# G6021: Comparative Programming

Exercise sheet 8

## 1 Imperative Language

The goal of this exercise is to write, in Haskell, an interpreter for a simple imperative programming language. This language has variables, assignments, conditionals and while loops.

The salient feature of imperative languages is that they use a memory that is updated by the program. We begin by modelling the memory in Haskell. A memory location will be represented as a pair: the name of the variable, and the contents (a number). We can then model the memory as a list of pairs.

1. Give types `Name` and `Memory` that can represent variable names and the memory respectively.
2. Write a function `update :: Name -> Integer -> Memory -> Memory` that given a variable name, a value and a memory, will update the memory with the new value. This function should replace any previous value. If there is no previous value, then the memory should be extended with the values given.
3. Write a function `find :: Name -> Memory -> Integer` that will return the contents associated with a memory location. If the variable does not exist in the memory, return 0.



Next we model arithmetic and Boolean expressions. An arithmetic expression can be a number, a variable, the sum of two expressions, or the product of two expressions. They are given by the following syntax:



```
E ::= Number | Var | E + E | E * E
```

A Boolean expression can be one of: the conjunction ("and") of two Boolean expressions, equality test, or less-than test, for two arithmetic expressions, or the negation of a Boolean expression. The following is the syntax for Boolean expressions:

```
B ::= B and B | E = E | E < E | not B
```

We could include Boolean constants (True and False) also in this language. Try to add these if you have time.

4. Define a datatype `Aexp` for arithmetic expressions.
5. Write a function `evalA :: Aexp -> Memory -> Integer` that will evaluate an arithmetic expression, using the values of variables in the memory.
6. Define a datatype `Bexp` for Boolean expressions.
7. Write a function `evalB :: Bexp -> Memory -> Bool` that will evaluate a Boolean expression, using the values of variables in the memory.

Next we model commands. There are four constructs in this language: assignment, sequence of commands, conditionals and while loops. The following grammar formalises this:

```
C ::= Var = A | (C ; C) | If B then C else C | While B do C
```

Brackets are used to group commands in the syntax, but we will not write them if we can avoid doing so.

8. Define a datatype `Com` to represent Commands.

9. Write a function `evalC :: Com -> Memory -> Memory` that will take a command and a memory, and evaluate the command in the memory given, returning the new memory. The output of the interpreter is therefore the final memory. For example, the program:

   ```
   x = 5; z = 4
   ```

   executed in the empty memory should give a memory where `x` has the value 5, and `z` the value 4.

10. Your interpreter is now complete. Test with the following programs, converting them to the datatype and using the empty list for the initial memory.

    ```
    x = 1; y = x; x = z

    z = 5 ; x = 4 ; if z < x then y = z else y = x

    z = 5 ; x = 3 ; while x < z do (y=y+1 ; x=x+1)
    y =1; z = 5 ; x = 1 ; while not(z < x) do (y=y*x ; x=x+1)
    ```



## 2 Functional Language

The goal of the second exercise is to build an interpreter for the simplest functional language (the $\lambda$-calculus), using Haskell. Recall the $\lambda$-calculus is given by the following syntax:

$$M ::= var \mid (\lambda x.M) \mid (MN)$$

which are called variables, abstraction and application respectively. All programs that we evaluate will be closed terms (no free variables), and we will reduce to weak head normal form (we do not reduce under abstractions).

Reduction is given by $(\lambda x.t)u \to t\{x \mapsto u\}$, where $t\{x \mapsto u\}$ means substitution: replace all the free occurrences of the variable $x$ in $t$ by $u$. For the purpose of this exercise we can formulate substitution by the following:

$$\begin{array}{lcl} x\{y \mapsto V\} & = & V \text{ (if } x = y) \\ x\{y \mapsto V\} & = & x \text{ (if } x \neq y) \\ (\lambda z.M)\{y \mapsto V\} & = & \lambda z.M \text{ (if } z = y) \\ (\lambda z.M)\{y \mapsto V\} & = & \lambda z.(M\{y \mapsto V\}) \text{ (if } z \neq y) \\ (MN)\{y \mapsto V\} & = & (M\{y \mapsto V\})(N\{y \mapsto V\}) \end{array}$$

1. Define a datatype `Lam` to represent the terms in the $\lambda$-calculus. Variable names should be represented by the type `Name = [Char]`.

2. Define a type `Set` that will represent a set of variable names of a $\lambda$-term, and give three functions:

   (a) `add :: Name -> Set -> Set`
   that will add a name to a set.

   (b) `join :: Set -> Set -> Set`
   that will return the union of two sets.

(c) `remove :: Name -> Set -> Set`
that will remove a name from a set, it the name exits (otherwise, just return the set).

3. Write a function to compute the set of free variables of a $\lambda$-term. Include in your answer the type of this function.

4. Write a function that tests if a $\lambda$-term is a closed term. Include in your answer the type of this function.

5. Write a function `subst :: Name -> Lam -> Lam -> Lam` that will implement substitution as defined above.

6. Write a function `eval :: Lam -> Lam` that will reduce a $\lambda$-term to weak head normal form using any strategy.

7. Write the following $\lambda$-terms using your `Lam` datatype, and test your evaluator by evaluating them.
   $(\lambda x.x)(\lambda x.x)$
   $(\lambda fx.f(fx))(\lambda x.x)$
   $(\lambda fx.f(fx))(\lambda x.x)(\lambda x.x)$
   $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$