2018 Haskell January Test
Constant Propagation

This test comprises three parts (with one optional additional part) and the maximum mark is 30. The **2018 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 should therefore define your own tests to complement the ones provided.

1 Introduction

A *compiler* takes the source code of a program, e.g. written in Java, C, Haskell etc. and translates it into some form of low level "machine code" that can be executed directly by the hardware of the target platform, e.g. a desktop computer.

As part of the compilation process the program is typically subjected to a number of automatic optimisations, each of which is designed to reduce the compiled program's execution time. When applying these optimisations considerable care is needed to ensure that the optimised program computes the same answer as the original. In order to simplify the analysis needed to ensure this, most modern compilers first translate the program into an intermediate representation known as *Static Single Assignment* (SSA) and then apply the optimisations to the program's SSA representation.

In this exercise you're going to implement two commonly-applied compiler optimisations, *constant propagation* and *constant folding*, to the internal SSA representation of imperative functions, which are akin to Java methods with `int` return type.

2 SSA and constant propagation/folding

(You may wish to skip some of the details in this section on first reading and return to it later when you have read Section 3 and completed Part I.)

Static Single Assignment (SSA) is a way of representing a program where each variable is assigned exactly *once*, hence the term "single assignment", and where operators like `+`, `*` etc. are applied to constants and variables only, i.e. there are no nested operator applications. Variables must be defined, i.e. assigned to, before they can be referenced, but once defined they can be referred to any number of times. The word "static" simply relates to the fact that SSA is a way of representing (static) program text.

To illustrate the idea, consider the following function, written in an imaginary source language for defining functions:

```
1:  basicBlock() {
2:    x = 1
3:    y = 2
4:    x = x + y
5:    y = x * 3
6:    $return = y
7:  }
```

We'll refer to the notation as "pseudocode", as it's not really the source code of a proper language. However, functions written in pseudocode look a bit like Java methods with an implicit `int` return type, but without the semicolons. Integers are the only type supported so the pseudocode has no type annotations.

The value to be returned by the function is explicitly assigned to the special variable `$return`, i.e. there is no `return` statement, as in Java. You can think of `$return` as being a special global variable that is visible to all functions. In general, functions can take an arbitrary number of arguments, but this one has none, and so always returns the same value.

A function, `showFun`, has been provided for you in the template which will display the representation of a given function in the above pseudocode form. For example, you can output the above by typing `showFun basicBlock` at the ghci prompt.

2.1 Single assignment

Note that the variables `x` and `y` in this example are re-assigned having been initialised to their respective values (`1` and `2`). In an equivalent SSA representation each assignment must be to a *unique* identifier. In this example that's easy: just invent multiple numbered versions of each identifier, e.g. `x0`, `x1`, `y0`, `y1` and use these instead (`showFun basicBlockSSA`):

```
1:  basicBlock() {
2:    x0 = 1
3:    y0 = 2
4:    x1 = x0 + y0
5:    y1 = x1 * 3
6:    $return = y1
7:  }
```

You can now see why the single-assignment property is important: if a variable is assigned to a constant, e.g. `x0 = 1`, then every time we see the variable elsewhere in the body of the function we can safely replace it with its value and the original assignment can be deleted. This is because the same variable can never be re-assigned – a very familiar concept to Haskell programmers! This replacement process is called *constant propagation*. By propagating the constants `x0 = 1` and `y0 = 2` it is easy to see that we can rewrite the function thus:

```
1:  basicBlock() {
2:    x1 = 1 + 2
3:    y1 = x1 * 3
4:    $return = y1
5:  }
```

Furthermore, if we now spot the fact that `1 + 2 = 3` – this is called *constant folding* – then we can establish that `x1 = 3` and hence, by further propagation, that `y1 = 3 * 3`. We can now use constant folding again to rewrite this assignment to `y1 = 9` and this can then be propagated to the `return` statement. The resulting function body ends up as a one-liner (`showFun basicBlockOptimised`):

```
1:  basicBlock() {
2:    $return = 9
3:  }
```

2.2 Phi functions: conditionals

The problem comes when a variable can be assigned to two or more *different* values, e.g. depending on the outcome of a conditional. As an example, consider

the function shown on the left below (`showFun example`)[1] and its SSA form, which is shown on the right. Note the argument (`x`).

```
1:  example(x) {
2:    a = 1
3:    b = x + 2
4:    c = 3
5:    if (x == 10) {
6:      a = 1
7:      c = 5
8:    }
9:    d = a + 3
10:   e = d + b
11:   $return = e + c
12: }
```

```
1:  example(x) {
2:    a0 = 1
3:    b0 = x + 2
4:    c0 = 3
5:    if (x == 10) {
6:      a1 = 1
7:      c1 = 5
8:    }
9:    a2 = ???
10:   c2 = ???
11:   d0 = a2 + 3
12:   e0 = d0 + b0
13:   $return = e0 + c2
14: }
```

In the SSA form, each assignment has been given a unique target variable, as required. However, there are now two choices for the values assigned to `a2` and `c2`, depending on whether or not the body of the "true" branch of the conditional (lines 6 and 7) is executed. The trick is to introduce a special construct called a *phi* function, usually written using the Greek letter ϕ, but here written verbatim as `PHI`, which indicates the possible values that the target variable can be assigned to (`showFun exampleSSA` will display the complete SSA version):

```
1:  example(x):
      :
9:    a2 = PHI(a1, a0)
10:   c2 = PHI(c1, c0)
      :
13:   $return = e0 + c2
14: }
```

For example, line 9 says that `a2` can either assume the value of `a1` or the value `a0`, depending on which path the program's execution took in order to reach that line[2]. In this exercise we won't attempt to execute phi functions, but will instead remove them once any optimisations have been applied, as would a compiler. This removal process forms Part III of the exercise.

Let's see how the optimisation might proceed with the above example. The variables `a0`, `c0`, `a1` and `c1` are all assigned constant values so we can replace them throughout with their respective values, i.e. `1`, `3`, `1` and `5`. Doing so we end up with:

[1] This is a contrived example and would score no points in a programming style contest, but don't worry about that here!

[2] In general phi functions can take any number of arguments, but in this exercise they will always have exactly two, by construction.

```
1:  example(x) {
2:    b0 = x + 2
3:    if (x == 10) {
4:    }
5:    a2 = PHI(1, 1)
6:    c2 = PHI(5, 3)
7:    d0 = a2 + 3
8:    e0 = d0 + b0
9:    $return = e0 + c2
10: }
```

Notice that the conditional is still there (this is important) but both its branches are empty. Now look at line 5: this says that, regardless of the value of the conditional, the variable `a2` will always be assigned the value `1` – you can verify this by inspecting the original non-SSA version of the function. In general, any assignment of the form `v = PHI(c, c)`, where `c` is a constant, can be replaced with `v = c` and we'll consider this to be another instance of constant folding. Applying this rule here yields `a2 = 1`, a new constant assignment, which will replace the original. As we've now uncovered a new constant assignment we repeat the propagation process by removing `a2 = 1` from the function and replacing `a2` with `1` throughout. From this we establish, by further substitution and constant folding, that `d0` is 4. Another iteration of the propagation process replaces the assignment to `e0` with `e0 = 4 + b0` and we end up with (`showFun exampleSSAPropagated`):

```
1:  example(x) {
2:    b0 = x + 2
3:    if (x == 10) {
4:    }
5:    c2 = PHI(5, 3)
6:    e0 = 4 + b0
7:    $return = e0 + c2
8:  }
```

There's nothing more we can do now, as there are no further constant assignments to propagate. All that remains is to remove the assignments involving phi functions – a process that's often called "unPhi". In this case this has the effect of inserting the assignment `c2 = 5` at the end of the "true" branch of the preceding conditional and, similarly, `c2 = 3` at the end of the "false" branch (see Section 4.3 for the rules). This results in the following, optimised, version (`showFun exampleOptimised`):

```
1:  example(x) {
2:    b0 = x + 2
3:    if (x == 10) {
4:      c2 = 5
5:    } else {
6:      c2 = 3
7:    }
8:    e0 = 4 + b0
9:    $return = e0 + c2
10: }
```

2.3 Phi functions: do-while loops

What if the program contains a do-while loop? Recall that in a do-while loop of the form `do <body> while <cond>` the `body` of the loop is executed repeatedly until the condition, `<cond>`, evaluates to false, i.e. it executes *whilst* the condition is true. Importantly, the body of a do-while loop is executed *at least once*, unlike a while loop where the body may not be executed at all, depending on the value of the condition.

Similar to a conditional, some assignments in a do-while loop will require a phi function as the variable being assigned can assume one of two possible values depending on whether the loop body is being executed for the first time, or the n^{th} time, $n > 1$. To illustrate this consider the loop on the left below (`showFun loop`) for computing $\sum_{i=0}^{n}(i+2k)(i+2k+1)$ but in the specific case where $k = 0$[3]. Note that this is not in SSA form.

```
1:  loop(n) {
2:    i = n
3:    k = 0
4:    sum = 0
5:    if (i == 0) {
6:      $return = 0
7:    } else {
8:      do {
9:        sum = sum + (i + 2 * k) *
                      (i + 2 * k + 1)
10:       i = i + -1
11:     } while (i > 0)
12:     $return = sum
13:   }
14: }
```

```
1:  loop(n) {
2:    i0 = n
3:    k0 = 0
4:    sum0 = 0
5:    if (i0 == 0) {
6:      $return = 0
7:    } else {
8:      do {
9:        sum1 = PHI(sum0, sum2)
10:       i1 = PHI(i0, i2)
11:       k1 = k0 * 2
12:       a0 = i1 + k1
13:       k2 = k0 * 2
14:       b0 = k2 + 1
15:       b1 = i1 + b0
16:       m0 = a0 * b1
17:       sum2 = sum1 + m0
18:       i2 = i1 + -1
19:     } while (i2 > 0)
20:     $return = sum2
21:   }
22: }
```

In the first iteration of the loop, `i` assumes the value of the argument (`n`). In subsequent iterations it assumes the value of the *updated* `i`, i.e. from the assignment `i = i + -1`. In the SSA version of the function shown on the right (`showFun loopSSA`) an assignment to a phi function has therefore been planted accordingly. A similar line of reasoning applies to `sum`.

Notice that there are several redundant expressions that can be eliminated by a combination of constant propagation and constant folding. By propagating the constant `k0 = 0` we can establish that `k1` and `k2` are both `0 * 2` which, by constant folding, gives `k1 = 0` and `k2 = 0`. From this we can estab-

[3]This code might be auto-generated from a version where `k` is also a parameter, but where the compiler has decided to build a "specialisation" of the function for a particular value of `k`, here 0.

lish that `a0 = i1 + 0 = i1`, another type of constant folding. Similarly, we also have that `b0 = 1` and therefore that `b1 = i1 + 1`. Doing all this we end up with the optimised SSA representation shown on the left below (`showFun loopSSAPropagated`).

```
1:  loop(n) {
2:    i0 = n
3:    if (i0 == 0) {
4:      $return = 0
5:    } else {
6:      do {
7:        sum1 = PHI(0, sum2)
8:        i1 = PHI(i0, i2)
9:        a0 = i1
10:       b1 = i1 + 1
11:       m0 = a0 * b1
12:       sum2 = sum1 + m0
13:       i2 = i1 + -1
14:     } while (i2 > 0)
15:     $return = sum2
16:   }
17: }
```

```
1:  loop(n) {
2:    i0 = n
3:    if (i0 == 0) {
4:      $return = 0
5:    } else {
6:      sum1 = 0
7:      i1 = i0
8:      do {
9:        a0 = i1
10:       b1 = i1 + 1
11:       m0 = a0 * b1
12:       sum2 = sum1 + m0
13:       i2 = i1 + -1
14:       sum1 = sum2
15:       i1 = i2
16:     } while (i2 > 0)
17:     $return = sum2
18:   }
19: }
```

In order to remove the phi functions here (see Section 4.3) new assignments to `sum1` and `i1` must be inserted both before the start of the loop (first loop entry) and then at the end of the loop (subsequent entries). The two phi function arguments tell you what to place on the right hand side in each case. The final version of the function is shown on the right above (`showFun loopOptimised`).

Notice that the number of multiplications (`*`) and additions (`+`) have been reduced from 3 and 5 in the original loop body to 1 and 3 respectively. However, the function isn't completely optimal, e.g. it is possible in this example to eliminate `a0`, `sum2` and `i2`, but we'll not worry about this here. Redundant variables will invariably be removed by a later phase of compilation, e.g. by *register allocation*, which is not part of this exercise.

2.4 Correctness

We can check that the original and optimised versions of a function compute the same results by executing the two versions for the same input arguments(s) and comparing their outputs. The first part of this exercise involves building an interpreter for functions that will enable you to do this.

3 Haskell representation

For the purposes of this exercise you're going to work with the internal representation of our pseudocode language via the following Haskell data types:

```
type Id = String
```

```
fact :: Function
fact
  = ("fact",
     ["n"],
     [If (Apply Eq (Var "n") (Const 0))
        [Assign "$return" (Const 1)]
        [Assign "prod" (Const 1),
         Assign "i" (Var "n"),
         DoWhile
           [Assign "prod" (Apply Mul (Var "prod") (Var "i")),
            Assign "i" (Apply Add (Var "i") (Const (-1)))
           ]
           (Apply Gtr (Var "i") (Const 0)),
         Assign "$return" (Var "prod")
        ]
     ]
    )
```

Figure 1: The representation of a function for computing factorials

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

type Block = [Statement]

data Statement = Assign Id Exp |
                 If Exp Block Block |
                 DoWhile Block Exp
               deriving (Eq, Show)

data Exp = Const Int | Var Id | Apply Op Exp Exp | Phi Exp Exp
         deriving (Eq, Show)

data Op = Add | Mul | Eq | Gtr
        deriving (Eq, Show)
```

A function is represented by a triple comprising its name (a `String`), a list of its argument identifiers (a `[String]`) and the function *body*, which is a *block* of code (`Block`) represented by a list of zero or more *statements*. A statement (`Statement`) is either the assignment of a variable to an expression (`Assign`), a conditional (`If`) with a predicate (`Exp`) and a block (`Block`) for the true and false branches respectively, or a do-while loop (`DoWhile`) comprising the loop body (a `Block`) and the termination predicate (`Exp`).

Booleans are not supported directly, so they will be encoded as integers: true will be represented by the integer 1 and false by 0. The predicate expressions in the representation of conditionals (`If`) and do-while loops (`DoWhile`) can be assumed always to evaluate to either 0 or 1.

Expressions are either integer constants (`Const`), e.g. 9, -1, 0, ..., variable identifiers (`Var`), binary operator applications (`Apply`) or phi functions (`Phi`).

In general, operator arguments may contain arbitrary nested subexpressions. However, if the function is in SSA form then operator arguments will comprise only constants and variables.

Identifiers are represented by strings and we will assume that they all begin with a lower case letter, e.g. `"x"`, `"a1"`, `"number"`, ..., except for the special function return identifier `$return`. The operators supported are addition (`Add`, equivalent to `+` in Java), multiplication (`Mul`, equivalent to `*`), equal (`Eq`, equivalent to `==`) and greater-than (`Gtr`, equivalent to `>`).

As an example, Figure 1 shows the representation of the following function, which is not in SSA form, for computing the factorial of a given integer, `n`≥ 0 (`showFun fact`):

```
1:  fact(n) {
2:    if (n == 0) {
3:      $return = 1
4:    } else {
5:      prod = 1
6:      i = n
7:      do {
8:        prod = prod * i
9:        i = i + -1
10:     } while (i > 0)
11:     $return = prod
12:   }
13: }
```

4 What to do

There are three parts to this test, plus an optional part that is not for credit. Most of the marks are allocated to Parts I and II. If you get stuck on one part you can switch to one of the others, as they can be solved independently.

The example functions referred to in this specification are all defined in the template for testing purposes and the following naming convention applies:

- `fun` – the original function with name `fun`
- `funSSA` – an SSA version of the function
- `funSSAPropagated` – the SSA version of the function after constant propagation/folding
- `funOptimised` – as above but also after removal of phi functions (`unPhi`).

As you have already seen, there is a predefined function, `showFun`, for displaying functions, which you should use for testing. Another function, `showBlock`, is also provided for displaying blocks, e.g.

```
*Main> let (_, _, body) = basicBlock in showBlock body
1:    x = 1
2:    y = 2
3:    x = x + y
4:    y = x * 3
5:    $return = y
```

In most cases no attempt has been made to format the internal representations of the various sample functions to make them more readable; you should use `showFun` and `showBlock` to display these and any representations you generate yourself.

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

1. All variable names, except for `$return`, will begin with a lower-case letter and all variables will have a value when they are referenced, i.e. there will be no uninitialised ("free") variables.
2. All functions will assign their result to the special variable `$return` before exiting.
3. Predicates will always evaluate to either 0 (false) or 1 (true).
4. There will be no nested operator applications in a function in SSA form, i.e. the arguments to all operators in SSA form will comprise only constants and variables. This restriction *only* applies to functions in SSA form.

Finally, note that phi functions will occur in one of two places only in this exercise:

1. Immediately after a conditional
2. At the beginning of a do-while loop

4.1 Part I: Interpreter

The first task is to write an interpreter for executing `Functions`. Execution will be with respect to a `State`, which is a list that associates variable identifiers with their integer bindings:

```
type State = [(Id, Int)]
```

A look-up function has been defined for you in the template for returning the binding of a given item (e.g. identifier) in a given look-up table (e.g. state); it prints a helpful error message if the item isn't found:

```
lookUp :: (Eq a, Show a) => a -> [(a, b)] -> b
lookUp i table
  = fromMaybe (error ("lookup failed on identifier: " ++ show i))
              (lookup i table)
```

In addition to various test functions, a number of example states and expressions are also provided for testing purposes – see the examples below.

1. Define a function `update :: (Id, Int) -> State -> State` that will update a given state with a new binding for a variable. If the variable is not bound in the state its binding should be added. For example:

```
*Main> s1
[("x",7),("y",8)]
*Main> update ("x", 3) s1
[("x",3),("y",8)]
*Main> update ("z", 0) s1
[("z",0),("x",7),("y",8)]
```

The order of the elements in the resulting state is not important.

[2 Marks]

2. Define a function `apply :: Op -> Int -> Int -> Int` that applies a given operator to its two arguments (both `Int`s). For example:

```
*Main> apply Add 7 5
12
*Main> apply Gtr 4 9
0
```

[2 Marks]

3. Define a function `eval :: Exp -> State -> Int` that will evaluate a given expression with respect to a given state. There are two preconditions: the variables in the expression will all be bound in the state and there will be no phi functions (`Phi`), as they are not executable. For example,

```
*Main> s1
[("x",7),("y",8)]
*Main> e1
Var "x"
*Main> e2
Apply Mul (Apply Add (Var "x") (Const 1)) (Var "y")
*Main> eval e1 s1
7
*Main> eval e2 s1
64
```

[3 Marks]

4. Define two mutually recursive functions: `execStatement :: Statement -> State -> State` and `execBlock :: Block -> State -> State` that will execute statements and blocks respectively. Note that they must be mutually recursive because `If` and `DoWhile` statements themselves contain blocks. The result should be the state after execution of the statement/block. The decision as to whether to execute the first or second branch of a conditional (`If`) will be determined by the value of the corresponding predicate expression: 1 for the first and 0 for the second, corresponding to true and false respectively. To execute a do-while loop the termination condition should be tested with respect to the state *after* each execution of the loop body – it is *not* a while loop!

   You can test these functions using `execFun`, defined for you in the template, which will execute the a given function by applying `execBlock` to

the function's body and an initial state containing the values of the functions arguments:

```
execFun :: Function -> [Int] -> State
execFun (name, args, p) vs
  = execBlock p (zip args vs)
```

For example,

```
*Main> execFun fact [5]
[("$return",120),("i",0),("prod",120),("n",5)]

*Main> execFun loop [4]
[("$return",40),("i",0),("sum",40),("k",0),("n",4)]

*Main> execFun loopOptimised [4]
[("$return",40),("i1",0),("sum1",40),("i2",0),("sum2",40),
("m0",2),("b1",2),("a0",1),("i0",4),("n",4)]
```

Again, the order of the elements in the resulting state is unimportant.

For the `loop` function note that when $k = 0$, $\sum_{i=0}^{n}(i+2k)(i+2k+1) = n(n+1)(n+2)/3$, which is 40 when $n = 4$.

[7 Marks]

4.2 Part II: Constant propagation and folding

For each of the functions in this section there is a precondition that all blocks and/or expressions are in SSA form, so operator arguments will comprise only constants and variables.

1. Define a function `foldConst :: Exp -> Exp` that will simplify expressions in SSA form using (only[4]) the following rules:

   - If `c` is a constant then `PHI(c, c)` can be simplified to `c`.
   - If `c1` and `c2` are constants then `c1 <op> c2` can be replaced by applying the operator `<op>` to `c1` and `c2`.
   - If `v` is a variable then `v + 0` and `0 + v` can be replaced by `v`.

   For example,

   ```
   *Main> e3
   Phi (Const 2) (Const 2)
   *Main> e4
   Apply Add (Const 0) (Var "x")
   *Main> foldConst e3
   Const 2
   *Main> foldConst e4
   Var "x"
   ```

   [3 Marks]

[4] You can probably think of other folding optimisations, e.g. `1 * x = x` but you should only encode the rules given, in case you mess up the autotesting.

2. Define a function `sub :: Id -> Int -> Exp -> Exp` that will substitute a given identifier with a given integer in a given expression and then apply the `foldConst` function above to the result. For example,

```
*Main> e5
Apply Add (Var "a") (Var "x")
*Main> sub "a" 0 e5
Var "x"
```

[3 Marks]

3. Define a function `propagateConstants :: Block -> Block` that will apply constant propagation and constant folding to a given code block in SSA form. There are several ways of doing this and here are two you might consider:

   (a) **Worklist Algorithm**: The idea is to maintain a *worklist*, which is a list of constant assignments extracted from a block of code, each in the form of an `Assign` statement or (better?) an `(Id, Int)` pair. Starting with the original code block, build the initial worklist, i.e. the list of constant assignments in the block, removing those assignments from the block in the process (see below). In each iteration of the algorithm, remove one of the items, `(v, c)`, say, from the worklist – it doesn't matter which, but presumably the item at the head is a good choice! Now *scan* the modified block replacing `v` with `c` and applying constant folding where possible, using the `sub` function above. Clearly you need to locate every expression (`Exp`) in the block in order to do this.

   There is one special case: given an assignment to the return variable, `$return = e`, you should apply substitutions to `e`, but you must *not* remove the assignment if `e` ends up as a constant; recall that functions must always return a value.

   The result of each scan should be a (possibly) modified block *and* a new worklist comprising any new constant assignments that were generated as a result of the scan. For example, substituting `x` for `6` in `v = x + 1` will result in a new constant assignment `v = 7`, so `(v, 7)` will need to form part of the worklist returned and the assignment `v = 7` will need to be removed from the block. At the end of the scan, the new worklist is then appended (`++`) to the current worklist and the process continues with the modified block. Eventually, the worklist will become empty (`[]`) whereupon the result is the current (modified) version of the block. This suggests a helper function with a type something like the following:

```
type Worklist = [(Id, Int)]

scan :: Id -> Int -> Block -> (Worklist, Block)
```

   How should you build the initial worklist? Well, you could perform a "dummy" scan such as `scan "$INVALID" 0 block` where `$INVALID` is an identifier that is known not to appear in the block. This will return the list of constant assignments and a modified version of the

block with those assignments removed, but no substitutions will be performed – exactly what you want to get started.

For example,

```
*Main> let (_, _, b) = basicBlockSSA

*Main> showBlock b
1:    x0 = 1
2:    y0 = 2
3:    x1 = x0 + y0
4:    y1 = x1 * 3
5:    $return = y1

*Main> let (worklist, modifiedBlock) = scan "$INVALID" 0 b

*Main> worklist
[("x0",1),("y0",2)]

*Main> showBlock modifiedBlock
1:    x1 = x0 + y0
2:    y1 = x1 * 3
3:    $return = y1
```

(b) **Substitution algorithm**: The idea is similar to the worklist algorithm above except that the "worklist" now comprises a single *function* that applies any number of substitutions (`sub`) to a given expression *in a single scan*. This suggests the following type for the scan function:

```
scan :: (Exp -> Exp) -> Block -> (Exp -> Exp, Block)
```

Clearly, to add a new substitution to the worklist you'll need to compose (`.`) substitution functions. For example, if `s1` and `s2` are substitution functions of type `Exp -> Exp` then `s2 . s1` (likewise `s1 . s2`) is a function, also of of type `Exp -> Exp`, that will apply both substitutions simultaneously.

How should you build the initial worklist now? Well, how about applying the "identity" substitution to the original block, viz. `scan id block`? How do you know when to terminate the process, as you no longer have a list of work items that you can inspect? One way is to keep going until the "modified" block you get back from `scan` is the same as the one you passed in. Another way is to arrange for `scan` also to return a boolean telling you whether or not any expressions were modified during the scan. Maybe you can think of another.

A function called `applyPropagate` is provided in the template for testing the output of your `propagateConstants` function. For example,

```
*Main> showFun (applyPropagate exampleSSA)
1:  example(x) {
2:    b0 = x + 2
3:    if (x == 10) {
```

```
4:    }
5:    c2 = PHI(5, 3)
6:    e0 = 4 + b0
7:    $return = e0 + c2
8:  }

*Main> applyPropagate loopSSA == loopSSAPropagated
True
```

[7 Marks]

4.3 Part III: UnPhi

Because we're working with a fairly high level representation of functions, the rules for removing phi functions are straightforward:

- If a conditional (`If`) statement is followed by one or more phi function assignments then, for each assignment of the form `v = PHI (e1, e2)` add the statement `v = e1` to the end of the first (true) block of the conditional and `v = e2` to the end of the second (false) block; the phi assignment is deleted.
- If the body of a do-while (`DoWhile`) loop begins with one or more phi function assignments then, for each assignment of the form `v = PHI(e1, e2)` add the statement `v = e1` before the start of the do-while loop and `v = e2` to the end of the body of the do-while loop; again, the phi assignment is deleted.

Thus, define a function `unPhi :: Block -> Block` that will remove the phi assignments from any function in SSA form. A function `applyUnPhi` has been defined in the template for testing purposes. Another function, `optimise`, will test the combination of both `propagateConstants` and `unPhi`. However, you can also use your interpreter. For example,

```
*Main> showFun max2SSAPropagated
1:  max2(x, y) {
2:    if (x > y) {
3:      m0 = x
4:    } else {
5:      m1 = y
6:    }
7:    m2 = PHI(m0, m1)
8:    $return = m2
9:  }

*Main> showFun (applyUnPhi max2SSAPropagated)
1:  max2(x, y) {
2:    if (x > y) {
3:      m0 = x
4:      m2 = m0
5:    } else {
```

```
6:      m1 = y
7:      m2 = m1
8:    }
9:    $return = m2
10: }

*Main> applyUnPhi (loopSSAPropagated) == loopOptimised
True

*Main> optimise loopSSA == loopOptimised
True

*Main> execFun loop [4]
[("$return",40),("i",0),("sum",40),("k",0),("n",4)]

*Main> execFun (optimise loopSSA) [4]
[("$return",40),("i1",0),("sum1",40),("i2",0),("sum2",40),
("m0",2),("b1",2),("a0",1),("i0",4),("n",4)]
```

[3 Marks]

4.4 Part IV: Generating SSA form

If you've finished Parts I-III you might now like to attempt the missing part of the story. Thus, define a function `makeSSA :: Function -> Function` that will convert a given function to SSA form. If there are no conditionals or do-while loops this is actually a straightforward process of making all variables single assignment and breaking down expressions involving nested operator applications into multiple assignments. In general, however, you need to introduce phi functions. What are the rules?

There is no credit for attempting this part, but it might be used as a tie-breaker for the Haskell Prize. Please include a comment in your submission outlining your strategy.

Good luck!