Assignment 4

COMP 302 Programming Languages and Paradigms
Prakash Panangaden

Due Date: 21st March 2016

This assignment has two programming questions and two questions for which we expect written answers. Put the programming assignment in a file with the extension .fs as usual. Put the other two questions in a **pdf** file.

In the first two questions we will implement part of a type checker for a new made-up language called Monty. In this assignment we are looking exclusively at the part of Monty that deals with its type system. We will define an F# type called `typExp` to encode the types of Monty. Note that Monty has a polymorphic type system.

We will use this to implement unification which is part of the *type reconstruction* algorithm for Monty. Let us recap basic facts about unification first. Unification is one of the central operations in type-reconstruction algorithms, theorem proving and logic programming systems. In the context of type-reconstruction, unification tries to find an instantiation for the free variables in two types τ_1 and τ_2 such the two types are syntactically *identical*. If such an instantiation exists, we say the two types τ_1 and τ_2 are *unifiable*.

Here is the definition of the types of Monty. Remember we are writing a program in F# that deals with another language (Monty). Do not confuse the types of the language Monty with F# types. The type definition below is written in f# it *describes* the types of Monty.

```
type typExp =
  | TypInt
  | TypVar of char
  | Arrow of typExp * typExp
  | Lst of typExp
```

We want to implement unification for Monty type expressions. This means finding a substitution for the type variables. A substitution is a list of pairs; each pait gives a type expression for a type variable. We say that two type expressions are *unifiable* if there is a substitution that makes them identical. This is key step in the type reconstruction algorithm.

We are using simple characters to represent type variables with the constructor TypVar prefixing it. In the substitution we just write the character and not this constructor but this is just for convenience. I have only got one base type (TypInt) and one unary type constructor (Lst) and one binary type constructor (Arrow). A type that we would write as $int \to (`a \to `b - list)$ would be written in our representation as

```
Arrow(TypInt, Arrow(TypVar 'a', Lst (TypVar 'b')))
```

The above is an example of a polymorphic type expression. Here is another one:

```
Arrow(TypVar 'a', Arrow (TypInt, Lst (TypInt)))
```

These are, of course, not the same. Are they unifiable? In other words, is there some replacement for the type variables that makes these two type expressions identical? Yes there is! If we replace both the type variables with TypInt they will both be identical. This is the kind of thing that we will write our program to discover. Here is a script of the program in action.

```
> let te3 = Arrow(TypInt, Arrow(TypVar 'a', Lst (TypVar 'b')));;
val te3 : typExp = Arrow (TypInt,Arrow (TypVar 'a',Lst (TypVar 'b')))
> let te4 = Arrow(TypVar 'a', Arrow (TypInt, Lst (TypInt)));;
val te4 : typExp = Arrow (TypVar 'a',Arrow (TypInt,Lst TypInt))
> let unifier = unify te3 te4;;
val unifier : substitution = [('b', TypInt); ('a', TypInt)]
> applySubst unifier te3;;
val it : typExp = Arrow (TypInt,Arrow (TypInt,Lst TypInt))
> applySubst unifier te4;;
val it : typExp = Arrow (TypInt,Arrow (TypInt,Lst TypInt))
```

Notice that this is more than simple pattern matching. Each type expression constrains the other. We are finding the *most general unifier.*

Question 1[30 points]

In this question you will implement some of the auxiliary functions needed for unification. First of all recall that we do not allow a substitution where a variable is replaced by a type expression containing it. So we need to check if a type variable occurs in a type expression. This is called the "occur check." Before we call occur check we strip off the TypVar constructor so we need a function of the following type.

```
occurCheck : v:char -> tau:typExp -> bool
```

A substitution is defined as follows.

```
type substitution = (char * typExp) list
```

Now we want a function that performs a replacement of a variable with a type expression. This should have the following type.

```
val substitute : tau1:typExp -> v:char -> tau2:typExp -> typExp
```

This replaces all occurrences of the type variable `TypVar v` with the type expression `tau1` in the type expression `tau2`.

The above function just does one replacement. A substituion is a whole list of these so we need another function called `applySubst`

```
val applySubst : sigma:substitution -> tau:typExp -> typExp
```

Question 2[35 points]

In this question you are asked to implement a function `unify` which checks whether two types are unifiable and produces the unifier if there is one. It should return the most general unifier or fail with an appropriate error message.

```
val unify : tau1:typExp -> tau2:typExp -> substitution
```

I have written a template file which you can find on the course web site. There are also some more examples shown there.

Question 3[20 points] Informally *derive* the type for the apply-list function defined below. This function takes a list of functions and produces a single function which is the composite of all of them. If the list is empty it returns the identity function.

The code is shown below.

```
let rec apply_list l =
  match l with
  | [] -> (fun x -> x)
  | f::fs -> (fun x -> apply_list(fs)(f x))
```

Question 4[15 points] When the following expression is evaluated; the actual result is 7, as shown in the execution script below. Draw environment diagrams showing the bindings just as

1. the `let u =2` block is opened,
2. the execution of `f(4)` is about to begin and
3. the execution of the *body* of the function is about to begin.

```
let x = 1 in
let f =
  let x = (let u = 2 in u + x) in
    fun z -> x + z
f(4);;

val x : int = 1
val f : (int -> int)
val it : int = 7
```

Yes, this is the mid-term question. This time I want you to do it perfectly!