2015 Haskell January Test
Functions, Procedures and Memoisation

This test comprises four parts and the maximum mark is 30. Parts I, II and III are worth 28 of the 30 marks available. The **2015 Haskell Programming Prize** will be awarded for the best overall solution.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

Memoisation is an optimisation that involves remembering the values returned by a function in order to avoid computing the same value more than once. The mapping between a function's argument value(s) and its corresponding result is stored in a table, the *memo table* for that function, and if the function is called again with the same argument(s) the result is retrieved from the table, rather than being re-computed. In order for memoisation to work the function being memoised must return the *same* answer when given the same argument value(s), i.e. they must be 'pure' functions, like those in Haskell. Memoisation can sometimes change the fundamental complexity of a function, but this must be traded off against the increased cost, in both time and space, of maintaining a memo table.

1.1 An Example

The classical example that illustrates the idea is the function to compute the n^{th} number in the *Fibonacci* sequence, $fib(n), n \geq 0$, which begins $0, 1, 1, 2, 3, 5, 8, ...$ and so on. The *zeroth* number is defined to be 0, the first is 1 and the successive numbers are given by the sum of the previous two. To simplify the examples in this exercise we'll omit the zeroth number and will consider the sequence to begin $1, 1, 2, 3, ...$ etc. In that case the function that computes $fib(n)$, $n > 0$, can be written in Haskell as follows:

```
fib :: Int -> Int
-- Pre: n > 0
fib n
  = if n < 3 then 1 else fib (n - 1) + fib (n - 2)
```

The call graph for the execution of `fib n` is shown in Figure 1 (top). The nodes of the graph show the argument values for the calls to `fib` that will be made during a standard evaluation of `fib n`, e.g. using GHC(i). Notice that there is an enormous amount of redundancy, e.g. the call graph for `fib (n - 1)` appears once; that of `fib (n - 2)` twice, `fib (n - 3)` three times, and so on. Indeed, `fib (n - k)` is computed `fib (k + 1)` times during the evaluation of `fib n`, $0 \leq$ `k` $\leq n$ and it can be shown that the overall execution time for `fib n` is *exponential* in `n`.

Now suppose that when we compute the value of `fib n` for the first time we associate `n` with the computed value of `fib n` in a memo table, e.g. by assigning that value to the $\mathtt{n}^{th}$ element of a Java-like array. The idea is to retrieve the saved value from the memo table if we later attempt to recompute the same expression, `fib n`. The effect of the memoisation in this case is to eliminate the redundant computations entirely, so that `fib n` is evaluated exactly once for all `n` > 0. With memoisation the call graph now essentially shares nodes with identical values, as shown in Figure 1 (bottom): two or more references to a node implies that the first reference will compute the value and memoise it and the remaining references will extract the saved value from the memo table.

Note that we need to be able to detect whether or not a memo table entry has been previously computed and this can be done using a corresponding table of booleans: the n^{th} element will be true if the n^{th} element of the memo table has been computed; false otherwise. Note also that, as described, the boolean and memo tables need to be *updated* during the execution of the program – this is easy to do in a language like Java, but is not so easy in Haskell. We'll therefore be exploring memoisation in a very simple *procedural* language for which you will be required to implement an interpreter. As it happens, the bulk of the exercise is concerned with the development of this interpreter.

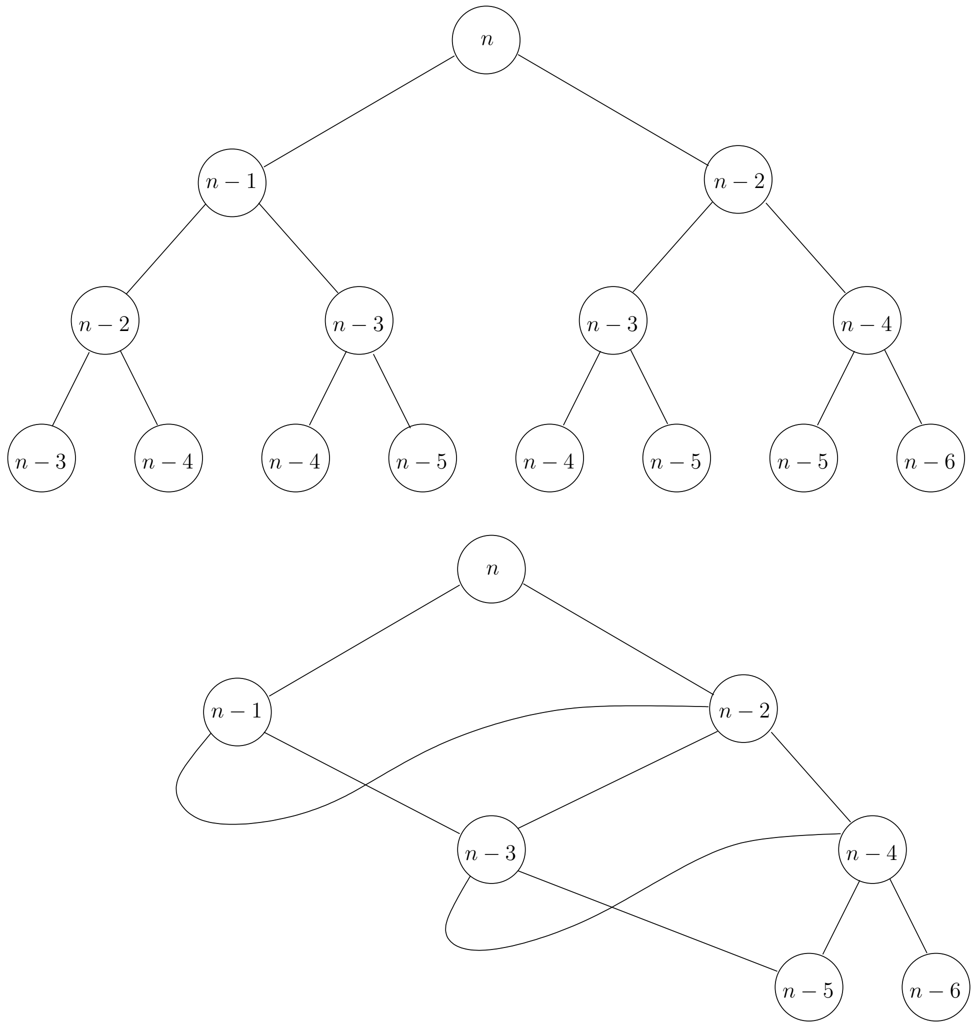

Figure 1: The Fibonacci call pattern before and after memoisation.

2 The Language

The language you will be working with is a very simple procedural language containing an embedded purely functional sublanguage. We'll begin with this sublanguage.

2.1 Expressions and 'pure' functions

The functional component of the language is centred around *expressions* which can be one of:

1. A *constant* value. The language supports just two value types: *integers* and *one-dimensional arrays* of integers (see below). Note that booleans are not supported directly, so they are encoded as integers: true will be represented by the integer 1 and false by 0.
2. A *variable* reference. User-defined variable identifiers will always begin with a letter, e.g. `x`, `y`, `y'`, `VAR`, `camelCase` etc.

3. A *binary operator application.* The operators supported are addition, multiplication, less-than, equal and index. The first four, corresponding to `+`, `*`, `<` and `==` in Haskell, take integer parameters only (we can't compare two arrays) and produce an integer result. For both less-than and equal the results will be either 0 or 1, representing false and true respectively. The index operator implements array indexing: the first argument is the array and the second the (integer) index.
4. A *conditional* expression comprising a predicate and 'then' and 'else' expressions akin to Haskell's `if P then Q else R` or Java's *ternary operator* `P ? Q : R`.
5. A *user-defined function application.* Functions in the language are first-order (no partial application) and a function definition comprises the name of the function, a list of argument identifiers and a right-hand side expression, akin to an uncurried Haskell function definition like `f(x1, x2, ..., xn) = e`.

Programs in this functional sublanguage can be defined at an abstract level by the following Haskell type definitions:

```
type Id = String

data Value = I Int | A [(Int, Int)]
           deriving (Eq, Show)

data Op = Add | Mul | Less | Equal | Index
        deriving (Eq, Show)

data Exp = Const Value |
           Var Id |
           OpApp Op Exp Exp |
           Cond Exp Exp Exp |
           FunApp Id [Exp]
         deriving (Eq, Show)

type FunDef = (Id, ([Id], Exp))
```

2.2 Arrays and array indexing

Integers and arrays are distinguished by the constructors `I` and `A` in the `Value` data type. The elements of an array are represented by a list of pairs that maps an array index to a value. For example, an array represented by `A [(3, 1), (0, 3), (5, 12)]` has values defined at indices 0, 3 and 5 so indexing the array at these positions will return values 3, 1 and 12 respectively. Note that the order of the elements within the mapping list is unimportant and that indices do not have to form a contiguous range. Furthermore, there should be no duplicate entries for a particular index: an array with values at n defined (and unique) indices should be represented by a mapping list with exactly n elements. The value of an array at an unspecified index is assumed to be zero, so an attempt to index the above array at positions 1, 2, 4, 6, 7... will deliver 0. In this sense arrays in the language are *unbounded* even though their representations are finite.

2.3 Examples

To illustrate the language at the source level we'll invent a "pseudocode" notation that is a mixture of Haskell- and Java-like syntax. You won't need to manipulate source code – we'll just use it to help clarify the examples. As an example, the function `biggest` that returns the biggest of two integers might be written in pseudocode as:

```
biggest(m, n)
  = if m < n then n else m
```

This will be represented by the following `FunDef`:

```
biggest :: FunDef
biggest
  = ("biggest",
     (["m", "n"], Cond (OpApp Less (Var "m") (Var "n"))
                       (Var "n")
                       (Var "m")
     )
    )
```

As another example, the `fib` function defined above would be written identically in pseudocode notation, but without the type declaration. This can be represented by:

```
fib :: FunDef
fib
  = ("fib",
     (["n"], Cond (OpApp Less (Var "n") (Const (I 3)))
                  (Const (I 1))
                  (OpApp Add (FunApp "fib" [OpApp Add (Var "n") (Const (I (-1)))])
                             (FunApp "fib" [OpApp Add (Var "n") (Const (I (-2)))]))
     )
    )
```

To illustrate array indexing, the following sums elements `0`...`n` of an array `a`:

```
sumA(a, n)
  = if n < 0 then 0 else a[n] + sumA(a, n - 1)
```

and this will be represented thus:

```
sumA :: FunDef
sumA
  = ("sumA",
     (["a", "n"], Cond (OpApp Less (Var "n") (Const (I 0)))
                       (Const (I 0))
                       (OpApp Add (OpApp Index (Var "a") (Var "n"))
                                  (FunApp "sumA"
                                     [Var "a", OpApp Add (Var "n")
                                                         (Const (I (-1)))]))
     )
    )
```

2.4 Expression evaluation

An evaluator for expressions will need to carry around a list of user-defined function definitions (`FunDefs`), in order to implement function application (`FunApp`), and a *state*, which will provide the values of the variables that are in scope, e.g. the values of the function's arguments. The evaluator will treat the latter as *local* variables. Later on we'll need also to consider *global* variables, in order to implement memo tables. In general, therefore, a state may contain both local and global variables, so we introduce the following types:

```
data Scope = Local | Global
           deriving (Eq, Show)

type Binding = (Id, (Scope, Value))

type State = [Binding]
```

For the purposes of expression evaluation it matters not whether a variable is local or global, so the `Scope` field can be ignored for now.

The result of an expression evaluation is an object of type `Value`, which may may be either an integer of the form `I n` for some integer `n`, or an array of the form `A elems` where `elems` is the list mapping indices to values.

At this point you are in a position to implement Parts I and II. You may wish to complete these before reading on.

3 Procedures

The rest of the language is procedural and can be thought of as a very simple stripped-down version of Java, without objects. A procedure (a 'method' in Java parlance) comprises the procedure name and a list of argument names, as for a function. The difference is that the right-hand side of a procedure is a list of *statements*, which we shall refer to as a *block*. Each statement can be one of:

1. An *assignment to a variable*, e.g. `x = e` in pseudocode syntax, where `e` is an expression, as above. If the variable is already in scope then its value is replaced with the value of the expression on the right-hand side. Otherwise the assignment serves to introduce `x` as a new *local* variable.

2. An *assignment to an array element*, e.g. `a[i] = e` in pseudocode, where `a` is an array, `i` is an integer index and `e` is an expression. For the program to be well formed the array variable `a` must already be in scope, i.e. an element assignment of the form `a[i] = e` cannot be used to define a new array `a`.

3. An *'if' statement*, equivalent to a Java statement of the form `if (P) then {B} else {B'}`, where `P` is a boolean-valued expression (predicate) and `B` and `B'` are blocks. The predicate `P` can be assumed to evaluate to either false (0) or true (1). Note that 'if' *statements* should not be confused with conditional *expressions* above.

4. A *while loop*, e.g. `while (P) {B}` in pseudocode, where `p` is a predicate and `b` is a block of statements. The predicate `P` can be assumed to evaluate to either false (0) or true (1) and the block `B` is executed repeatedly whilst the predicate is true.

5. A *user-defined procedure call.* If a procedure returns a value then that value can be bound to a variable, as in `x := p(e1, ..., en)` in pseudocode, where `p` is the procedure name and the `ei` are the argument expressions; if not, then the procedure call will be of the form `p(e1, ..., en)`. Note that procedure calls are *not* expressions in this language[1], so binding the result of a procedure call is *not* the same as assigning an expression's value to a variable. A 'binding' statement (`:=`) in the pseudocode is thus used to make the distinction clear.

6. A *return* statement, e.g. `return e` in pseudocode, where `e` is an expression. It can be assumed that a return statement will always be the final statement to be executed within a block.

Procedures, as defined, can be represented by the following Haskell types:

```
type Block = [Statement]

data Statement = Assign Id Exp |
                 AssignA Id Exp Exp |
                 If Exp Block Block |
                 While Exp Block |
                 Call Id Id [Exp] |
                 Return Exp
               deriving (Eq, Show)

type ProcDef = (Id, ([Id], Block))
```

Note that a procedure definition (`ProcDef`) is similar to a function definition (`FunDef`) except that the right-hand side is a block rather than an expression.

A procedure call of the form `x := p(e1, ..., en)` in pseudocode will be represented by the expression `Call "x" "p" [e1, ..., en]`. If the procedure does not return a result then the first `Id` in the representation will be `""`, as in: `Call "" "p" [e1, ..., en]`.

3.1 Block execution

Because a procedure can invoke both user-defined functions (during evaluation of an `Exp`) and user-defined procedures, an executor for statements must carry around both a list of `FunDef`s, as above, and a list of `ProcDef`s. As with the expression evaluator it will also need to access the values of the variables that are in scope via an additional `State` parameter.

The state passed to a procedure should contain bindings for the procedure's argument values and the values of the *global* variables at the point of call. The local variables that are in scope at the point of call should not be included as they are not visible outside the calling procedure. Importantly, if an argument has the same name as one of the global variables then the argument takes precedence over the global variable with regards its scope. The global variable in that case will thus be 'invisible' during the execution of the procedure body.

Any changes made to the global variables in the called procedure will be reflected in the state returned from it. The state after a call should therefore be comprised of the local variables that existed in the state prior to the call together with the (possibly modified) global variables in the state returned from the call. The state returned may also contain additional local variables used by the calling procedure, but these should should be ignored as they are out of scope as soon as the called procedure returns. The only exception is when a value is returned from a procedure call:

[1]This is a deliberate design decision that ensures that expression evaluation cannot have side effects.

the statement `return e` in pseudocode will be implemented by assigning a special *local* variable `$res` to the value of `e` and this will be passed back to the caller as part of the returned state. If the first argument of the corresponding `Call` constructor is a non-empty string, `"x"`, say, then the binding for `"x"` in the calling procedure's state must be updated with the value of the special `$res` variable when the procedure returns. If the variable is `""` then no update is required. Note that for a program to be well formed there *must* be a binding for `$res` in the returned state if the caller is expecting one; that is, for each call of the form `Call x p es` where `x` is non-empty the procedure `p` will always exit with a `Return` statement.

3.2 Examples

The following procedure, expressed in pseudocode notation, sums two given integers and assigns the result to a variable, `gSum`.

```
gAdd(x, y) {
  gSum = x + y;
}
```

If `gSum` is in scope as a global variable in the state passed to `gAdd` then the effect will be to change its value as a result of the call; if not then the assignment will simply construct a new local variable with the same name. The internal representation of `gAdd` looks like this:

```
gAdd :: ProcDef
gAdd
  = ("gAdd",
      (["x", "y"], [Assign "gSum" (OpApp Add (Var "x") (Var "y"))])
     )
```

Note that `gAdd` does not return a value, so a call to it will be represented by a statement of the form `Call "" "gAdd" [ex, ey]` where `ex` and `ey` are the argument expressions whose values will be bound to `x` and `y` respectively in the state passed to `gAdd`.

A second example is a procedural version of the `sumA` function above that uses a while loop to accumulate the array sum, as shown by the following pseudocode:

```
sumA'(a, n) {
  s = 0;
  i = 0;
  limit = n + 1;
  while (i < limit) {
    s = s + a[i]
    i = i + 1;
  }
  return s;
}
```

Note that all the variables here are local. The internal representation looks like this:

```
sumA' :: ProcDef
sumA'
  = ("sumA'",
     (["a", "n"], [Assign "s" (Const (I 0)),
                   Assign "i" (Const (I 0)),
                   Assign "limit" (OpApp Add (Var "n") (Const (I 1))),
                   While (OpApp Less (Var "i") (Var "limit"))
                         [Assign "s" (OpApp Add (Var "s")
                                                (OpApp Index (Var "a") (Var "i"))),
                          Assign "i" (OpApp Add (Var "i") (Const (I 1)))
                         ],
                   Return (Var "s")]
     )
    )
```

You are now in a position to implement Part III. You should complete this before reading on and/or attempting Part IV.

4 Implementing Memoisation

In this exercise memoisation will be applied only to pure functions and each such function will be replaced by *two* mutually-recursive procedures. The first will manage updates to the global memo table for the function and the second will be the memoised *procedural* version of the function that invokes the table manager in place of recursive function calls. For example, the two procedures required to implement the `fib` function above might be written in pseudocode as follows[2]

```
fibTableManager(n) {
  if (fibPres[n] == 0) {
    x := fibM(n);
    fibPres[n] = 1;
    fibTab[n] = x;
  }
  return fibTab[n];
}
```

```
fibM(n) {
  if (n < 3) {
    fn = 1
  } else {
    fn1 := fibTableManager(n - 1);
    fn2 := fibTableManager(n - 2);
    fn = fn1 + fn2;
  }
  return fn
}
```

Notes:

1. The code shown is not entirely optimal – we could plant a return in both arms of the 'if' statement on the right and avoid the need for the variable `fn`, for example. However, it is the sort of code that might be generated automatically by a translator. Indeed, this is the subject of Part IV.

2. The variables `fibPres` and `fibTab` here are both global arrays that are assumed to have been initialised at the top-most level.

[2]The expression `fibPres[n] == 0` is actually asking whether `fibPres[n]` is false. An expression like `not(fibPres[n])` might therefore be preferable, but the language does not support unary operations as defined.

3. In order to simplify the exercise all memoisable functions are assumed to take just a single argument which enables the corresponding memo table to be implemented as a one-dimensional array. In general, this will be not be the case, of course.

4.1 Generating the table manager

The table manager procedure for a function `f` can be generated entirely mechanically by instantiating a skeleton procedure that can be written in pseudocode in the form:

```
tableManager(n) {
  if (isPresent[n] == 0) {
    x = f(n);
    isPresent[n] = 1;
    table[n] = x;
  }
  return table[n];
}
```

with specific values of `tableManager`, `n`, `f`, `isPresent` and `table`. The following Haskell function `memoise`, defined in the template, implements exactly this instantiation:

```
memoise :: Id -> Id -> Id -> Id -> Id -> ProcDef
memoise p a f pt mt
  = (p,
     ([a], [If (OpApp Equal (OpApp Index (Var pt) (Var a)) (Const (I 0)))
                 [Call "x" f [Var a],
                  AssignA pt (Var a) (Const (I 1)),
                  AssignA mt (Var a) (Var "x")
                 ]
                 [],
            Return (OpApp Index (Var mt) (Var a))
           ]
     )
    )
```

For example, the call `memoise "fibTableManager" "n" "fibM" "fibPres" "fibTab"` will generate the representation of `fibTableManager` shown in pseudocode above, i.e.:

```
("fibTableManager",
 (["n"], [If (OpApp Equal (OpApp Index (Var "fibPres") (Var "n")) (Const (I 0)))
             [Call "x" "fibM" [Var "n"],
              AssignA "fibPres" (Var "n") (Const (I 1)),
              AssignA "fibTab" (Var "n") (Var "x")
             ]
             [],
          Return (OpApp Index (Var "fibTab") (Var "n"))
         ]
 )
)
```

4.2 Generating the memoised procedure

The memoised *procedural* implementation of a pure function is much harder to generate and is the subject of Part IV of this exercise. In the case of `fib` the result of the translation should be a representation of the `fibM` procedure above, i.e. something like this:

```
("fibM", (["n"],
          [If (OpApp Less (Var "n") (Const (I 3)))
              [Assign "$3" (Const (I 1))]
              [Call "$1" "fibTableManager" [OpApp Add (Var "n") (Const (I (-1)))],
               Call "$2" "fibTableManager" [OpApp Add (Var "n") (Const (I (-2)))],
               Assign "$3" (OpApp Add (Var "$1") (Var "$2"))
              ],
           Return (Var "$3")
          ]
         )
)
```

Because this is generated automatically from the internal representation of the `fib` function the variable names have to be chosen so as not to clash with any user-defined variables; hence the identifiers `$1`, `$2` and `$3`, which do *not* begin with a letter. Notice how the recursive *function* calls have been replaced by *procedure* calls and that the conditional expression in `fib` (constructor `Cond`) has been replaced by an 'if' statement (constructor `If`).

5 What to do

There are four parts to this test and most of the marks are for Parts I–III. Part IV is worth only two of the 30 marks available and is hard, so you are advised to attempt it only when you have completed Parts I–III.

The example functions above are all defined in the template for testing purposes. There are also some pre-defined utility functions that will prove useful throughout:

```
lookUp :: (Eq a, Show a) => a -> [(a, b)] -> b
lookUp x t
  = fromMaybe (error ("\nAttempt to lookUp " ++ show x ++
                      " in a table that only has the bindings: " ++
                      show (map fst t)))
              (lookup x t)

-- Turns an Int into an Exp...
intToExp :: Int -> Exp
intToExp n
  = Const (I n)

-- Turns a [Int] into an Exp...
listToExp :: [Int] -> Exp
listToExp
  = Const . listToVal

-- Turns a [Int] into a Value...
```

```
listToVal :: [Int] -> Value
listToVal xs
  = A (zip [0..] xs)
```

Note that the `lookUp` function provided prints a helpful error message if you mess up (you *will* mess up!). Note also that if a state contains two bindings for the same variable then the *leftmost* binding, i.e. the most recent, will be returned by `lookUp`.

There are also a number of predefined states, arrays and expressions that you can use for testing purposes, including the following sample `State` that we shall refer to shortly:

```
sampleState
  = [("x", (Local, I 5)), ("y", (Global, I 2)), ("a", (Global, listToVal [4,2,7]))]
```

You may assume throughout that all programs are well formed in the following sense:

1. All operators, functions and procedures will always be applied to the correct number of arguments, all of which will be of the appropriate type.
2. Boolean-valued expressions will always evaluate to either 0 (false) or 1 (true).
3. In an array assignment of the form `a[i] = e` the array `a` will always be in scope.
4. In a procedure call of the form `x := p(...)` the procedure `p` will always exit via a `Return` statement.
5. A `return` statement will always be the last statement to be executed in a procedure's defining code block (there is no 'dead code').

5.1 Part I: Basic utilities

1. Define a function `getValue :: Id -> State -> Value` that uses the predefined `lookUp` function above to look up the `Value` of a given variable in a given state. The scope of that variable should be ignored. For example, `getValue "x" sampleState` should return `I 5`. A precondition is that there is a binding for the variable identifier in the state.

   [1 Mark]

2. Define two functions, and `getLocals, getGlobals :: State -> State` that will return the list of local and global variables in a given state respectively, in the order in which they appear in that state. For example, `getLocals sampleState` should return `[("x",(Local,I 5))]` and `getGlobals sampleState` should return `[("y",(Global,I 2)),("a",(Global,A [(0,4),(1,2),(2,7)]))]`.

   [2 Marks]

3. Define a function `assignArray :: Value -> Value -> Value -> Value` such that the expression `assignArray a i v` returns an array that is the same as `a` except for its `i`th element, which should be bound to `v` in the array's mapping list. For example, `assignArray (getValue "a" sampleState) (I 2) (I 1)` should return, for example, `A [(2,1), (0,4), (1,2)]`; note that the order of the elements in the array's mapping list is unimportant, but no index should appear more than once.

   [2 Marks]

4. Define a function `updateVar :: (Id, Value) -> State -> State` that will update the value of a variable in a given state *whilst preserving its scope.* If the variable does not have a binding in the state then a new one should be added in which the variable has `Local` scope. For example,

   ```
   *Main> sampleState
   [("x",(Local,I 5)),("y",(Global,I 2)),("a",(Global,A [(0,4),(1,2),(2,7)]))]
   *Main> updateVar ("x", I 6) sampleState
   [("x",(Local,I 6)),("y",(Global,I 2)),("a",(Global,A [(0,4),(1,2),(2,7)]))]
   *Main>  updateVar ("z", I 3) [("g", (Global, I 8))]
   [("g",(Global,I 8)),("z",(Local,I 3))]
   ```

   The order of the elements within the state is unimportant, unless the variable being updated has two or more bindings in the given state. In that case the new (correct) binding must appear to the left of any others that remain after the update.

   [3 Marks]

5.2 Part II: Expression evaluation

1. Define a function `applyOp :: Op -> Value -> Value -> Value` that will apply the given operator to the given argument values. For the case of `Index` (array index) the result should be zero (representation `I 0`) if there is no binding for the given index in the given array. For example,

   ```
   *Main> applyOp Add (I 6) (I (-2))
   I 4
   *Main> applyOp Mul (I 3) (I 4)
   I 12
   *Main> applyOp Less (I 7) (I 0)
   I 0
   *Main> applyOp Equal (I 2) (I 2)
   I 1
   *Main> applyOp Index (A [(1,1),(0,3)]) (I 0)
   I 3
   *Main> applyOp Index (A [(1,1),(0,3)]) (I 2)
   I 0
   ```

   [3 Marks]

2. Define a function `bindArgs :: [Id] -> [Value] -> State` that, given a list of variable identifiers and corresponding values will generate a list of bindings, i.e. a `State`, in which each variable is declared as being `Local` and bound to its corresponding value. For example,

   ```
   *Main> bindArgs ["x", "a"] [I 6, A [(1,1),(0,3)]]
   [("x",(Local,I 6)),("a",(Local,A [(1,1),(0,3)]))]
   ```

   A precondition is that the two lists have the same length.

   [1 Mark]

3. Define two mutually recursive functions: `eval :: Exp -> [FunDef] -> State -> Value` and `evalArgs :: [Exp] -> [FunDef] -> State -> [Value]` for evaluating expressions. `eval` evaluates an expression in the context of a given list of `FunDef`s and `State`, and returns the value of the expression. `evalArgs` should apply `eval` to each element of a list of expressions, giving back a list of values. The `eval` function should obey the following rules:

   - The value of a constant `(Const c)` is just `c`.
   - The value of a variable is obtained from the given state (use `getValue`).
   - The value of a conditional will be the value of either the 'then' or 'else' subexpression, depending on the value of the predicate, which can be assumed always to evaluate to one of the integers 0 (false) or 1 (true), suitably tagged with the `I` constructor.
   - The value of an operator application is obtained by invoking `applyOp` on the evaluated arguments.
   - To evaluate the application of a function `f` to a list of expressions `es`, say, you first need to look up `f` in the list of `FunDefs` provided; this will give you a pair, `(as, e)`, say, where `as` is the list of argument names and `e` is the right-hand side expression. Next you need to evaluate each expression in `es` to give a list of values, `vs` say, using `evalArgs`. Then you need to bind the `as` to the `vs` using `bindArgs` – this will add the `Local` scope tag accordingly. Finally, you need to evaluate `e` in a state that is the one given augmented with the bindings generated by `bindArgs`. Note that you can use `++` to add new bindings to a state, so long as the new bindings are placed to the *left* of those already in the state.

   For example,

   ```
   *Main> sampleState
   [("x",(Local,I 5)),("y",(Global,I 2)),("a",(Global,A [(0,4),(1,2),(2,7)]))]
   *Main> eval (Const (I 1)) [] sampleState
   I 1
   *Main> eval (Var "y") [] sampleState
   I 2
   *Main> eval (OpApp Add (Var "x") (Const (I 2))) [] sampleState
   I 7
   *Main> eval (Cond (Const (I 1)) (Var "x") (Const (I 9))) [] sampleState
   I 5
   *Main> eval (FunApp "fib" [Const (I 6)]) [fib] sampleState
   I 8
   ```

   The test expressions here are defined in the template as `e1`, ..., `e5`.

   [8 Marks]

5.3 Part III: Procedure execution

Define two mutually recursive functions: `executeStatement :: Statement -> [FunDef] -> [ProcDef] -> State -> State` and `executeBlock :: Block -> [FunDef] -> [ProcDef] -> State -> State` that will respectively execute a single statement and a block (list of statements). The effect of the execution is to transform the given input state into a final state.

The parameters of these functions are similar to those for expression evaluation above. However, to implement procedure calls a list of `ProcDef`s is also required. Note that the list of function definitions (`FunDef`s) is required solely for expression evaluation, via `eval`.

5.3.1 Notes

- When executing an assignment statement (`Assign`), if the variable being assigned to does not have a binding in the current state then a new binding must be added (Section 3). Using `updateVar` defined earlier will ensure that this happens automatically.
- An assignment to an individual array element (`AssignA`) assumes that the array being updated is in scope, i.e. that the array identifier has a binding in the given state. Use `updateVar` and `assignArray` to implement this.
- When implementing procedure calls you need to be particularly careful to pass the correct state into the called procedure and to return the correct state at the end; you may wish to re-read Section 3.1 before developing your solution. Use the functions `getLocals` and `getGlobals` to implement the rule for `Call`.

5.3.2 Testing

In order to test your implementation you can use some of the examples included in the template. For example, if `sampleArray` is defined to be the expression `listToExp [9,5,7,1]` then:

```
*Main> sampleArray
Const (A [(0,9),(1,5),(2,7),(3,1)])
*Main> executeBlock [Call "s" "sumA'" [sampleArray, intToExp 3]][] [sumA'] []
[("s",(Local,I 22))]
```

which binds the *local* variable `s` to the sum of the elements of the sample array, i.e. 22. To save you typing the template includes some test functions that will execute example blocks of code like the one above, e.g.:

```
*Main> execSumA' [9,5,7,1] 3
[("s",(Local,I 22))]
```

Note that if the top-level state is initialised to contain `s` as a *global* variable when we make the call then we get a slightly different result:

```
*Main> executeBlock [Call "s" "sumA'" [sampleArray, intToExp 3]]
       [] [sumA'] [("s", (Global, I 0))]
[("s",(Global,I 22))]
*Main> execGlobalSumA' [9,5,7,1] 3
[("s",(Global,I 22))]
```

You are also now in a position to explore the performance benefits of memoisation when calculating fibonacci numbers. Start by trying to calculate `fib 35` using the naive recursive version defined in the template:

```
*Main> executeBlock [Return (FunApp "fib" [intToExp 35])] [fib] [] []
[("$res",(Local,
```

If you get bored waiting, hit `CTRL-C`! Now try the memoised version, which consists of the two mutually-recursive procedures described in Section 4 (`fibState` is primed with empty global arrays named `"fibPres"` and `"fibTab"`):

```
*Main> fibState
[("fibPres",(Global,A [])),("fibTab",(Global,A []))]
*Main> executeBlock [Call "f" "fibM" [intToExp 35]] [] [fibM, fibTableManager] fibState
[("fibPres",(Global,A [(34,1),(33,1),(32,1),(31,1),(30,1),(29,1),(28,1),(27,1),
(26,1),(25,1),(24,1), (23,1),(22,1),(21,1),(20,1),(19,1),(18,1),(17,1),(16,1),
(15,1),(14,1),(13,1), (12,1),(11,1),(10,1),(9,1),(8,1),(7,1),(6,1),(5,1),(4,1),
(3,1),(1,1),(2,1)])), ("fibTab",(Global,A [(34,5702887),(33,3524578),
(32,2178309), (31,1346269), (30,832040),(29,514229),(28,317811),(27,196418),
(26,121393),(25,75025),(24,46368), (23,28657),(22,17711),(21,10946),(20,6765),
(19,4181),(18,2584),(17,1597),(16,987), (15,610),(14,377),(13,233),(12,144),
(11,89),(10,55),(9,34),(8,21),(7,13),(6,8),(5,5),(4,3),(3,2),(1,1),(2,1)])),
("f",(Local,I 9227465))]
```

(To save you typing the test function `execFibM 35` does the same thing.) Wow! Instant! The resulting state includes the two global tables which have been modified during the execution: the `fibPres` table, where all the entries are 1, corresponding to true, and the memo table `fibTab`, whose $\texttt{n}^{th}$ element is `fib n`. The result we want (the 35^{th}) has been bound to `f` as the result of the call, i.e. 9227465.

[8 Marks]

5.4 Part IV: Memoisation

The final part of this exercise invites you to translate a pure function into a memoised procedure. For example, in the case of the `fib` function the translator should return the representation of a procedure that is equivalent to `fibM` in Section 4 above. Several things have to happen in this case: 1. The function name needs to change to a procedure name (`fib` → `fibM`); 2. The recursive calls to `fib` in the right-hand side need to be replaced with calls to the procedure that manages the memo table (here called `fibTableManager`); 3. The conditional *expression* needs to be replaced with an 'if' *statement* which involves turning the 'then' and 'else' expressions into statement blocks; 4. An explicit `Return` needs to be added.

The top-level function to do the translation has been defined in the template thus:

```
translate :: FunDef -> Id -> [(Id, Id)] -> ProcDef
translate (name, (as, e)) newName nameMap
  = (newName, (as, b ++ [Return e']))
  where
    (b, e', ids') = translate' e nameMap ['$' : show n | n <- [1..]]
```

Note that this implements 4. above in that it adds the `Return` statement. Your job is to define the helper function: `translate' :: Exp -> [(Id, Id)] -> [Id] -> (Block, Exp, [Id])` to do the rest. The second argument is a mapping table that identifies the names of the functions that have been memoised (there may be more than one) and the corresponding procedures that need to be called in their place. In the example, a call to the function `fib` needs to be replaced with a call to the procedure `fibTableManager` so the table will comprise the single pair `[("fib", "fibTableManager")]`. The third argument is a list of fresh variable identifiers, the infinite list `[$1, $2, ...]`, which will be required if the translator needs to generate new local variable identifiers, e.g. to bind the results of procedure calls. The result tuple comprises: 1. The block of code that should replace the expression – if the expression contains no memoised function calls this should be `[]`; 2. An expression that, when evaluated in conjunction with the returned code block, will return the value of the original expression; 3. The list of unused fresh variable identifiers (a suffix of the original list).

Hint 1: You may need to extend the block of code you are accumulating (`Block`) with either: i. a procedure call, i.e. when replacing a memoised function call, ii. an 'if' *statement* – you'll need this if at least one of the predicate, 'then' or 'else' expressions have been 'proceduralised' into a non-empty block, iii. an assignment statement, i.e. to ensure that the result from each arm of an 'if' statement is assigned to the same variable.

Hint 2: In some sense the 'clever' bit here is the returned expression. If the expression you're given contains *no* memoised function calls then the body returned should be `[]` and the returned expression should be exactly the one we started with. You'll see that the top-level `translate` function simply generates a `ProcDef` that returns the expression's value in that case. What expression should you return in general?

To test your solution, try running `translate` on the various test functions in the template, e.g. `fib`, `testFun` etc. In particular, the constant `fibMGenerator` in the template, which is defined in terms of `translate`, should generate a procedure that is equivalent to `fibM` in Section 4 and also defined in the template. Depending on how you define your translation function, the fresh variable names inserted by your translator may be different to those shown in `fibM`; that's absolutely fine provided the generated procedure does the right thing. If you happen to choose exactly the same variable names then you might be able to establish that if `fibMGenerator = translate fib "fibM" [("fib", "fibTableManager")]` then:

```
*Main> fibMGenerator == fibM
True
```

Good luck!

[2 Marks]