, so don’t be concerned about overflow:

```haskell
st> 'factorial = \x. if (= x 0) then 1 else * x (factorial (- x 1)); main = factorial 21' runHaskellTL
-4249290049419214848
```

Several other numeric comparisons which evaluate to booleans are implemented in `HaskellTLPrelude`, but they are simply implemented in terms of `=`:

```haskell
st> 'isFiveToTen = \x. if (le x 10) then (ge x 5) else false; main = isFiveToTen 5' runHaskellTL
true
st> 'isFiveToTen = \x. if (le x 10) then (ge x 5) else false; main = isFiveToTen 11' runHaskellTL
false
```

Errors

Haskell Total Landscaping errors—if the whole language isn’t an error—are pervasive, and should persist past all obstacles. In actuality, though, they will only be tested as rators or rands of applications:

```haskell
st> 'main = error 42' runHaskellTL
error
st> 'infinite = \x. infinite (+ x 1); main = infinite error' runHaskellTL
error
```

Syntactic Sugar

Let bindings and tuples are just syntactic sugar. You do not need to do anything directly to implement them:

```haskell
st> 'main = let x = 42 in x' runHaskellTL
42
st> 'main = [10 21 42] 3' runHaskellTL
42
```

IO

If `main` resolves to an IO monad, then `runHaskellTL` will call `performIO:heap:` on that IO monad, thus performing I/O:

```haskell
st> 'main = writeNum 42' runHaskellTL
42
42
```

Note that in the above example, 42 is written twice because the first is the behavior of the `writeNum` monad, and the second is the behavior of the GNU Smalltalk REPL. Monadic binding is done through the `bind` function, which behaves like `>>=`:

```haskell
st> 'main = bind readNum (\x. writeNum (+ x 30))' runHaskellTL
12
42
42
```
In this case, the 12 is the keyboard input, the first 42 is the behavior of `writeNum`, and the third is the behavior of the GNU Smalltalk REPL. As the result of binding is itself an IO monad, further bindings can be performed, creating arbitrarily long chains of monads:

```
12
31
32
32
32
```

There is no `unsafePerformIO`.

### State

State is stored with references and the `get` and `put` monads. Here is an example that repeatedly decrements the value in a reference until it reaches zero:

```
31
32
32
```

Here is the example code, written with clearer indentation:

```
findZeroBadly = \r. bind (get r) \x. if (= x 0) then (writeNum x) else bind (put r (- x 1)) \ig.
    findZeroBadly r;
main = bind (ref 42) \r. findZeroBadly r
```

And, the example code with clearer indentation:

```
findZeroBadly = 
    \r. 
        bind (get r) \x.
            if (= x 0) then
                (writeNum x)
            else
                bind (put r (- x 1)) \ig.
                findZeroBadly r;
main = 
    bind (ref 42) \r.
        findZeroBadly r
```

Here is an example which computes the factorial of a number, with both the number and the result in references:

```
36
36
36
36
36
```

And, the example code with clearer indentation:

```
refFac = 
    \i. \o.
        bind (get i) \x.
            bind (get o) \y.
                if (= x 0) then
                    writeNum y
                else
                    bind (put i (- x 1)) \ig.
                    bind (put o (* x y)) \ig.
                    refFac i o;
        main = bind (ref 10) \i. bind (ref 1) (refFac i)'
runHaskellTL
3628800
3628800
```

And, the example code with clearer indentation:

```
refFac = 
    \i. \o.
        bind (get i) \x.
            bind (get o) \y.
                if (= x 0) then
                    writeNum y
                else
                    bind (put i (- x 1)) \ig.
                    bind (put o (* x y)) \ig.
                    refFac i o;
        main = 
            bind (ref 10) \i.
                bind (ref 1) (refFac i)
Finally, to put it all together, here is a program which takes a number as input, and then as many times as that number specifies, takes other numbers as inputs, and prints the factorial of those numbers:

```haskell
refFac = \i. \o. bind (get i) \x. bind (get o) \y. if (= x 0) then writeNum y else bind (put i (- x 1)) \ig. bind (put o (* x y)) \ig. refFac i o; refFacIn = bind readNum \i. bind (ref i) \i. bind (ref 1) \o. refFac i o; refFacN = \i. if (le i 1) then refFacIn else bind refFacIn \ig. (refFacN (- i 1)); main = bind readNum refFacN
```

And, its clearer version:

```haskell
refFac =
  \i. \o.
  bind (get i) \x.
  bind (get o) \y.
  if (= x 0) then
    writeNum y
  else
    bind (put i (- x 1)) \ig.
    bind (put o (* x y)) \ig.
    refFac i o;

refFacIn =
  bind readNum \i.
  bind (ref i) \i.
  bind (ref 1) \o.
  refFac i o;

refFacN =
  \i.
  if (le i 1) then
    refFacIn
  else
    bind refFacIn \ig. (refFacN (- i 1));

main = bind readNum refFacN
```

Rights

Copyright © 2020, 2021 University of Waterloo.
This assignment is intended for CS442 at University of Waterloo.
Any other use requires permission from the above named copyright holder(s).