Name: Milson Munakami

Student ID: 114012811

Course: CS 354 Programming Languages

Instructor: Dr. Jim Buffenbarger

Assignment #3: Control Flow, Data Types, and Subroutines

Exercise 6.1

I think these statements are not contradictory. As we know, precedence and associativity rules define the order in which binary arithmetic operators are applied within an expression but they do not specify the order in which the operands of a given operator are evaluated. The most compilers are free to evaluate the operands of a binary operator in either order unless the operation does not affect the outcome, the compiler is free to choose the most efficient way to perform the operation.
For example, left Associativity of arithmetic add operator in this expression a(x) + b(y) + c(z) makes the compiler to rearrange it as (a(x) +b(y)) +c(z) due to left associativity of + operator but it doesn't determine whether compiler calculate a(x) or b(y) as it doesn’t matter to the end result.

Exercise 6.23

We could accomplish the same task using do (repeat) loop as follows:

```
do {
  line = read_line();
  if (!all_blanks(line)) {
                    consume_line(line);
            }
} while !(all_blanks(line));
```

As we can see there is duplicate part of the code, we can omit this using a flag variable as:

```
bool complete = false;
do {
  line = read_line();
  if (all_blanks(line)) {
                    complete = true;
} else {
                    consume_line(line);
}
} while (!complete);
```

Comparision: Clearly, both these above mentioned alternatives require introduction of extra lines of code or variable than the midtest version. It omits the break statement so it will be more clear and well-structured than the midtest version.

Exercise 6.24

The **goto** statement is considered harmful as it can produce unmaintainable code.

We can achieve same functionality using the following code but as you can see we need more conditional breaks on the code.

```
int first_zero_row = -1; /* none */
int i, j;
for (i = 0; i < n; i++) {
    int all_zeroes = 1;
    for (j = 0; j < n; j++) {
        if (A[i][j]) {
            all_zeroes = 0;
            break;
        }
    }
    if (all_zeroes) {
        first_zero_row = i;
        break;
    }
}
```

Exercise 7.1

Name equivalence is used to check the compatibility of type, independent of the contexts and implementation. It simplifies the type testing. A type get exactly one definition. Modern programming languages use some form of name equivalence for types instead of structural equivalence because it is safer and cleaner as everything has exactly one definition. It simplifies the implementation of type checking so that it makes it possible for the compiler to reject code that accidentally mixes different data types. Two data types are structurally equivalent if they have the same public members and define the same interface. The advantage of structural equivalence is that the compiler doesn't need any mechanism to ensure whether code is using the same shared type.

Exercise 7.2

Structural equivalence says all variables have same type. Strict name equivalence equates types of A and B, because they refer back to the same type declaration but C, D are incompatible. Under Loose name equivalence A, B, and C are all mutually compatible as they refer to the same outermost constructor i.e. they refer to the same declaration after factoring out any type aliases. But D is not compatible to any of these variables.

Exercise 7.8

It will consume 240 bytes. Consider short labelled as s will take an address N so s takes up the space from N up to N + 2 (multiple of 2 i.e. size for s), then c takes up one byte, from N + 2 to N + 3. At this point we cannot start t there, because N + 3 is odd (since N is even). So we must force up 1 by alignment. Thus t ranges from N + 4 to N + 6; similarly d from N + 6 to N + 7; r from (forced up 1 by alignment to make it multiple of 8) N + 8 to N + 16; i from N + 16 to N + 20 so we need at least 20 bytes for this record. We need to insert 4 bytes of additional padding at the end of each element to make size as multiple of 8.

So, I conclude that A contains 10 elements with 24 bytes each and thus 24 * 10 = 240 bytes for the array.

The final structure will look like this:

s s c | t t d | r r r r r r r r i i i i | | | | = 24 bytes for each element

Exercise 8.3

The following C application demonstrates the order in which subroutine parameters are evaluated in *evaluate* method. But the evaluation is determined by the Compiler and can't be rely on.

```
#include <stdio.h>

static void foo (int a, int b, int c) {}
static int evaluate(int n, int order){
   printf("Argument %d executed in Order: %d \n", n, order);
   return n;
}
int main(void)
{
   foo (evaluate (0, 1), evaluate (1, 2), evaluate (2, 3));
   return 0;
}
```

Exercise 8.4

Many operating systems ensures that different instances of foo will occupy same space in the stack. As, local variable i is not never initialized, if the stack space has not been used for anything else in the meantime it may use the value from the previous instance of foo. If system creates the stack from space filled by zeros and if the space occupied by foo's activation record is not used by anything else before it is called, then i will start with the initial value zero. But, it cannot be always certainly guaranteed so i can take any garbage value if it is preoccupied and hence can show nondeterministic behavior.

Exercise 8.6

This is a simple application in Java to demonstrate the different modes of parameter passing.

```
int x;
public static void main(String[] args)
{
        x = 1;
        callFuntion(x);
        System.out.println(x);
}

public void callFuntion (int y)
{
        y += 2;
        System.out.println(x + ", ");
}
```

For parameter **pass by value**, the output is 3, 1 (default behavior of Java program)
For parameter **pass by reference**, the output is 3, 3
For parameter **pass by value/result**, the output is 1, 3 (similar to Pass by Reference difference is the value is assigned back to the actual.)
For parameter **pass by name**, the output is 3, 3 (need to use macro to implement)

Exercise 8.9

We should not be confused between parameters and variables; they are very different things. As a parameter, x represents just the number 2, not some address. In FORTRAN, the parameter attribute is used to define a named constant. If we use the named constant as a variable, then any assignment to the dummy argument modifies the value of a named constant. Obviously, we can't modify the value of a constant. The named constants do not have addresses in Fortran IV implementations. To solve this problem, in some other implementations, named constants have addresses but they are located in write-protected (**read-only**) areas of memory. If we attempt to modify them, the compiler will show weird and fatal run-time errors.