# 2022 Haskell January Test
# **Supercombinators**

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

The **2022 Haskell Programming Prize** will be awarded for the best overall solution, or solutions.

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

Haskell is a higher-order language, which means that functions can be passed to other functions as parameters and returned from them as results. Furthermore, named functions can be defined locally in let expressions and `where` clauses. This creates a problem for a compiler such as GHC: what should it do with *free variables* in user-defined functions, i.e. variables that are in scope in the function's definition but that are not one of the function's parameters? Here is an example (it uses a Haskell let expression, rather than a where clause, but either will do):

```
let
  f x = let
          g y = y + x
        in g
in f 3 8
```

Here, `g` is a function that adds `x` to things, where the `x` is the parameter of the outer function, `f`. We say that the variable `x` is "free" in the body of `g` because it is not one of `g`'s parameters. The variable `y`, which is, is said to be "bound".

Why do free variables present a problem? It's not important that you understand the details, but the crux of the problem is that functions like `g y = y + x` may be applied outside the scope where `x` has a meaning – we sometimes say that the function "escapes" the scope, or context, in which it is defined. You can see this here: in the application `f 3 8`, equivalently `(f 3) 8`, the subexpression `f 3`, is equivalent to an instance of `g` in which `x` is 3. The inner `let` cannot simply return `g` as written; it must also somehow record the fact that it's an instance of `g` in which `x` is "bound to 3".

There are several ways to solve the problem. We could create a copy of `g` at run time and replace the `x` as soon as we know its value, but that's very expensive. Alternatively, we could carry round a reference to `g` as written, coupled with a data structure that somehow records the values of the free variables, but this also gets messy[1].

The preferred solution is to "lift" all functions to the topmost level. To do this we need to augment them with extra parameters corresponding to the free variables in the original body of the function. This transformation is called *lambda lifting*[2] and has the effect of transforming the above program to something like this:

```
let
  $f x = $g x
  $g x y = + y x
in $f 3 8
```

Here you can see that the function `g` has been lifted to the topmost level and renamed `$g`. Functions derived from existing ones will be renamed with a "`$`" to make it clear what's happened. The function *f* has also been renamed (to `$f`), although it's not strictly necessary to do this as it was already defined in the topmost let. Importantly, notice that `$g`'s body no longer has any free variables, as the `x` has now been added as a parameter. To preserve the meaning of the program all references to the original `g` must be replaced by the *partial application* `$g x`, as has been done here throughout. Bingo! There are no nasty free variables, yet the program behaves exactly as before.

[1] The details are unimportant, but the problem is that a function's free variables may be defined in different scopes, so in general you may have to chain together many structures, each storing the values of variables at a given scope.

[2] The name stems from the fact that every function can be treated as an anonymous (unnamed) function, or *lambda expression*.

Lifted functions are "pure" functions in the sense that their bodies refer only to the function's parameters. Such functions are called *supercombinators*[3] and this exercise if about lifting all nested functions in a program to supercombinators at the program's topmost level.
**Recap**: The *arity* of a named function is the number of arguments in its definition, e.g. for `let f x y z = x + z in ...` the arity of `f` is 3. A partial application of a function is an application in which the number of arguments supplied is less than the function's arity. For example, `f 3 4` is a partial application of the above `f` and in this case the result is a function that requires one additional argument before the rule for `f` can be invoked.

Adage: A nice adage that may help you as you work through the exercise is the following:

> *"When leaving a let expression, please take all your belongings with you"!*

# 2 A tiny language

In order to explore supercombinators we'll invent a tiny higher-order, functional language that supports nested scope via let expressions – a bit like a stripped-down version of Haskell.

A program comprises a top-level expression and the value of the program is the value of that expression. There are just five expression types: constants (`Const`), which will be taken to be integers, identifiers (`Id`), which will be represented internally by strings, user-defined functions (`Fun`), function applications (`App`) and let expressions (`Let`).

There are five primitive functions: the first four correspond to Haskell's `+`, `-`, `*` and `<=` operators. The primitive `ite` corresponds to a Haskell conditional, with `App "ite" [p, q, r]` being the equivalent of Haskell's `if p then q else r`.

We will sometimes write programs in an imaginary "source syntax", which looks very much like Haskell syntax, except for primitive function applications which will be rendered in prefix form, e.g. `+ y x` rather than `y + x`. Note that conditionals will be rendered as they would be written in Haskell.

A let expression comprises a set of definitions each binding a variable to an expression. The definitions can be mutually recursive, as in Haskell, so each may depend on the others. In order to simplify the lambda lifter we will assume that user-defined functions only appear in let expressions, i.e. that all user-defined unctions are named. Thus, for example, we will exclude anonymous functions, such as `(\x -> + x 1) 5` as written in Haskell, even though they can be represented using the `Exp` data type; instead we will assume that such functions are named before using them, as in: `let succ x = + x 1 in succ 5`.

Programs (expressions) in our tiny language will be represented by the following types (module `Types`):

```
data Exp = Const Int |
           Var Id |
           Fun [Id] Exp |
           App Exp [Exp] |
           Let [Binding] Exp
         deriving (Eq, Show)

type Id = String
```

[3]It is possible to translate user-defined functions into a small set of predefined "combinators", each of which is a pure function of predetermined size. The prefix "super" stems from the fact that supercombinators are derived from user-defined functions and so, in some sense, are "super-sized".

```
type Binding = (Id, Exp)
```

Primitives are treated as variables, e.g. `Var "+"` represents the primitive addition function. As an example, the expression `let succ x = + x 1 in succ 5` will be:

```
Let [("succ",Fun ["x"] (App (Var "+") [Var "x",Const 1]))]
    (App (Var "succ") [Const 5])
```

This corresponds to test expression `e2` in the module `Examples`. Note that argument names in function definitions, e.g. `"x"` above, and argument expressions in function applications, e.g. `Const 1` above, are represented by lists.

To make testing easier, a pretty-printer called `putExp` is provided in the `Types` module that displays the representation of a function in our assumed source syntax. For example,

```
*SC> putExp e2
let
  succ x = + x 1
in succ 5

*SC> putExp e3
let
  f x = if <= x 1
        then
          1
        else
          * x (f (- x 1))
in f 6
```

Note: `e3` computes the factorial of 6.

There are no booleans in the language, so you can assume that the boolean `true` will be represented by the integer 1 (`Const 1`) and `false` by 0 (`Const 0`).

At this point you are able to answer the questions in Parts I and II, so we'll cover these now before moving on to Part III.

# 3 What to do

There are three parts to the test and you are advised to answer the questions in order, as they get progressively harder.

All expressions are assumed to be well formed throughout: all primitive functions are assumed to be applied to argument expressions of appropriate type and all user-defined functions are assumed to be named via a let expression.

A function `lookUp :: Id -> [(Id, a)] -> a` is provided in the template that will look up the binding of an identifier in a given table. It aborts with a useful error message if no binding is found.

A number of test expressions, `e1`, `e2`, ..., can be found in the `Examples` module. Where appropriate, the lifted form of expression `eN` is included and named `eN'`. Recall that you can pretty-print expression `eN` by typing `putExp eN`.

Note that the pretty-printer will display unnamed functions for testing purposes, even though the language only allows such functions to be bound (named) in a let expression. For example, `putExp e5` will output `Fun x y -> x`.

## 3.1 Part I: Preliminaries

1. Define a function `isFun :: Exp -> Bool` that returns `True` if the given expression is a function (`Fun`) and `false` otherwise. For example:

```
*SC> isFun e2
False

*SC> e5
Fun ["x","y"] (Var "x")
*SC> isFun e5
True
```

**[1 Mark]**

2. Using `isFun`, or otherwise, define a function `splitDefs :: [Binding] -> ([Binding], [Binding])` that will split the definitions in a given list of bindings into two sublists: the function definitions and variable (non-function) definitions. The order of the elements in each sublist should be the same as in the original list. For example:

```
*SC> putExp e4
let
  x = 3
  g y = + y x
in g 8
*SC> e4
Let [("x",Const 3),("g",Fun ["y"] (App (Var "+") [Var "y",Var "x"]))]
    (App (Var "g") [Const 8])
*SC> let Let bs _ = e4 in splitDefs bs
([("g",Fun ["y"] (App (Var "+") [Var "y",Var "x"]))],[("x",Const 3)])
```

**[2 Marks]**

3. Again using `isFun`, or otherwise, define a function `topLevelFunctions :: Exp -> Int` that will count the number of functions in the topmost let expression in a given program (`Exp`). If there is no let expression at the topmost level then the result should be 0. For example,

```
*SC> topLevelFunctions e4
1
*SC> topLevelFunctions e12
2
*SC> topLevelFunctions e20
0
```

**[2 Marks]**

## 3.2 Part II: Free variables

1. The module `Data.List` provides a `union` function on lists which takes two lists, each without duplicates, and forms their "union", i.e. a concatenation of the two lists without duplicates. For example:

```
*SC> union [1, 3] [1, 2, 3, 7]
[1,3,2,7]
```

Define a function `unionAll :: Eq a => [[a]] -> [a]`, a generalisation of `union`, that will return the union of a list of lists, each without duplicates. For example,

```
*SC> unionAll [[2, 5], [4, 2], [5, 1]]
[2,5,4,1]
```

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

**[2 Marks]**

2. Using `union` and `unionAll`, or otherwise, define a function `freeVars :: Exp -> [Id]` that will return the names of the free variables of a given expression. The rules are as follows:

   - A constant has no free variables.
   - A variable is free unless it is the name of a primitive, e.g. `Var "v"` is free but `Var "*"` is not. A list `prims`, which lists the names of the primitives, is defined in the template.
   - The free variables of an application is the union of those in the function and each argument expression. those of the body of the function (an `Exp`) *minus* the names of the function's parameters (see below).
   - The free variables of a user-defined function are those of the body of the function (an `Exp`) *minus* the names of the function's parameters (see below).
   - The free variables of a let expression is the union of the free variables of every expression it contains *minus* the variables bound by the let. This is because the definitions in a let are assumed potentially to be mutually recursive, so each can be "seen" by all the others. (Hint: use the definition of the `Exp` type to remind you where to look.).

   Each call to `freeVars` must return a *set* of variable names, represented as a list without duplicates. Thus "minus" above denotes subtraction over such lists. You can use the operator `\\` from module `Data.List` to implement this, e.g.

   ```
   *SC> [1, 3, 5] \\ [1, 3]
   [5]
   ```

   Then, for example:

   ```
   *SC> freeVars e7
   ["z"]
   *SC> freeVars e8
   ["y","z"]
   *SC> freeVars e9
   ["x"]
   ```

   The order of the elements of the list computed by `freeVars` is unimportant.

   **[7 Marks]**

## 3.3 Part III: Lambda lifting

Although lambda lifting is simple in principle there are some subtleties. Firstly, it is only necessary to lift *functions* to the topmost level. A let expression may also contain other variable bindings and these can stay put, although we must remember to replace any lifted function references with partial applications, as detailed above. For example, given the following variation on the above (`e4` in the `Examples` module):

```
let
  x = 3
  g y = + y x
in g 8
```

the function `g` will be lifted out, but the definition `x = 3` can remain to give the lifted form:

```
let
  $g x y = + y x
in let
     x = 3
   in $g x 8
```

Importantly, note that the reference to `x` in `g y = + y x` must be considered free for the purposes of lifting, despite the fact that is bound in the same let expression as `g`. That's because `g` will eventually end up at the topmost level, and therefore in a different scope to `x`.

Secondly, we must remember that the functions defined in a let expression may be mutually recursive, so the extra parameters added to each function in that expression must be the *union* of the free variables of those functions. However, references to other functions defined in the same let expression must be considered bound (not free) because those functions will be lifted together to the topmost level where they will sit in the same scope. You can think of the set of functions within a let expression as "siblings" in this sense. Recall that references to all other variables defined in the let must be considered free, as explained above.

To see why it's important to take the union of the functions' free variables, consider the following variation on the factorial example above, which is defined using two mutually-recursive functions and with reference to additional let-bound variables, `y` and `z`:

```
*SC> putExp e19
let
  y = 1
in let
     z = 1
     f x = if <= x y
           then
             y
           else
             g x
     g x = * x (f (- x z))
   in f 6
```

The problem is that `f` refers both to `y`, which is free, and `g`, which must be considered bound, as explained above. However, a similar thing can be said of `g`, where the variable `z` is free but `f` isn't. If we just include the extra parameter `y` when lifting `f` and, similarly, `z` when lifting `g`, then we would end up with the program on the left:

```
let
  $f y x
    = if <= x y
      then
        y
      else
        ($g z) x
  $g z x
    = * x (($f y) (- x z))
in let
     y = 1
   in let
        z = 1
      in ($f y) 6
```

```
let
  $f y z x
    = if <= x y
      then
        y
      else
        ($g y z) x
  $g y z x
    = * x (($f y z) (- x z))
in let
     y = 1
   in let
        z = 1
      in ($f y z) 6
```

This is wrong, because there is no `z` in scope in the body of `$f`; likewise `y` in `$g`. By taking the union of the free variables of both functions instead we ensure that `$f` carries the extra parameter `z`; similarly `y` in the case of `$g`. The effect can be seen on the right above[4]. Note that the extra parameters must all appear first and the various test cases used in this exercise assumes they are sorted alphabetically (see below). The order isn't important for correctness, however.

Finally, it turns out that name clashes can cause a problem[5] but we will assume that all functions and/or parameters have previously been re-named so we will never encounter the problem.

A function `lambdaLift` is defined in the template that will transform a program into its lambda-lifted form, so that all functions sit at the topmost level in the form of supercombinators:

```
lambdaLift :: Exp -> Exp
lambdaLift e
  = lift (modifyFunctions (buildFVMap e) e)
```

The lifting is thus performed in three phases, involving three separate passes over the given program. Your job is to implement each of these phases.

1. Define a function `buildFVMap :: Exp -> [(Id, [Id])]` that will build a table mapping function names (`Id`) to the names of the free variables of those functions (`[Id]`). The free variables of a function should be those computed by `freeVars` but *excluding* the names of any sibling functions defined in the same let expression. This is because the siblings will always sit in the same scope, even if/when they are lifted to the topmost level. You can use the \\ function to implement the exclusion. You should also find the `splitDefs` function from Part I useful. Thus, for example:

   ```
   *SC> putExp e1
   let
     f x = let
             g y = + y x
           in g
   in f 3 8
   *SC> buildFVMap e1
   ```

[4]This strategy may not always produce the most optimal code, but it yields correct code and is sufficient for this exercise.

[5]For example, if `f`'s parameter above were named `z`, rather than `x` then we'd end up trying to add `z` to its parameters when there's already one there.

```
[("f",[]),("g",["x"])]

*SC> buildFVMap e2
[("succ",[])]

*SC> buildFVMap e19
[("f",["y","z"]),("g",["y","z"])]
```

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

Note: In the various test expressions the free variable references added to function definitions and function applications during lambda lifting are sorted alphabetically. For consistency, you might therefore wish to apply `sort` from `Data.List` before adding them to your mapping table.

Hint: Remember that functions are defined via let expressions so you need to search every `Exp` in the given program. Again, use the data type definition to remind you where to look.

**[6 Marks]**

2. Define a function `modifyFunctions :: [(Id, [Id])] -> Exp -> Exp` that will use a free variable mapper (e.g. from `buildFVMap`) to modify every user-defined function definition and every function reference in a given program, as follows.

   - When you encounter a definition of the form `(f, Fun as e)` in a let expression you should use the mapper to look up the function's free variables and add those to the *front* of `as`. The function should be also be renamed by prefixing the name with a `'$'` to indicate that it is now a supercombinator. Don't forget that the body of the function (an `Exp`) must also be modified.
   - When you encounter a reference to a function, e.g. `Var f`, where `f` has a binding in the mapper, you should replace the reference with a partial application of the renamed function (again, prefix it with a `'$'`) to its free variables, which are given by the mapper. The only exception is when the function has no free variables, in which case you should just return a reference to the renamed function[6].
   - Constants should be returned unmodified and applications should be modified by applying the `modifyFunctions` to the function and argument expressions recursively.
   - You do not need a separate rule for functions (`Fun`) as these will only appear (named) in let expressions and so will be covered by the above.

   For example,

```
*SC> mapper = buildFVMap e1
*SC> putExp (modifyFunctions mapper e1)
let
  $f x = let
           $g x y = + y x
         in $g x
in $f 3 8

*SC> putExp (modifyFunctions (buildFVMap e19) e19)
```

[6] It's not wrong to add a partial application in this case - it's just not necessary.

```
let
  y = 1
in let
     z = 1
     $f y z x = if <= x y
                then
                  y
                else
                  ($g y z) x
     $g y z x = * x (($f y z) (- x z))
   in ($f y z) 6
```

**[6 Marks]**

3. Finally, define a function `lift :: Exp -> Exp` that will lift all combinators to the topmost level. You can implement this in any way you wish, but the suggestion is that you use a helper function `lift' :: Exp -> (Exp, [Supercombinator])` that, given an expression, will return a version of the expression with all supercombinator definitions removed, together with a separate list containing those definitions. A `Supercombinator` here is simply a named function, i.e. a binding between a function name and a function expression (`Fun`). It's therefore just an example of a `Binding`, which is already defined in the `Types` module, hence:

   ```
   type Supercombinator = Binding
   ```

   There is one final optimisation: when building a let expression anywhere, if there are no remaining local definitions then the let expression can be replaced by its body expression, i.e. `Let [] e` can be replaced by `e`. Thus, for example:

   ```
   *SC> topLevelFunctions e1
   1
   *SC> topLevelFunctions (lambdaLift e1)
   2
   *SC> putExp (lambdaLift e1)
   let
     $f x = $g x
     $g x y = + y x
   in $f 3 8

   *SC> lambdaLift e4 == e4'
   True

   *SC> lambdaLift e14 == e14'
   True

   *SC> putExp (lambdaLift e19)
   let
     $f y z x = if <= x y
                then
                  y
   ```

```
            else
              ($g y z) x
  $g y z x = * x (($f y z) (- x z))
in let
     y = 1
   in let
        z = 1
      in ($f y z) 6
```

Note, for example, that `($f y z) 6`, whose representation is `App (App (Var "$f") [Var "y", Var "z"]) [Const 6]`, is equivalent to `$f y z 6`, whose representation is `App (Var "$f") [Var "y", Var "z", Const 6]`. Either will do, although the use of the mapping table, as described, will naturally produce the former (the `f` in the original program gets replaced with `$f y z`).

Good luck!

**[4 Marks]**