CS442
Assignment 3

University of Waterloo

Winter 2023

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.

In this and all assignments, any behavior which we do not explicitly define will not be tested, so you may define it however you wish, or allow your program to fail. Make sure though that it actually *is* undefined; ask on Piazza if you're unsure.

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

```
submit cs442 a3 .
```

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

Build You a Haskell

You are provided with `hasntkell.st`, an implementation of the types for the syntax tree of a (severely) reduced version of Haskell, called Hasn't Kell, or Hasntkell for short. Hasn't Kell 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 nth 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.

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

```
Object subclass: HasntkellExp [
    isVar.
    isAbs.
    isApp.
    isBoolLiteral.
    isTrue.
    isFalse.
    isIf.
    isNum.
    isNumExp.
    isError.
    isIOAbs.
    isIO.
    isValue.
    reduceWith: block.
    freeVars.
    freeVars: map.
    printString.
]

HasntkellExp subclass: HasntkellVar [
    HasntkellVar class >> withName: name.
    dup.
    isVar.
    name.
    freeVars: map.
    displayString.
]

HasntkellExp subclass: HasntkellAbs [
    HasntkellAbs class >> withVar: var body: body.
    dup.
    isAbs.
    var.
    body.
    freeVars: map.
    displayString.
]

HasntkellExp subclass: HasntkellApp [
    HasntkellApp class >> withRator: rator rand: rand.
    dup.
    isApp.
    rator.
    rand.
    freeVars: map.
    displayString.
]

HasntkellExp subclass: HasntkellBoolLiteral [
    isBoolLiteral.
    isValue.
]

HasntkellBoolLiteral subclass: HasntkellTrue [
    value.
    dup.
    isTrue.
    displayString.
]

HasntkellBoolLiteral subclass: HasntkellFalse [
    value.
    dup.
    isFalse.
    displayString.
]
```

```
HasntkellExp subclass: HasntkellIf [
    HasntkellIf class >> withCondition: scond thenExp: sthene elseExp: selsee.
    dup.
    isIf.
    condition.
    thenExp.
    elseExp.
    freeVars: map.
    displayString.
]

HasntkellExp subclass: HasntkellNum [
    HasntkellNum class >> withValue: value.
    dup.
    isNum.
    isValue.
    value.
    displayString.
]

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

HasntkellExp subclass: HasntkellError [
    dup.
    isError.
    isValue.
    displayString.
]

HasntkellExp subclass: HasntkellIOAbs [
    HasntkellIOAbs class >> withConstructor: scons.
    dup.
    isIOAbstraction.
    apply: exp.
    displayString.
]

HasntkellExp subclass: HasntkellIO [
    isIO.
    displayString.
]

HasntkellIO subclass: HasntkellReadNum [
    dup.
    performIO: globals heap: heap.
]

HasntkellIO subclass: HasntkellWriteNum [
    HasntkellWriteNum class >> new: sexp.
    dup.
    performIO: globals heap: heap.
]

HasntkellIOAbs subclass: HasntkellCurryBinding [
    HasntkellCurryBinding class >> new: sleft.
    dup.
    apply: exp.
]
```

```
HasntkellIO subclass: HasntkellBinding [
    HasntkellBinding class >> withLeft: sleft right: sright.
    dup.
    performIO: globals heap: heap.
]

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

Object subclass: HasntkellPrelude [
    HasntkellPrelude class >> globals.
]

String extend [
    runHasntkell.
]
```

The `HasntkellExp`, `HasntkellVar`, `HasntkellAbs`, and `HasntkellApp` classes act exactly like the `LambdaExp` family of classes from Assignment 2. The other subclasses of `HasntkellExp` are new types of expressions in Hasn't Kell. New `is`* methods have been added for all the new subclasses, and the `HasntkellExp>>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 `HasntkellBoolLiteral` class represents boolean literals, with its two subclasses `HasntkellTrue` and `HasntkellFalse`.

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

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

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

The `HasntkellIOAbs` class represents an IO abstraction, such as Haskell's `putStrLn`. This has to be a separate kind of abstraction from `HasntkellAbs` 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 `HasntkellIOAbs` with the given expression, returning a `HasntkellIO`.

The `HasntkellIO` 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 `HasntkellReadNum` and `HasntkellWriteNum` classes represent our two IO monads, which read and write numbers as lines.

The `HasntkellCurryBinding` and `HasntkellBinding` methods represent the single-argument curried and complete forms of a bound pair of IO monads.

The `HasntkellParser` class is like `LambdaParser`, but with extended syntax for the new features of Hasn't Kell. As well as all the methods that `LambdaParser` had, because Hasntkell has a file syntax with declarations, it additionally implements `HasntkellParser 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 := HasntkellParser new: f.
e := p parseFile: globals.
...
e := HasntkellParser parseFile: f withGlobals: globals.
```

The `HasntkellPrelude` class represents the standard library, i.e., `Prelude`. The `HasntkellPrelude 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 `HasntkellPrelude globals`.

Finally, the `String` class itself is extended with a `runHasntkell` method, such that (once your assignment is complete), you can run Hasntkell code by simply calling that method on a string containing Hasntkell 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, and will *not* be testing any code that evaluates to an abstraction, so variable naming is not a concern. 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:

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

You can (and should!) start this by using the `Lambda` class you implemented in Assignment 2. Hasntkell'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 `Hasntkell` 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 Hasn't Kell 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 `Hasntkell` 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 `HasntkellAbs` or a `HasntkellIOAbs`, then the reduction should get stuck.

Remember to implement semantics for every non-terminal `Hasntkell`* class. In a `HasntkellApp` in which the rator is a `HasntkellIOAbs` (i.e., its `isIOAbs` 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 `HasntkellIO` 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 `HasntkellExp`. For instance, when adding two numbers, you should create a `HasntkellNum` with the actual result as its value, not just a number. When performing an equality comparison, you should create a `HasntkellTrue` or `HasntkellFalse`, not just `true` or `false`.

2 State Monad

Extend `a3.st`, adding the following class:

```
Object subclass: HasntkellRefLib [
    HasntkellRefLib class >> globals: dict.
]
```

The `HasntkellRefLib` `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 `HasntkellIOAbs` values) which evaluate to IO monads of your own creation; you may want to look at the implementation of `HasntkellWriteNum` 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 `HasntkellCurryBinding`, `HasntkellBinding`, and `'bind'` for reference.

Note that since Hasntkell 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 Hasntkell 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 `HasntkellExp`. 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'` abstraction should evaluate to a `'put'` monad black box when called with two arguments, M_1 and M_2. The monad's `performIO:hea:` method should fully evaluate *both* of its arguments, and interpret M_1 (the first) as a label ℓ, and M_2 (the second) as a value v. The heap should be updated such that the value v is associated with the label ℓ. The return is ℓ, not v.

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 `HasntkellRefLib`. A suggested, but not required, design for those classes is:

```
HasntkellIO subclass: HasntkellRef [
    HasntkellRef class >> new: sexp.
    dup.
    performIO: globals heap: heap.
]

HasntkellIO subclass: HasntkellGet [
    HasntkellGet class >> new: sexp.
    dup.
    performIO: globals heap: heap.
]
```

```
HasntkellIOAbs subclass: HasntkellCurryPut [
    HasntkellCurryPut class >> new: sleft.
    dup.
    apply: exp.
]

HasntkellIO subclass: HasntkellPut [
    HasntkellPut 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 `Hasntkell>>eval` but not `HasntkellRefLib class>>globals:`, you can stub out the latter with a trivial non-implementation:

```
Object subclass: HasntkellRefLib [
    HasntkellRefLib class >> globals: dict [
        ^dict
    ]
]
```

which of course won't work, but will allow `String>>runHasntkell` to operate.

Basics

Hasn't Kell is the λ-calculus at its heart, and so anything you could do in the λ-calculus, you can do in Hasntkell, 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`:

```
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
    ' runHasntkell
(^f.(two (three f)))
st> 'main = (^x. x) (^y. y)' runHasntkell
(^y.y)
```

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

Globals

The λ-calculus has no direct reduction for variables, but Hasntkell does. If the variable is present in the global scope, then the Variable rule applies:

```
st> 'fortytwo = 42; main = fortytwo' runHasntkell
42
st> 'main = fortytwo' runHasntkell
fortytwo
```

Be careful about evaluation order. If you replace variables too early, your result will be incorrect:

```
st> 'fortytwo = 14; main = (^fortytwo. fortytwo) 42' runHasntkell
42
" This would have returned 14 if we'd replaced fortytwo at the wrong time "
```

Numbers and Math

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

```
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' runHasntkell
6
```

Of course, this could be done without Church numerals:

```
st> 'main = * 2 3' runHasntkell
6
```

Subtraction operates within the natural numbers:

```
st> 'main = - 2 3' runHasntkell
0
st> 'main = - 3 2' runHasntkell
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, Hasntkell also supports native booleans:

```
st> 'main = true' runHasntkell
true
st> 'main = false' runHasntkell
false
```

As well as numeric comparison:

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

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

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

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

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

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

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

Errors

Hasn't Kell 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:

```
st> 'main = error 42' runHasntkell
error
st> 'infinite = ^x. infinite (+ x 1); main = infinite error' runHasntkell
error
```

Syntactic Sugar

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

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

IO

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

```
st> 'main = writeNum 42' runHasntkell
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 >>=:

```
st> 'main = bind readNum (^x. writeNum (+ x 30))' runHasntkell
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:

```
st> 'echo = bind readNum (^x. writeNum (+ x 30)); main = bind echo (^x. echo)' runHasntkell
1
31
2
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:

```
st> 'findZeroBadly = ^r. bind (get r) ^x. if (= x 0) then (writeNum x) else bind (put r (- x 1)) ^ig.
    findZeroBadly r ; main = bind (ref 42000) ^r. findZeroBadly r' runHasntkell
0
0
```

This example may take a moment (and the garbage collector may holler a lot), but if your implementation of NOE is correct, it should be impossible to overflow the stack with this kind of code.

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
```

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

```
st> '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)'
    runHasntkell
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:

```
st> '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' runHasntkell
3
1
1
5
120
20
2432902008176640000
2432902008176640000
```

And, its clearer version:

```
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