The University of Melbourne

Department of Computing and Information Systems

COMP30020/COMP90048

Declarative Programming

Semester 1, 2013

Exam Duration: Two hours.

Reading Time: Fifteen minutes.

Length: This paper has 4 pages including this cover page.

Authorised Materials: None.

Instructions to Invigilators: Each student should be supplied with a script book.

Instructions to Students: Answer all of the questions in the script book provided. Start your answer for each question on a separate page and write the question number at the top of the page. The number of marks for each question (totaling 60) is given; where no breakdown is given for multi-part questions, each part is worth the same number of marks. Minor errors in Haskell and Mercury syntax etc, will be tolerated but you should add comments if you feel it will aid understanding of what you have written.

Calculators: Calculators are not permitted.

Baillieu Library: Exam paper to be held by the library.

Question 1 [6 marks]

What would be printed if the following expressions are typed into `ghci`? Recall `:t` tells `ghci` to just print the type of the expression. If any expressions would result in errors, give the general nature of the error, for example "type error", rather than detailed error messages.

(a) `:t (+3)`

(b) `:t foldr`

(c) `:t return`

(d) `filter (\x->True) [2]`

(e) `[Nothing] ++ [Nothing]`

(f) `[Just Just]`

Question 2 [4 marks]

For each of the following pairs of terms, give the most general unifier, or state the unification fails.

(a) `f(A, g(B), 3)` and `f(1, g(2), C)`

(b) `f(1, g(A), A)` and `f(1, B, 3)`

(c) `f(A, B, A)` and `f(1, g(1), g(3))`

(d) `f(1, A, C)` and `f(A, B, g(3))`

Question 3 [8+2=10 marks]

Consider the following C and Haskell code:

```
// find sum of values in key-value list
    sum = 0;
    while(x) {
        sum += x->val;
        x = x->next;
    }

-- sum of values in list of (key,value) pairs
val_sum kvs = foldr (+) 0 (map snd kvs)
```

(a) With reference to the code above, describe three ways in which Haskell programs tend to re-use definitions more than C programs.

(b) With reference to the code above, discuss one disadvantage of Haskell compared to C.

Question 4 [5+2+2+2+2+2=15 marks]

Often in compilation we are interested in taking a nested expression and "flattening" it into several simple expressions which have no nesting. For example, the Haskell expression `v0*v1+1.5` is equivalent to `let v2=v0*v1; v3=v2+1.5 in v3`. This transformation requires introducing new variables (`v2` and `v3`), each appearing on the left side of one equation. The expression on the right side of each equation contains a single operation, and the final expression (`v3`) contains none.

As well as an input expression, we need to know the first number to use for the new variables, and it is convenient to also return the "next" number which can be used (one more than the largest new variable number used in the generated equations). In the example, the first number is 2 and the next number is 4. Below we give appropriate type definitions for simple cases, a type signature for the `flatten` function and how the example above would be represented:

```
data Expr = Var Int | Num Double | BinopExpr Binop Expr Expr
data Binop = Plus | Minus | Times | Div
data Eqn = Eqn Int Binop Expr Expr -- the Int is the var number on LHS

flatten :: Expr -> Int -> (([Eqn], Expr), Int)

flatten (BinopExpr Plus (BinopExpr Times (Var 0) (Var 1)) (Num 1.5)) 2
 = (([Eqn 2 Times (Var 0) (Var 1),Eqn 3 Plus (Var 2) (Num 1.5)],Var 3),4)
```

(a) Given a Haskell definition of the function `flatten` described above.

(b) Instead of flatten returning values of type `(([Eqn], Expr), Int)` (a pair containing a pair), it could use a triple instead: `([Eqn], Expr, Int)`. Other than simplicity, very briefly give one advantage of using triples instead of nested pairs.

(c) Very briefly, give one advantage of returning a pair containing a pair.

(d) In Mercury, the following predicate could be used instead of the flatten function (where types `expr` and `eqn` are defined appropriately):

```
:- pred flatten(expr, int, list(eqn), expr, int).
```

Given an appropriate Mercury declaration which describes the mode and determinism of this predicate.

(e) The Mercury predicate could also have the arguments in a different order, as follows:

```
:- pred flatten(expr, list(eqn), expr, int, int).
```

Briefly give an advantage of this argument order.

(f) The questions above illustrate how multiple output arguments in Mercury can avoid creation of pairs etc. which would be necessary in Haskell. Briefly give two other examples where Mercury can avoid creating data structures which would be necessary in Haskell.

Question 5 [5+5=10 marks]

This question is about design of data types. Suppose we want to represent numeric and Boolean expressions (unrelated to the previous question). One possibility is to use the following data type (for simplicity we include only a limited number of operators).

```
data Ex = ExNum Double | ExT | ExF |  -- numbers, True, False
    ExVar String | ExOp Op [Ex]       -- variables, operators+args
data Op =
    -- unary operators:
    UMinus | Not |          -- numeric/Boolean, respectively
    -- binary operators:
    Plus | Minus | Times |  -- numeric
    LT | EQ | LE |          -- numeric comparisons (returns Boolean)
    And | Or |              -- Boolean
    -- ternary operator:
    ITE -- for representing "if BoolExp then NumExp1 else NumExp2"
```

(a) When designing data types, we typically have a choice about how general or flexible our representation is — how many different things can be represented by the data type(s). Assuming our aim is to avoid errors in our code (or detect any errors as quickly as possible), briefly discuss how general we should attempt to make our representations — for example, should we use more general or less general representations, and why?

(b) Suggest an alternative to the `Ex` and `Op` types above which may lead to fewer errors in code which manipulates expressions. State any assumptions you make.

Question 6 [3+4+6+2=15 marks]

Consider the following cord data type:

```
data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)
```

(a) Write a Haskell function `c_len :: Cord a -> Int` which returns the number of elements in a cord. Attempt to make the code as simple as possible.

(b) The `map` and `foldr` functions over lists can be adapted in a uniform way for many other data types (as discussed in lectures and workshop exercises). Write a function `c_foldr`, including a type signature, which adapts `foldr` to cords.

(c) Write a Haskell function `cl_foldr` which computes the `foldr` of the list corresponding to a cord. Note that the type of `cl_foldr` may not be the same as the type of `c_foldr` above. It should be equivalent to the following definition, where `cord_to_list` is defined separately and converts from a cord to a list:

```
cl_foldr' f b c = foldr f b (cord_to_list c)
```

However, your definition must avoid creating the intermediate list (it must not use `cord_to_list`).

(d) Write a definition of `cord_to_list` in terms of your `cl_foldr` function and state its worst case time complexity.