2021 Haskell January Test
Graph colouring and register allocation

This test comprises five parts and the maximum mark is 30.

The **2021 Haskell Programming Prize** will be awarded for the best overall solution, or solutions.

Part V carries no marks, so will only be used as a tie-breaker when determining the prize winner(s).

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 are therefore advised to define your own tests to complement the ones provided.

1 Introduction

Graph colouring involves assigning one of C given colours to each node of a graph so that no two nodes that are connected have the same colour. Graph colouring has many applications in scheduling, timetabling and program optimisation. In this exercise we will be concerned with the problem of *register allocation* which here means renaming some or all of the variables in a program so that the new names correspond to the registers of the target machine. This can substantially improve the performance of the compiled program because register accesses are much faster than normal variable accesses, which typically involve referencing main memory. Register allocation is an important topic in optimising compilers, but we'll consider here just a part of the problem.

2 Graph colouring

Figure 1 (left) shows an example of a graph with four nodes that has been successfully coloured using two colours.

In general it may not be possible to colour a graph with the given number of colours. Furthermore, a particular colouring algorithm may not be able to find a colouring when, in theory, one exists. In these cases one or more of the nodes may end up being left out, which is equivalent to assigning "blank" colours to those nodes. To illustrate this, Figure 1 (middle) shows a graph that cannot be coloured using just two colours and you can see that the colouring shown has left out node 1 by assigning it a blank colour, which is shown in white. It is possible to colour this graph using three colours, however, as shown in Figure 1 (right).

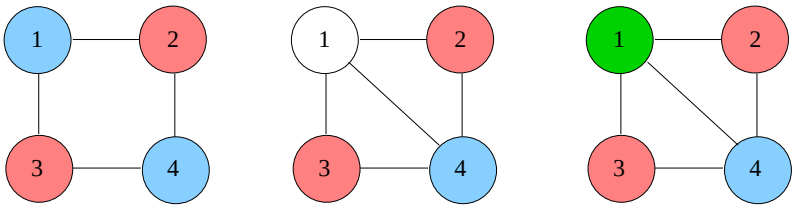


Figure 1: Graph colourings

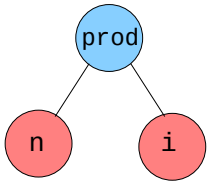
3 Register allocation

Compilers perform register allocation on the low-level representation of a program, but in this exercise we'll apply the same ideas to the source program instead. Of course, a lot of important details will have to be omitted to make the problem tractable, but many of the key ideas still apply.

3.1 Interference graphs

There are many algorithms for performing register allocation. The one we will be using here is based on colouring an *Interference Graph* (IG), which is a graph whose nodes represent the variables in a program and where an edge between nodes `v` and `v'` means that there is at least one point in a program where the variables `v` and `v'` are both "live". A variable is live at a given point if its value may be needed later on in the execution of the program. Variables that are live

```
   fun fact(n) {
0:   if (n == 0) {
1:     return = 1;
     } else {
2:     prod = 1;
3:     i = n;
4:     while (i >= 1) {
5:       prod = prod * i;
6:       i = i + -1;
       }
7:     return = prod;
     }
   }
```




```
   fun fact(n) {
0:   R2 = n;
1:   if (R2 == 0) {
2:     return = 1;
     } else {
3:     R1 = 1;
4:     while (R2 >= 1) {
5:       R1 = R1 * R2;
6:       R2 = R2 + -1;
       }
7:     return = R1;
     }
   }
```

Figure 2: Factorial function

simultaneously at some point in a program are said to *interfere*, hence the name "interference graph". Two variables that interfere cannot be renamed to use the same register.

Don't worry if the concepts of live variables and interference are unfamiliar to you - you don't need to understand the details in order to solve the problem.

3.2 Register allocation via graph colouring

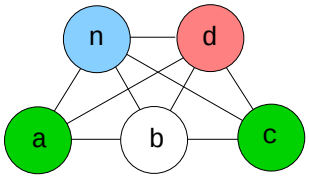
Given R available registers in the target computer, the problem of replacing some or all of the variables of a program with register references is equivalent to finding an R-colouring of the interference graph for that program. To illustrate this, Figure 2 (left) shows the source code of a function for computing the factorial of a given integer `n`. Its interference graph is shown in Figure 2 (middle), which has been successfully coloured using two colours. In this example, the colour blue corresponds to register 1 (variable `prod`) and red to register 2 (variables `n` and `i`). The graph tells us that the variables `n` and `prod` interfere, as do `prod` and `i`, but that `n` and `i` do not. Thus `n` and `i` can be renamed to use the same register, but `prod` must use a different one. Using this assignment of colours (registers) to variables we can rename the variables of the original factorial function without changing its behaviour; the transformed function is shown in Figure 2 (right) where a reference to register `i` is denoted `Ri`, `i`=1,2.

Notice that the assignment `i = n` on line 3 of the original function has been eliminated in the transformed version, as both `i` and `n` both map to register 2 (`R2`). The assignment `R2 = R2`, and in general any assignment of the form `v = v`, is clearly redundant. The ability to remove some assignment statements from a program is a nice side effect of register allocation.

Another example is shown in Figure 3 where we assume that there are three registers (colours) available. The nodes `a`, `c`, `d` and `n` in the interference graph have been coloured green (register 2), green, red (register 2) and blue (register 1) respectively, so the corresponding program variables get mapped to `R3`, `R3`, `R2` and `R1` respectively. However, node `b` could not be coloured using just the three available colours and so is left out; recall that this is equivalent to colouring the node white. Thus, the program variable `b` cannot be renamed, other than to itself, which is what will actually happen. The transformed program is shown on the right.

Remark: In practice, if a compiler cannot colour an interference graph completely using the available colours it will typically add code to the program to save register values to memory so that it can re-use a previously allocated register – this is called *register spilling*. Here we will side-step that problem altogether and say that a program variable is either mapped everywhere

```
   fun f(a, n) {
0:   b = 0;
1:   while (n >= 1) {
2:     c = a + b;
3:     d = c + b;
4:     if (d >= 20) {
5:       b = c + d;
6:       d = d + 1;
       }
7:     a = c * d;
8:     n = n + -1;
     }
9:   return = d;
   }
```




```
   fun f(a, n) {
0:   R3 = a;
1:   R1 = n;
2:   b = 0;
3:   while (R1 >= 1) {
4:     R3 = R3 + b;
5:     R2 = R3 + b;
6:     if (R2 >= 20) {
7:       b = R3 + R2;
8:       R2 = R2 + 1;
       }
9:     R3 = R3 * R2;
10:    R1 = R1 + -1;
     }
11:  return = R2;
   }
```

Figure 3: A function with an incomplete graph colouring

to the same register, or not mapped at all, equivalent to mapping it to itself, as above. This simplifies the register allocation problem considerably, but that's fine, as we're only focusing on a part of the problem.

4 Representation

The language you will be working with is similar to a stripped-down version of Kotlin with the following differences:

- The only type supported is integer (`Int`), so variables and functions have no associated type decorator: everything is an `Int`.
- Booleans are represented as integers, with `false` and `true` being represented by 0 and 1 respectively.
- The only operators supported are addition (`+`), multiplication (`*`), equals (`==`) and greater-or-equal (`>=`).
- Function results are returned by assigning to a special `return` variable, rather than by treating `return` as a keyword.

A function comprises a sequence of *statements*, which we shall refer to as a *block*. Each statement is either a variable assignment (e.g. `v = e`), a conditional of the form `if (p) {b1} else {b2}` or a while loop of the form `while (p) {b}`. The two arms of each conditional and the body of each while loop are themselves blocks and may be empty.

The right-hand side of an assignment and the predicates attached to conditionals and while loops are *expressions*, each of which is either a constant (e.g. `0`, `1`, `-9`, ...), a variable reference (e.g. `x`, `prod`, ...) or the application of an operator to two (sub)expressions (e.g. `x + y - z`, `a * 2`, `n >= 0`, ...).

4.1 Program representation

The following Haskell data types define the internal representation of functions, which should be self-explanatory:

```
type Id = String

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

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

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

type Block = [Statement]

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

The three components of a `Function` are the function name (`Id`), the argument names (`[Id]`) and the function body (`Block`). For example, the `fact` function above will be represented by:

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

It is useful to be able to display programs in "pseudo" source syntax as illustrated in the above figures. A function `showFun` is provided for this purpose which also numbers (only) the lines associated with assignments and the predicates attached to `if` and `while` statements. For example, the program on the left of Figure 2 can be produced via `showFun fact`. A similar function called `showBlock` is provided for displaying blocks (see later).

4.2 Graph representation

A graph will be represented by a pair comprising a list of node labels and a list of the edges of the graph, each of which is a pair of labels. Graphs in general are polymorphic, so labels can have any

type `a`. An interference graph (`IG`) is a specific type of graph where the nodes represent program variable names, i.e. strings (type synonym `Id` above).

Colours will be represented by *positive* integers, and the "blank" colour (white) by the integer 0. A `Colouring` of a `Graph a` is a mapping from labels (type `a`) to colours. Hence:

```
type Edge a = (a, a)

type Graph a = ([a], [Edge a])

type IG = Graph Id

type Colour = Int

type Colouring a = [(a, Colour)]
```

Notice that a graph comprising a single node has no edges. As an example, the graph of Figure 1 (middle) might be represented by `([1,2,3,4],[(1,2),(1,3),(2,4),(3,4)])` and its 2-colouring by: `[(2,2),(1,0),(3,2),(4,1)]`. Similarly, the interference graph for the `fact` function above might be represented by `(["i","n","prod"],[("i","prod"),("n","prod")])` and its 2-colouring by `[("i",2),("n",2),("prod",1)]`. The order of the elements within these various lists is unimportant.

5 Graph colouring algorithm

The problem of determining whether or not a graph can be coloured with C colours is *NP-complete*, which essentially means that you can only answer the question for sure by looking at all possible colourings – a "brute force" approach. This is computationally intractable for all but the simplest problems.

In order to make graph colouring practical it is necessary to use *heuristics*, i.e. rules that lead to efficient solutions in many cases, but that do not always guarantee to find a colouring when one exists. In this exercise we will use Chaitin's colouring algorithm which is known to work well when applied to the register allocation problem considered here.

5.1 Chaitin's algorithm

Chaitin's colouring algorithm proceeds as follows: to colour a graph g with at least one node, using colours $1, 2, ..., C$:

- Pick the node n (or any node n, if there is more than one) in g with the smallest *degree*. The degree of a node is the number of other nodes to which it is connected.
- Remove n and its associated edges from g, giving g'.
- Recursively colour g', giving a `Colouring`, $cMap$ say, i.e. a mapping from node labels to colours, for each node in g'.
- Colour n with the smallest integer colour, c say, that is different to the colours assigned to n's neighbours in g. If no colours are available assign n the "blank" colour $c = 0$.
- Return the result of adding (n, c) to $cMap$.

The base case is where the graph has no nodes, in which case the result is the empty colouring, i.e. the empty mapping `[]`.

For example, to colour the interference graph for `fact` we start with g=`(["i","n","prod"], [("i","prod"), ("n","prod")])`. The nodes `"i"` and `"n"` both have the smallest degree, 1, so we can choose either to remove first. Given a choice we'll arbitrarily pick the one with the smallest identifier, assuming the usual dictionary ordering on strings, so we pick `"i"` first, because `"i"<"n"`. Removing `"i"` from g we get g'=`(["n","prod"], [("n","prod")])`. Colouring g' recursively yields `[("n",2),("prod",1)]` – the 1 and 2 here correspond to colours. Node `"i"` has only one neighbour in g, i.e. `prod`, so we pick the numerically smallest colour that is not the same as the one assigned to `prod`, i.e. 2. Hence, the final colouring is `[("i",2), ("n",2), ("prod",1)]`. The ordering isn't important, but here `("i",2)` has been added to the left of the list returned (presumably using `:`).

Colouring the graph in Figure 3, but now using three colours, proceeds similarly. The original graph has five nodes. We first pick and remove node `"a"`, because it has the (equal) smallest degree, 2, and its `Id` is lexicographically smaller than the other node with degree 2, i.e. `"c"`. Recursively colouring the graph with `"a"` removed yields the colouring `[("b",0),("c",3),("d",2),("n",1)]`. Node `"a"` is connected to `"b"`, `"d"` and `"n"` in the original graph and these have been coloured blank (0), 2 and 1 respectively. This leaves one spare colour, i.e. 3, so we add `("a",3)` to the list returned.

Why is `"b"` assigned the blank colour, 0 (white)? Because when it is removed, leaving nodes `"c"`, `"d"` and `"n"`, those three nodes end up being coloured 3, 2 and 1 respectively. Because `"b"` is connected to all three in the original graph there are no colours left to assign to it.

6 What to do

There are five parts to this exercise. Part V carries no marks and is really hard, so you should only attempt this question if you have completed Parts I–IV and want to compete for the prize.

The `Examples` module contains some examples that you can use for testing. These include the factorial function of Figure 2 (`fact :: Function`) and its various associated structures, and similarly the function in Figure 3 (`fig3 :: Function`). The transformed versions are suffixed with "`Transformed`".

All programs, and the associated data structures, are assumed to be well formed according to the rules of Section 4. Also, all variables referenced will be defined either by virtue of being named parameters of the function or the targets of some initial assignment in the body of the function – there are no undefined variables.

A function `lookUp :: (Eq a, Show a, Show b) => a -> [(a, b)] -> b` is provided in the `Types` module, which you should find useful.

6.1 Part I: Preliminaries

1. Define a function `count :: Eq a => a -> [a] -> Int` that will count the number of occurrences of a given item in a list. For example

```
*Alloc> count 'a' "b"
0
*Alloc> count 3 [2,4,1,3,2,3,3]
3
```

[1 Mark]

2. Using `count`, or otherwise, define a function `degrees :: Eq a => Graph a -> [(a, Int)]` that will compute the degree of each node (type `a`) in a given graph. The degree of a node is the number of edges which either begin or end at that node. For example (`fig1Left` is the graph of Figure 1 (left) without the colours):

```
*Alloc> degrees ([1], [])
[(1,0)]
*Alloc> fig1Left
([1,2,3,4],[(1,2),(1,3),(2,4),(3,4)])
*Alloc> degrees fig1Left
[(1,2),(2,2),(3,2),(4,2)]
```

   The order of the elements in the result list is unimportant.

   [3 Marks]

3. Define a function `neighbours :: Eq a => a -> Graph a -> [a]` that returns the neighbours of a given node in a graph. For example,

```
*Alloc> neighbours 1 ([1], [])
[]
*Alloc> neighbours 2 fig1Left
[1,4]
```

   The order of the elements in the result list is unimportant.

   [3 Marks]

4. Define a function `removeNode :: Eq a => a -> Graph a -> Graph a` that will return the result of removing a given node from a graph. For example,

```
*Alloc> factIG
(["i","n","prod"],[("i","prod"),("n","prod")])
*Alloc> removeNode "i" factIG
(["n","prod"],[("n","prod")])
*Alloc> removeNode "prod" factIG
(["i","n"],[])
```

   The order of the elements in the result list is unimportant.

   [3 Marks]

6.2 Part II: Graph colouring

1. Define a graph colouring function `colourGraph :: (Ord a, Show a) => Int -> Graph a -> Colouring a` that uses the algorithm of Section 5.1 to compute a colouring for a given graph. You may find the `lookUp` function defined in the `Types` module useful. The first argument is the number of colours, equivalently the maximum colour number, and the available colours range from 1 to that maximum. For example,

```
*Alloc> fig1Middle
([1,2,3,4],[(1,2),(1,3),(1,4),(2,4),(3,4)])
```

```
*Alloc> colourGraph 2 fig1Middle
[(2,2),(1,0),(3,2),(4,1)]

*Alloc> fig3IG
(["a","b","c","d","n"],[("a","b"),("a","d"),("a","n"),("b","c"),("b","d"),
("b","n"),("c","d"),("c","n"),("d","n")])

*Alloc> colourGraph 2 fig3IG
[("a",0),("b",0),("c",0),("d",2),("n",1)]

*Alloc> colourGraph 3 fig3IG
[("a",3),("b",0),("c",3),("d",2),("n",1)]
```

The order of the elements in the result list is unimportant. Note that the last example corresponds to Figure 3.

Hint: If you compute the list of node `degrees` in the given graph then the `Id` of the node you need to pick for removal is that of the *minimum* element of the list you get by swapping each `(Id, Int)` pair in that list. The list subtraction operator \\ defined in `Data.List` may also prove useful when assigning a colour to a node.

[8 Marks]

6.3 Part III: Variable renaming

1. Define a function `buildIdMap :: Colouring Id -> IdMap` that uses an interference graph colouring to compute a mapping (`IdMap`) from existing program variables (`Id`) to new ones. The type `IdMap` is defined in the `Types` module thus:

   ```
   type IdMap = [(Id, Id)]
   ```

   If node `v` is coloured 0 (white) then you should build the identity mapping entry `(v, v)`; if the node is coloured $n > 0$ then you should build the mapping entry `(v, "R`n`")`. Because the `"return"` variable is special, the mapping should always include the identity mapping `("return", "return")`. For example,

   ```
   *Alloc> factColouring
   [("i",2),("n",2),("prod",1)]
   *Alloc> buildIdMap factColouring
   [("return","return"),("i","R2"),("n","R2"),("prod","R1")]

   *Alloc> fig3Colouring
   [("a",3),("b",0),("c",3),("d",2),("n",1)]
   *Alloc> buildIdMap fig3Colouring
   [("return","return"),("a","R3"),("b","b"),("c","R3"),("d","R2"),("n","R1")]
   ```

   The order of the elements in the result list is unimportant.

   Hint: you can use the `show` function to generate the string representation of a given integer, e.g. `show 27` yields `"27"`.

   [2 Marks]

2. If one or more of the arguments to a function is mapped to a register then we need to prefix the body of the function with corresponding assignment statements, each of the form `Rn = a` where `a` is the argument name and where `(a, Rn)` is contained in a given `IdMap`. You can see an example of this in Figure 2, where the argument `n` has been mapped to the register variable `R2`. Hence define a function `buildArgAssignments :: [Id] -> IdMap -> [Statement]` that returns a list of such assignments. The `[Id]` is the list of argument names and the assignments should be produced in the same order as the corresponding arguments. For example,

```
*Alloc> idMap1
[("a","a"),("b","R1"),("y","R1"),("x","R6")]
*Alloc> buildArgAssignments ["x","y"] idMap1
[Assign "R6" (Var "x"),Assign "R1" (Var "y")]
```

[2 Marks]

3. Define a function `renameExp :: Exp -> IdMap -> Exp` that will rename the variable references in an expression using a given `IdMap`. For example,

```
*Alloc> e1
Apply Add (Var "a") (Var "b")
*Alloc> renameExp e1 idMap1
Apply Add (Var "a") (Var "R1")
*Alloc> e2
Apply Mul (Apply Add (Var "x") (Const 2)) (Var "y")
*Alloc>  renameExp e2 idMap1
Apply Mul (Apply Add (Var "R6") (Const 2)) (Var "R1")
```

[2 Marks]

4. Using `renameExp` Define a function `renameBlock :: Block -> IdMap -> Block` that will similarly rename the variables in a block. Before returning the renamed block you should first filter out any assignments of the form `v = v`, as discussed in Section 3.2. For example,

```
*Alloc> showBlock factB
0: if (n == 0) {
1:   return = 1;
   } else {
2:   prod = 1;
3:   i = n;
4:   while (i >= 1) {
5:     prod = prod * i;
6:     i = i + -1;
     }
7:   return = prod;
   }

*Alloc> showBlock (renameBlock factB factIdMap)
0: if (R2 == 0) {
1:   return = 1;
   } else {
```

```
2:   R1 = 1;
3:   while (R2 >= 1) {
4:     R1 = R1 * R2;
5:     R2 = R2 + -1;
     }
6:   return = R1;
   }
```

Notice that the assignment `i = n` in the original function gets renamed to `R2 = R2` and has thus been removed as part of the renaming (see Section 3.2).

A function `renameFun` has been defined in the template which uses the above definitions to complete the job of renaming functions:

```
renameFun :: Function -> IdMap -> Function
renameFun (f, as, b) idMap
  = (f, as, buildArgAssignments as idMap ++ renameBlock b idMap)
```

For example,

```
*Alloc> renameFun fact factIdMap == factTransformed
True
*Alloc> renameFun fig3 fig3IdMap == fig3Transformed
True
```

Also `showFun (renameFun fact factIdMap)` should correspond to Figure 2 (right).

[4 Marks]

6.4 Part IV: Interference graph construction

1. Define a function `buildIG :: [[Id]] -> IG` that will build the interference graph from the *live variable sets* of a function. These specify the variables that are live at each point in the body of a function (see Section 3). A "point" here corresponds to an assignment, or to a conditional in an `If` or `While` statement, i.e. to the numbered lines in the functions shown above. For example, there are eight numbered lines in the `fact` function (numbers 0–7) and there is one live variable set for each. These turn out to be:

   ```
   *Alloc> factLiveVars
   [["n"],[],["n"],["n","prod"],["i","prod"],["i","prod"],["i","prod"],["prod"]]
   ```

   It's not important (yet!) to understand all the details of live variable sets, but the predicate in the first line of the `fact` function, `n == 0`, refers only to `"n"` so only `"n"` is live at this point. The second line, `return 0`, contains no variable references, so the live set is `[]`. In the third statement, `prod = 1`, the variable `"n"` is live because it's referred to in a later statement, `i = n`, and no other variables are referenced, other than the target of the assignment. And so on.

   Building an interference graph from the live variable sets is actually quite simple: there is an edge between `v` and `v'` in the graph if there is at least one point where `v` and `v'` are live simultaneously, i.e. there is at least one element of the live variable set that contains both variables. For example `i` and `prod` are both live at points (list indices) 4 and 5, but `n` and `i` are never live together. Hence, for example:

```
*Alloc> buildIG factLiveVars
(["n","prod","i"],[("n","prod"),("prod","i")])
```

A function `sortGraph` for sorting the various components of a graph is included in the `Types` modules and you may find this useful for testing. The example IGs in the `Examples` module are already sorted. For example:

```
*Alloc> sortGraph (buildIG factLiveVars) == factIG
True
```

[2 Marks]

6.5 Part V: Completing the pipeline

This question carries no marks and there are few clues, so good luck! There are two questions here which together complete the register allocation "pipeline". You can pick either, or both, if you have time and want a challenge.

The *control flow graph* (CFG) of a given function contains, for each point in the body of a function:

1. A (*def*, *use*) pair, where "*def*" is the name of the variable assigned at that point and "*use*" is the list of variables used in the expression on the right of the assignment. If the point corresponds to the predicate in an `If` or `While` statement then *def* is taken to be the "dummy" variable `"_"`.

2. The list of successor points, "*succ*", which are integer indexes into the CFG. These are the program points that could possibly be visited next during the function's execution.

For example, after some helpful formatting, the CFG for the `fact` function (type `CFG` in the `Types` module) looks like:

```
*Alloc> factCFG
[(("_",["n"]),          [1,2]),
 (("return",[]),        []),
 (("prod",[]),          [3]),
 (("i",["n"]),          [4]),
 (("_",["i"]),          [5,7]),
 (("prod",["i","prod"]),[6]),
 (("i",["i"]),          [4]),
 (("return",["prod"]),  [])]
```

The program points are implicitly numbered 0–7 and if a statement is the last to be executed the successor list is `[]`.

The set of live variables at line n is given by the solution to the following *dataflow equations*, expressed using mathematical set notation:

$$live(n) = use(n) \cup \left(\bigcup_{s \in succ(n)} live(s) \right) \setminus def(n)$$

where, for example, $succ(n)$ corresponds to the successors of the n^{th} element of the given CFG, e.g. `[5,7]` for the conditional on line 4 of the `fact` function. $S \setminus x$ denotes the set S with element

x removed. Note that $line(n)$ is initially empty for all n and several passes over the CFG may be required to arrive at the solution. The solution is reached when the live variable sets are unchanged in successive passes. Hence, define one or both of:

1. A function `liveVars :: CFG -> [[Id]]` that computes the live variable sets from a given CFG. For example, the n^{th} element of `liveVars factCFG` and `factLiveVars` should contain the same variables, although the order of those variables is unimportant. Each element of `factLiveVars` is sorted, so `map sort (liveVars factCFG) == factLiveVars` should be `True`.

2. A function `buildCFG :: Function -> CFG` that builds the control flow graph for a given function. The order of elements in the *use* and *succ* lists is unimportant. To aid with testing, the `Types` module contains a function `sortCFG :: CFG -> CFG` that will produce a version of a given CFG with these lists sorted. For example, `sortCFG (buildCFG fact) == factCFG` should return `True` (`factCFG` is already sorted).

[0 Marks]