Principles of Programming Languages - Homework II (Solutions)

K. Subramani
LCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. Consider the following fragment of **C** code.

```
union{
   int x;
   float y;
   } a;

a.x = 7;
a.y = 8.3;
```

   Without implementing this code fragment, indicate, whether there are any errors. Now, implement this code fragment on your favorite **C** compiler and describe your observations.

   Solution: Ideally, an implementation *should* flag the above code as an **error**, since in a **union** variable, only one of the constituent objects can be accessed. However, this is very difficult to implement in a programming language; in particular, **C** compilers allow you to both compile and run the above code, without any errors. Attempting to print $a.x$, results in garbage output, whereas the code indicates that a valid value is stored in the corresponding memory location. □

2. Describe with examples the concepts of aliasing and side effects.

   Solution: The binding of an object (memory location) to two or more different names is called *aliasing*. For instance, in the code fragment below, the memory location pointed to by x is also pointed to by y.

```
int *x, *y;

x = (int *) malloc(sizeof(int));
*x = 1;
y = x;
```

   Side effects can be defined in various ways. A common definition of a side effect of a statement is any change in the value of a variable that persists beyond the execution of the statement. More generally, side effects can be defined as unintended consequences of statement execution. In the above code segment, the statement $*y = 2$ changes $*x$ too, whether or not it is intended. Typically, such changes are difficult to track and should be avoided. □

3. Explain with examples the concepts of static and dynamic scoping. What is the primary problem posed by dynamic scoping?

   Solution: If the scope of a binding is limited to the block in which its associated declaration appears and other blocks enclosed within it, then the scoping is said to be static. Most imperative languages, such as **C**, Pascal and Ada use static scoping rules, in order to determine the scope of a binding. Further, in static scoping, the binding of a name within a block takes precedence over its binding in blocks within which it is enclosed. For instance, in the code below, the name x has a global scope, i.e., it can be seen by all functions (except *r()*), whereas y is an *int* variable in function *p()* and a *char* variable in function *q()*. In *r()*, the name x is a *float* variable, as per static scoping.

```
int x;

void p (void)
 {
    int y;

 }

 void  q (void)
  {
    char y;

  }

 void  r (void)
   {
    float r;

   }
```

   In dynamic scoping, declarations are processed as they are encountered along an execution path of a program. Accordingly, different symbol tables can be created for the same code fragment, depending on how execution actually occurred. For instance, in the code fragment below, the *change()* function will alter the binding of x in *p()* as per dynamic scoping, whereas as per static scoping, the global variable x will be altered.

```
int x;

void  p (void)
  {
    char x;
    change();
  }

void  change (void)
  {
    x++;
  }
```

   The principal problem with dynamic scoping is *lack of predictability*, in that one cannot *a priori* deduce what the correct output should be. □

4. Consider the following declaration in **C** syntax:

```
struct CharTree {
    char data;
    struct CharTree left, right;
}
```

   Identify the principal problem with this declaration and rewrite it using the **union** technique described on Pages 213-214 of [Lou02]. How would you modify the fragment, so that it actually compiles?

   Solution: The principal problem with the above declaration is that it is not finite, akin to a recursive function call, without a base case. We could make the declaration terminate as follows:

```
union CharTree {
   enum {nothing} emptyCharTree;
   struct {
     char data;
     union CharTree left, right;
          } charTreeNode;
    };
```

   However, **C** still cannot compile the code and we need to use pointers to enable compilation.

```
struct CharTree {
  char data;
  CharTree *left, *right;
 };
```

   □

5. What are the typical kinds of type equivalences that you would expect to see in a programming languages? Give one example of each.

   Solution: The type equivalence notions in a typical programming language are as follows:

   (a) Structural Equivalence - Two types are said to be structurally equivalent, if they are built in exactly the same way, using the same type constructors from the same simple types. For example,

```
struct Rec1 {        struct Rec2 {         struct Rec3 {
   char x;              char x;               int y;
   int y;               int y;                char x;
};                   };                    };
```

   struct Rec1 is structurally equivalent to *struct Rec2* and neither of them is structurally equivalent to *struct Rec3*.

   (b) Name Equivalence - Two types are name equivalent if and only if they have the same name. Thus,

```
typedef int nint;
int a;
nint b;
int c;
```

   Variables a and c have the same type, whereas variable b has a type that is not equivalent to the type of a and c.

   (c) Declarative equivalence - Two types are equivalent if

      i. Both are name equivalent *structs* or name equivalent *unions*, or

      ii. Neither is a *struct* or *union* and they are structurally equivalent.

For instance,

```
   struct Rec1 {       struct Rec2 {
     char x;             char x;
     int y;              int y;
  };                  };
```

struct Rec1 and *struct Rec2* are not declaratively equivalent. However, in

```
    typedef int nint;
```

nint and *int* are declaratively equivalent.

□

References

[Lou02] Kennenth C. Louden. *Programming Languages: Principles and Practice*. Brooks/Cole, 2002.